CSE 234
Data Systems for Machine Learning

Arun Kumar

Topic 2: Deep Learning Systems

DL book; Chapters 5 and 6 of MLSys book
Academic ML 101

Generalized Linear Models (GLMs); from statistics

Bayesian Networks; inspired by causal reasoning

Decision Tree-based: CART, Random Forest, Gradient-Boosted Trees (GBT), etc.; inspired by symbolic logic

Support Vector Machines (SVMs); inspired by psychology

Artificial Neural Networks (ANNs): Multi-Layer Perceptrons (MLPs), Convolutional NNs (CNNs), Recurrent NNs (RNNs), Transformers, etc.; inspired by brain neuroscience

Deep Learning (DL)
Real-World ML 101

DL Systems in the Lifecycle

Data Scientist/ML Engineer

Source → Build → Deploy

ML/AI + Data Systems Infrastructure

Data acquisition
Data preparation

Feature Engineering
Training & Inference
Model Selection

Serving
Monitoring
Figure 1: Only a small fraction of real-world ML systems is composed of the ML code, as shown by the small black box in the middle. The required surrounding infrastructure is vast and complex.
Evolution of Scalable ML Systems

1980s
- In-RDBMS ML Systems
- Scalability

Mid 1990s
- ML on Dataflow Systems
- Manageability

Late 1990s to Mid 2000s
- Tree Learning Systems
- Developability

Mid 2000s to Mid 2010s
- Parameter Server
- Usability

Late 2000s to Early 2010s
- Deep Learning Systems
- ML System Abstractions

Mid 2010s
- Cloud ML/AI Services
- ML Platforms and Feature Stores

Late 2010s
- Onward
- TensorFlow
- PyTorch
- Cloud ML/AI Services
But what exactly is “deep” about DL?
Outline

❖ Introduction to Deep Learning
❖ Overview of DL Systems
❖ DL Training Systems
  ❖ Compilation and Execution
  ❖ Data Scaling
  ❖ Model Scaling
❖ DL Inference Systems
Unstructured Data Applications

- Many applications need to process unstructured data: text, images, audio, video, time series, etc.
- **Examples:** Machine translation, radiology, ASR, video surveillance, exercise activity analysis, etc.

Such data have low level formatting: strings, pixels, temporal shapes, etc.

Not intuitive what the *features* for prediction should be
Past Feature Engineering: Vision

- Decades of work on in machine vision on *hand-crafted* featurization based on crude heuristics

**Examples:**

- **Histogram of Oriented Gradient (HOG)**
  - **Fisher Vectors**
    - **Scale-invariant Feature Transform (SIFT)**

*Fig. 3. Histogram of oriented gradient extraction from face.*
Pains of Feature Engineering

- Ad hoc hand-crafted featurization had major cons:
  - Loss of information in “summarizing” data
  - Purely syntactic, lack “semantics” of objects
- Similar issues with hand-crafted text featurization, e.g., Bag-of-Words, parsing-based approaches, etc.

Q: Is there a way to mitigate above issues with hand-crafted feature extraction from such low-level data?
Learned Feature Engineering

❖ **Basic Idea:** Instead of hand crafting features, specify some *data type-specific invariants* and *learn feature extractors*

❖ **Examples:**

❖ Images have *spatial dependency*; not all pixel pairs are equal because nearby ones mean “something”

❖ Text tokens have local and global dependency in a sentence—not all words can go in all locations

❖ DL bakes in such data type-specific invariants to learn directly from (close-to-)raw inputs and produce outputs; aka “end-to-end” learning

❖ “Deep”: typically 3 or more layers to transform features
Different invariants baked into different DL sub-families

**Examples:** CNNs

Convolutional Neural Networks (CNNs) use *convolutions* to exploit invariants and learn hierarchy of relevant features from images.
Neural Architecture as Feature Extractors

- Different invariants baked into different deep learning models
- Examples: LSTMs

Long Short Term Memory Networks (LSTMs) use memory cells to exploit invariants in sequence data processing.
Neural Architecture as Feature Extractors

- Different invariants baked into different deep learning models
- **Examples**: Transformers

So-called “Attention” mechanism automatically weighs different parts/features of a sequence

Neural Architecture as Feature Extractors

- Also possible to mix and match learned featurizers in DL
- **Example**: CNN-LSTMs for time series

CNNs extract temporally relevant features locally, while LSTMs learn more global behavior; whole neural architecture (CNN-LSTM) is trained *end to end*.
Neural Architecture as Feature Extractors

- Also possible to mix and match learned featurizers in DL
- **Example:** CNN-LSTMs for video

CNNs extract visually relevant features at each time step, while LSTMs learn over those features across time; whole neural architecture (CNN-LSTM) is trained *end to end*
Flexibility is a superpower of DL methods:
- Almost any data type/structure as input and/or output
- Dependencies possible within input/output elements
Popularity of Deep Learning

- All major Web/tech firms use DL extensively; increasingly common in many enterprises and domain sciences too

Growing Use of Deep Learning at Google

# of directories containing model description files

Across many products/areas:
- Android
- Apps
- drug discovery
- Gmail
- Image understanding
- Maps
- Natural language understanding
- Photos
- Robotics research
- Speech
- Translation
- YouTube
- ... many others ...

![Graph showing the growth of unique project directories containing model description files over time.](chart.png)
Pros & Cons of DL (vs. Classical ML)

❖ **Pros:**

❖ **Accuracy:** Much higher than hand-crafted featurization on unstructured data

❖ **Flexibility:** Enables *unified* analytics of many modalities

❖ **Compact artifacts:** Succinct code, e.g., 5 lines in PyTorch vs. 500 of lines of raw Python/Java

❖ **Predictable resource use:** Useful during model serving

❖ **Cons:**

❖ **Neural architecture engineering:** Kinda resembles the pains of feature engineering

❖ **Large labeled data:** Needed in many cases to not overfit

❖ **High computational cost:** ‘Nuff said!
Discussion on Deep Learning
Outline

❖ Introduction to Deep Learning
❖ Overview of DL Systems
❖ DL Training Systems
  ❖ Compilation and Execution
  ❖ Data Scaling
  ❖ Model Scaling
❖ DL Inference Systems
**Q: What is a Deep Learning (DL) System?**

- A software system to specify, compile, and execute deep learning (DL) training and inference workloads on large datasets of any modality

**Specify**  
Neural computational graphs; auto-diff; SGD-based procedures

**Compile**  
Translate model computations (both training and inference) to hardware-specific kernels

**Execute**  
Place data and schedule model computations on hardware
Neural Computational Graphs (NCGs)

- Abstract representation of neural architecture and specification of training procedure

- A dataflow graph where the nodes represent operations in DL system’s API and edges represent tensors

- Tensor typically stored as NumPy object under the hood
DL System APIs

- PyTorch is the most common in academia and research; TensorFlow (TF) is more common in industry/production.

Most data scientists prefer the Python API.

Higher-level APIs are more succinct but more restrictive in terms of feature transformations.

Under the hood, TF compiles deep net specification to C++-based “kernels” to run on various processors.
Model Exchange Formats

❖ **Basic Goal:** *Portability* of model specification across systems
❖ These are domain-specific file formats that prescribe how to (de)serialize the neural architecture and training options
   ❖ Dataflow graph typically human-readable, e.g., JSON
   ❖ Weight matrices typically stored in binary format

*ONNX provides interoperability between frameworks*
Even Higher-level APIs

- Keras sits on top of APIs of TF, PyTorch; popular in practice
  - TF recently adopted Keras as a first-class API
- More restrictive specifications of neural architectures; trades off flexibility/customization for better usability
- Better for data scientists than low-level TF or PyTorch APIs, which may be better for DL researchers/engineers
- AutoKeras is an AutoML tool that sits on top of Keras to automate neural architecture selection + hyp.par. tuning
Outline

❖ Introduction to Deep Learning
❖ Overview of DL Systems
❖ DL Training Systems
  ❖ Compilation and Execution
  ❖ Data Scaling
  ❖ Model Scaling
❖ DL Inference Systems
Recall that DL training using SGD-based methods:

\[ W^{(t+1)} \leftarrow W^{(t)} - \eta \nabla \tilde{L}(W^{(t)}) \quad \nabla \tilde{L}(w^{(k)}) = \sum_{(y_i, x_i) \in B \subseteq D} \nabla l(y_i, f(w^{(k)}, x_i)) \]

Key difference with classical ML: weight updates are not one-shot but involve backpropagation.
Outline

- Introduction to Deep Learning
- Overview of DL Systems
- DL Training Systems
- Compilation and Execution
  - Data Scaling
  - Model Scaling
- DL Inference Systems
Backpropagation Algorithm

- An application of the chain rule from differential calculus
- Layers of neural net = series of function compositions

\[
\frac{d}{dx} [f(g(x))] = f'(g(x))g'(x)
\]

Forward pass

<table>
<thead>
<tr>
<th>Input x</th>
<th>1</th>
<th>1</th>
<th>output (\hat{y})</th>
</tr>
</thead>
</table>

Backprop/Backward pass

\[
\frac{\partial}{\partial w_{ij}} J(W) = a_j^{(0)} \delta_{i}^{(l+1)}
\]

(error term of the output layer)

\[
\delta^{(3)} = a^{(3)} - y
\]

(error term of the hidden layer)

\[
\delta^{(2)} = (W^{(2)})^T \delta^{(3)} \cdot \frac{\partial g(z^{(2)})}{\partial z^{(2)}}
\]

Input x → output \(\hat{y}\) → target y

https://sebastianraschka.com/faq/docs/visual-backpropagation.html
A key benefit of DL tools: gradients are computed *symbolically* and automatically

- No numerical methods/approximations needed
- Calculus is abstracted away!

Feasible because API to express arch. and loss function has pre-defined dataflow ops with known properties

- Code specifies derivatives of each op

Pioneered in Theano; now adopted in all DL tools
Differentiable Programming

- DL tools have heralded this new programming paradigm
  - Easily compose 1000s of functions using a hierarchy of more abstract APIs
  - “Model is the new code”!
- E.g., tf.math has ~130 functions, tf.nn has ~80 functions, Keras layers ~100 functions!

https://www.tensorflow.org/api_docs/python/tf/all_symbols
https://keras.io/api/
DL systems must translate DL code with even millions of tensor ops efficiently down to hardware kernels.

- Analogous to RDBMS's SQL translation stack.
- IR-based approach enables unified support for a variety of hardware backends, e.g., GPUs, CPUs, FPGAs, TPUs, other ASICs (e.g., on mobiles or IoT).
Hardware Kernels in DL Systems

- DL training is almost always performed on GPUs
  - NVIDIA’s CuDNN on top of base CUDA

- Optimized use of GPU memory/caches and PUs for DL ops, e.g., convolution
  - Much faster than best CPUs

- All popular DL systems support CuDNN backend
  - Some have new CUDA kernels for better control or memory handling
Translating a Neural Comp. Graph

- 2 major variants: static and dynamic
  - Static unrolls the NCG, compiles and optimizes the ops directly to hardware kernels in one go
  - Dynamic takes an interpreted approach; NGG structure itself can change on the fly!
- Static is more amenable to program optimizations and can be more scalable
- Dynamic is more flexible and popular in DL research
- Different DL sub-families have different requirements:
  - CNNs, transformers, RNNs on time series usually static
  - Fancier RNNs on text, graph NNs tend to be dynamic
DL Heterogeneity

❖ Dozens of DL sub-families are used in practice or at least studied!
❖ DL researchers keep designing new kinds of differentiable programs that stretch the capabilities of modern DL systems
❖ Facebook and Google are apparently working on a new PL for DL!

https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464
Compiler-level Optimizations

- Popular DL systems support compiler optimizations to reduce computations, reduce memory stalls, and/or raise hardware parallelism
  - Operator fusion of tensor arithmetic
  - Sharding of tensors across cores / PUs
  - Operator placement on multi-device environments
Review Questions

1. Why is DL popular on image data?
2. Discuss 2 key advantages of DL over classical ML.
3. Discuss 2 key disadvantages of DL over classical ML.
4. What is AutoDiff?
5. What is the benefit of dynamic compilation in DL?
6. Why is DL not (yet) popular on tabular data?
Outline

❖ Introduction to Deep Learning
❖ Overview of DL Systems
❖ DL Training Systems
  ❖ Compilation and Execution
  ❖ Data Scaling
  ❖ Model Scaling
❖ DL Inference Systems
Recap: 3 Parts of DL Training Iterate

- Forward pass to compute loss on mini-batch -> Backprop to compute gradients -> Updates of parameters

\[
W^{(t+1)} \leftarrow W^{(t)} - \eta \nabla \tilde{L}(W^{(t)})
\]
Recap: Distributed SGD via PS

- Distr. SGD needs to sync gradients/params across workers
- PS allows for async updates with gradients/params
Distributed DL Training

❖ **Goal**: Parallelize DL training with SGD on sharded data
❖ Many DL systems support PS-style sync/async distribution

<table>
<thead>
<tr>
<th>Training API</th>
<th>MirroredStrategy</th>
<th>TPUStrategy</th>
<th>MultiWorkerMirroredStrategy</th>
<th>CentralStorageStrategy</th>
<th>ParameterServerStrategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keras API</td>
<td>Supported</td>
<td>Supported</td>
<td>Experimental support</td>
<td>Experimental support</td>
<td>Supported planned post 2.3</td>
</tr>
<tr>
<td>Custom training loop</td>
<td>Supported</td>
<td>Supported</td>
<td>Experimental support</td>
<td>Experimental support</td>
<td>Supported planned post 2.3</td>
</tr>
<tr>
<td>Estimator API</td>
<td>Limited Support</td>
<td>Not supported</td>
<td>Limited Support</td>
<td>Limited Support</td>
<td>Limited Support</td>
</tr>
</tbody>
</table>

❖ Unfortunately, PS is a poor fit for most of DL:
❖ Non-trivial sizes of DL gradients, unlike classical ML
❖ Heavily communication-bound; very sub-linear speedup
❖ NB: PS was designed before the DL era!

[https://www.tensorflow.org/guide/distributed_training](https://www.tensorflow.org/guide/distributed_training)
**Introducing Horovod**

❖ **Goal**: Mitigate the communication bottleneck of distributed DL training, esp. for exchanging/syncing gradients

❖ **Basic Idea**:
Introducing Horovod

❖ **Goal**: Mitigate communication bottleneck for distributed DL training, especially to synchronize gradients

❖ **Intuition**: Do not sync up all gradients of DL NCG at once

❖ **Basic Idea**: “Ring AllReduce” from HPC world
  ❖ **Decentralized**, i.e., no designated manager/server
  ❖ **Ring topology** for workers to talk to each other
  ❖ **Sharded updates** exchanged among workers instead of sending all gradients of an iterate in one go
  ❖ **Multiple rounds** of talking for all to get in sync

❖ Logically equivalent to sequential SGD! No PS-style heuristics with stale updates, etc.
Assume a DL NCG’s params/gradients are logically sharded on a worker into roughly equi-sized bins.

In each round, a worker sends a bin and receives a different bin used to update resp. local copy; repeat until all synced.
Ring AllReduce Parallelization

❖ Given N workers, each talks to 2 peers 2*(N-1) times to sync up one iterate
❖ Do this for every iterate (mini-batch) of SGD
Horovod vs. PS

- Horovod is *synchronous* unlike PS philosophy but still better
- 2 key benefits of Horovod’s Ring AllReduce vs PS:
  - Better network utilization due to decentralization; it is bandwidth-optimal
  - Lower communication costs

N workers, M gradients/params size, K mini-batches per worker

**Total per-epoch comm. cost:**
- PS: $2MNK$
- Horovod: $2M(N-1)K$
Empirical Comparisons

- Horovod has higher speedups than PS (up to a limit)
Distributed PyTorch

- PyTorch’s DDP (Distr. Data Parallel) DL training added a few more systems tricks beyond Ring AllReduce:
  - Gradient Bucketing (exact)
  - Communication-Computation Pipelining (exact)
  - Send updates after every few mini-batches (heuristic)
- The first two preserve accuracy but third may hurt accuracy
Observation: An NCG has multiple layers of gradients

Basic Idea: “Bucket” multiple gradients onto one bin to reduce number of invocations of AllReduce

(Technically already possible in Horovod)
Distr. PyTorch: Overlap Comm.-Comp.

- **Observation**: Waiting for whole backprop to finish per iterate before syncing keeps network idle; likewise while network is working, worker’s PU is idle.

- **Basic Idea**: Stage layer’s gradients (adjust bin size) to *interleave* backprop computation with communication.
  - Standard systems trick of pipeline parallelism to hide (network) I/O latency.
Distr. PyTorch: Scalability

- Strangely, they show only scaleup plot, not speedup plots
- Scaleup depends on model and hardware

(c) BERT on NCCL  
(d) BERT on Gloo
## Tradeoffs of Horovod / Distr. PyTorch

<table>
<thead>
<tr>
<th>Pros:</th>
<th>Cons:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Usability:</strong></td>
<td>PyTorch is not well integrated with ETL stacks</td>
</tr>
<tr>
<td>Unlike PS, both support</td>
<td></td>
</tr>
<tr>
<td>data-scaling for dense</td>
<td></td>
</tr>
<tr>
<td>DL; reproducible</td>
<td></td>
</tr>
<tr>
<td><strong>Manageability:</strong></td>
<td>Distr. PyTorch hard to operate/govern; fault tol.</td>
</tr>
<tr>
<td>Horovod integrated with</td>
<td>hard in both</td>
</tr>
<tr>
<td>Spark and DL tools</td>
<td></td>
</tr>
<tr>
<td><strong>Efficiency:</strong></td>
<td>Still high comm. cost; somewhat sub-linear scaling</td>
</tr>
<tr>
<td>Faster than PS and other distr.</td>
<td></td>
</tr>
<tr>
<td>SGD tools</td>
<td></td>
</tr>
<tr>
<td><strong>Scalability:</strong></td>
<td>Not suitable for very large clusters; speedup</td>
</tr>
<tr>
<td>Reasonably high; work for</td>
<td>flattens</td>
</tr>
<tr>
<td>dozens of nodes</td>
<td></td>
</tr>
<tr>
<td><strong>Developability:</strong></td>
<td>May need DL systems expertise to use</td>
</tr>
<tr>
<td>No need to worry about</td>
<td></td>
</tr>
<tr>
<td>consistency tradeoffs</td>
<td></td>
</tr>
</tbody>
</table>
Your Reviews on TF

❖ (Walked through in class)
Outline

❖ Introduction to Deep Learning
❖ Overview of DL Systems
❖ DL Training Systems
  ❖ Compilation and Execution
  ❖ Data Scaling
  ❖ Model Scaling
❖ DL Inference Systems
Need for Model Scalability

- In some DL sub-families, especially NLP, models becoming larger than GPU memory:
  - V100: 16-32GB; A100: up to 80GB
  - GPT-2 has 1.5B parameters (~6GB)
  - Need space for data batch and intermediates too; GPU memory footprint can blow up by ~20x
  - GPT-3 is 175B!
Transformer architectures are the most common DL model type that faces model scalability issues.

http://jalammar.github.io/illustrated-transformer/
Another DL sub-family facing this issue: graph neural networks (GNNs), e.g., in social graph analytics

GNN+CNN combinations also arise for multimedia data
Model Scalability

- Typical approach today: **model parallelism**
  - Shard model across multiple GPUs
  - Exchange features / backprop updates periodically

- Layer-aligned sharding is common in model scaling; lower inter-GPU comm. costs
- But intra-layer sharding is making a comeback (FSDP)

Model Scaling: Pipelining

❖ A common optimization with layer-aligned sharding; GPipe from Google is an exemplar
❖ Stages out forward passes (and backward passes) across subsets of mini-batches called *micro-batches*

[Diagram showing the concept of pipelining in model scaling]

State-of-the-art DL model scaling tool from Microsoft

**DRAM offloading:** Spill shards of large model to DRAM, both model state and gradients
- Can scale to 10B parameters on single GPU!

**“3D” parallelism:** Combines data-parallelism, model-parallelism, and pipelining
- Mitigates the “bubble” issue of pure pipelining

Several other systems-level features supported:
- Easier checkpointing, efficient loading, mixed precision, memory layout/bandwidth optimizations, etc.

[https://www.deepspeed.ai/](https://www.deepspeed.ai/)
Model Scaling Approach: DeepSpeed

- Example of 8 micro-batches hybridized as 2-way data-parallel execution, with 2-GPU pipelining within each.
- AllReduce (AR) used to sync gradients at the end before model update step, akin to Horovod.
- Yields exact mini-batch gradient update (no inconsistency).

These are all different micro-batches (8 in all) of one mini-batch.
Model Scaling Approach: DeepSpeed

- Here is my (animated) revamp of DeepSpeed’s botched illustration. :)  
- Split data mini-batch into 8 micro-batches D1 to D8  
- Split 4-GPU cluster into 2 sub-clusters: {G1, G2}, {G3, G4}  
- Each sub-cluster has full model copy; AllRed. for 2-way data-par. across  
- Within each sub-cluster, split model into 2 shards: M1->M2; run them with 2-way pipelining for 4 micro-batches

Legend: Forw. & Backw. pass on shard M1 w/ micro-batch D1  
AllRed. for grads of shard M2  
Weight updates on shard M1

Time

G1: \(F_{M1,D1}\)  
G2: \(F_{M2,D1}\)  
G3: \(F_{M1,D5}\)  
G4: \(F_{M2,D5}\)
Examples of 3D parallelism, hybridizing model sharding, pipelining, and data-parallelism

https://www.deepspeed.ai/
Model Scaling Approach: FSDP

❖ More advanced hybridization of 3D parallelism as seen in DeepSpeed; now the recommended default in PyTorch
❖ Each layer is sharded across GPUs; data mini-batch too
❖ Most scalable form of model scaling today

❖ Stages out All-Reduce (as in Horovod/DDP/DeepSpeed) to Reduce-Scatter and All-Gather

https://engineering.fb.com/2021/07/15/open-source/fsdp/
Model Scaling Approach: FSDP

❖ Such per-layer sharding can raise communication between GPUs; tradeoff for ease of scalability
❖ Needs fast GPU-GPU interconnect/network to work well

Neither GPipe nor DeepSpeed/FSDP dominate; tradeoffs based on NCG specs, batch size, GPU specs, and # GPUs

https://engineering.fb.com/2021/07/15/open-source/fsdp/
Peer Instruction Activity

(Switch slides)
Discussion on DL training systems
Review Questions

1. Why is PS a poor fit for DL training?
2. Why does Horovod perform better than PS for DL training?
3. Explain 1 advantage and 1 disadvantage of PyTorch DDP over Horovod.
4. Why does pure pipeline parallelism for model scaling underutilize GPUs?
5. Briefly explain 2 systems techniques in DeepSpeed to make model scaling more efficient.
6. Briefly explain 1 key systems technique in FSDP compared to DeepSpeed that raises model scalability.
Outline

❖ Introduction to Deep Learning
❖ Overview of DL Systems
❖ DL Training Systems
  ❖ Compilation and Execution
  ❖ Data Scaling
  ❖ Model Scaling
❖ DL Inference Systems
Why Study DL Inference?

- DL inference is a strict subset of training: on an example, just do forward pass to get prediction.
  
  **Q:** Why bother optimizing DL inference any further?

- Qualitative differences of inference vs training:
  - Happens *far more often* than training; economies of scale for reducing inference cost.
  - Many apps need *near real-time* inference, e.g., Web.
  - NCG/weights are *fixed* for inference stage, enabling deeper systems optimizations.
Background: Roofline Analysis

- A tool from comp. arch. to understand if/how some systems optimizations can help
- Fundamental issue: keep PU busy vs memory stalls
Optimizing NCG Inference

- DL models tend to have high arithmetic intensity; but there is a spectrum on memory-bound vs compute-bound

- Different layers within same DL models also fall on diff. points in the spectrum

- Hand-optimizing is tedious/hard; need automated compiler to do it

Figure 10: Roofline [47] of an FPGA-based DL accelerator running ResNet inference. With latency hiding enabled by TVM, performance of the benchmarks is brought closer to the roofline, demonstrating higher compute and memory bandwidth efficiency.
Peer Instruction Activity

(Switch slides)
The TVM Compiler

❖ **Goal**: A unified compiler to support multiple DL frameworks’ inference on multiple hardware backends extensibly

❖ **Challenges**: hardware heterogeneity; so many DL ops

---

**Memory Subsystem Architecture**

- **CPU**:
  - L3
  - L2
  - L1D
  - L1I
  - implicitly managed

- **GPU**:
  - L2
  - SM
  - L1/TX
  - RF

- **‘TPU’**:
  - Activation Buffer
  - explicit management
  - Wgt. FIFO
  - Accum. Register File

**Compute Primitive**

- **Scalar**
- **Vector**
- **Tensor**
The TVM Compiler

❖ Approach: A unified intermediate representation (IR) + series of optimizations + ML-based instruction scheduler

![Diagram showing the TVM Compiler process]
Compiler Optimizations in TVM

❖ Standard compilers tricks (matters for any PL):
  ❖ Operator fusion
  ❖ Data layout transformations
  ❖ Nested parallelism for memory access
❖ New techniques designed for DL NCGs and hardware:
  ❖ Tensorization of almost all ops
  ❖ Pipelining to hide memory stalls
  ❖ ML-based schedule generation
Operator Fusion

- **Technique**: Combine two or more tensor ops into a single “larger” op
- **Benefit**: Avoids memory stall for intermediate results; so, helps reduce runtimes, especially on GPUs
- TVM categories all tensor ops based on fusability and has rules to inject this optimization
Data Layout Transformations

- **Technique**: Sharding intermediate tensors in axis-oriented or tile-oriented
- **Benefit**: Maximizes data parallelism for ops on PUs
- Too complex to handcode with rules
- TVM decouples tensor op spec. vs exact instructions by using a code-generation approach
  - Allows for backend-specific unrolling and sizing
Data Layout Transformations

\[
A = t.placeholder((1024, 1024))
B = t.placeholder((1024, 1024))
k = t.reduce_axis((0, 1024))
C = t.compute((1024, 1024), \text{lambda } y, x:\t.sum(A[k, y] * B[k, x], \text{axis}=k))
s = t.create_schedule(C.op)
\]

\[
\text{for } y \text{ in range(1024):}
\quad \text{for } x \text{ in range(1024):}
\quad \quad C[y][x] = 0
\quad \text{for } k \text{ in range(1024):}
\quad \quad C[y][x] = A[k][y] * B[k][x]
\]

\+ Loop Tiling
\[
yo, xo, ko, yi, xi, ki = s[C].tile(y, x, k, 8, 8, 8)
\]

\[
\text{for } yo \text{ in range(128):}
\quad \text{for } xo \text{ in range(128):}
\quad \quad C[yo*8:yo*8+8][xo*8:xo*8+8] = 0
\quad \text{for } ko \text{ in range(128):}
\quad \quad \text{for } yi \text{ in range(8):}
\quad \quad \text{for } xi \text{ in range(8):}
\quad \quad \quad C[yo*8+yi][xo*8+xi] = A[ko*8+ki][yo*8+yi] * B[ko*8+ki][xo*8+xi]
\]

\+ Cache Data on Accelerator Special Buffer
\[
CL = s.cache_write(C, vsla.acc_buffer)
AL = s.cache_read(A, vsla.inp_buffer)
\# additional schedule steps omitted ...
\]

\+ Map to Accelerator Tensor Instructions
\[
s[CL].tensorize(yi, vsla.gemm8x8)
\]

\[
inp\_buffer AL[8][8], BL[8][8]
acc\_buffer CL[8][8]
\]

\[
\text{for } yo \text{ in range(128):}
\quad \text{for } xo \text{ in range(128):}
\quad \quad vsla.fill_zero(CL)
\quad \text{for } ko \text{ in range(128):}
\quad \quad \quad vsla.dma_copy2d(AL, A[ko*8:ko*8+8][yo*8:yo*8+8])
\quad \quad \quad vsla.dma_copy2d(BL, B[ko*8:ko*8+8][xo*8:xo*8+8])
\quad \quad \quad vsla.fused_gemm8x8_add(CL, AL, BL)
\quad \quad \quad vsla.dma_copy2d(C[yo*8:yo*8+8,xo*8:xo*8+8], CL)
\]
Nested Parallelism

- GPUs have complex hierarchy of on-device memory/caches
- **Technique**: Groups of threads fetch shared data regions (e.g., accumulator) to higher cache and reuse it
- **Benefit**: Reduces delay caused by memory stalls
Tensorization of NCG Ops

- **Technique:** Allow declarations of NCG ops in tensor form
- **Benefit:** Extensibility to convert ops to different forms of parallel micro-kernels on hardware, e.g., lower precision

```python
w, x = t.placeholder((8, 8)), t.placeholder((8, 8))
k = t.reduce_axis((0, 8))
y = t.compute((8, 8), lambda i, j:
    t.sum(w[i, k] * x[j, k], axis=k))

def gemm_intrin_lower(inputs, outputs):
    ww_ptr = inputs[0].access_ptr("r")
    xx_ptr = inputs[1].access_ptr("r")
    zz_ptr = outputs[0].access_ptr("w")
    compute = t.hardware_intrin("gemm8x8", ww_ptr, xx_ptr, zz_ptr)
    reset = t.hardware_intrin("fill_zero", zz_ptr)
    update = t.hardware_intrin("fuse_gemm8x8_add", ww_ptr, xx_ptr, zz_ptr)
    return compute, reset, update

gemm8x8 = t.decl_tensor_intrin(y.op, gemm_intrin_lower)
```
Pipelining to Hide Memory Latency

- **Technique:** Interleave computation instruction and memory access instruction
- **Benefit:** Hides latency of memory stall; keeps PUs busy
- Achieved with multithreading on CPUs and GPUs; for accelerators, TVM has primitives to avoid out-of-order
ML-based Instruction Schedule

❖ So many configurable optimization choices (data layouts, lower level kernels, pipelining choices, etc.) make it too complex to create optimal final hardware instructions

❖ Technique: Use ML in compiler!
  ❖ “Explorer” module constructs candidate configs; ML “cost model” predicts performance

❖ Benefit:

<table>
<thead>
<tr>
<th>Method Category</th>
<th>Data Cost</th>
<th>Model Bias</th>
<th>Need Hardware Info</th>
<th>Learn from History</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blackbox auto-tuning</td>
<td>high</td>
<td>none</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Predefined cost model</td>
<td>none</td>
<td>high</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td><strong>ML based cost model</strong></td>
<td>low</td>
<td>low</td>
<td><strong>no</strong></td>
<td><strong>yes</strong></td>
</tr>
</tbody>
</table>
ML-based Instruction Schedule

- **TensorOp Specification**
  - Database
  - Schedule Space Template
  - Schedule Explorer
  - ML Cost Model
  - Tracker

- **Device Cluster**
  - Raspberry Pi
  - Mali GPU
  - Nvidia GPU
  - FPGA Board
  - ...

- **Training Data**
  - Query: Loop AST
  - Feature Extraction
    - e.g. touched memory size
    - \( \begin{array}{cccccc} 
    \text{xi} & \text{yi} & \text{k} & \text{xo} & \text{yo} \\
    \text{C} & 2 & 4 & 4 & 16 & 64 \\
    \text{A} & 1 & 2 & 16 & 16 & 64 \\
    \text{B} & 2 & 2 & 16 & 64 & 64 \\
    \end{array} \)
  - XGBoost
    - cost prediction

- **Graph**
  - Relative Speedup
    - TVM: ML-based Model
    - TVM: Blackbox Genetic Algorithm
    - TVM: Random Search
    - Baseline: cuDNN
    - Number of Trials
## Tradeoffs of TVM for DL Inference

<table>
<thead>
<tr>
<th></th>
<th>Pros:</th>
<th>Cons:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Usability:</strong></td>
<td>Highly general; supports many DL tools and target processors</td>
<td>N/A (compilers are mostly hidden from DL users)</td>
</tr>
<tr>
<td><strong>Manageability:</strong></td>
<td>Apache project; large community to help</td>
<td>Extra dependency to manage for DL users</td>
</tr>
<tr>
<td><strong>Efficiency:</strong></td>
<td>Faster than CuDNN on GPUs; fast on other h/w</td>
<td>Likely slower than an ASIC-specific compiler stack</td>
</tr>
<tr>
<td><strong>Scalability:</strong></td>
<td>N/A (for inference only)</td>
<td>No native support for model scaling; outsources to DL tool</td>
</tr>
<tr>
<td><strong>Developability:</strong></td>
<td>Easily extensible; many optimizations port well</td>
<td>DL tool engineers must use TVM primitives for best perf.</td>
</tr>
</tbody>
</table>
Your Reviews on TVM

❖ (Walked through in class)
Bonus: FlashAttention

(Not included in the syllabus)
Outline

❖ Introduction to Deep Learning
❖ Overview of DL Systems
❖ DL Training Systems
  ❖ Compilation and Execution
  ❖ Data Scaling
  ❖ Model Scaling
❖ DL Inference Systems