From Algorithm to Deployment

- ML Algorithms
  - Maths, Convergence, Proof, Models, Accuracy
- Programming
  - API
- Execution
- Hardware Design
  - Acceleration for specialized operators
  - Memory capacity, bandwidth
  - Memory hierarchy
  - Communication latency and bandwidth
  - Communication topology
API Abstraction

- **Vallina C/Python/…**
  - for-loops, array, scalar math ops
  - Tedious, Error-prone
- **Vectorized representation**
  - numpy, ndarray, dot. Linear algebra.
  - Multiple implems + Hide impl details
- **Operators**
  - MatMul, Softmax, Convolution
- **Layers**
  - Dense, Conv2D, Transformer
- **Models**
  - Layers
  - Control Flow
Machine Learning Frameworks / Compilers

- **User-friendly APIs**
  - Operators, Layers
  - Optimizers, Loss functions
  - Auto gradient, parameter update
  - Data loading
  - Multi-device, Multi-machine

- **Intermediate Representation**
  - Graph
  - High-level instruction sets (MLIR, LLVM)
  - Opportunities for auto optimization
    - (Imagine optimizing hand written C/Python)

- **Support various accelerator hardware**
  - Computation, Memory, Communication
TensorFlow: Graph

- **Node: Op**
  - Add, MatMul, Conv2D
  - Abstract device-, execution backend-, and language independent API
  - Implemented by Op Kernels written in C++, specialized on $<\text{Type}, \text{Device}>$

- **Edge: Data dependency**
  - Tensors (ref-counted, n-dimensional array buffers in device memory)
  - Control dependencies: A->B means A must finish before B can run
  - Resource handles to state (e.g. variables, input data pipelines)
TensorFlow: Graph

- Node: Op
- Edge: Data dependency

Graph Analysis & Transformation

- Auto gradient (chain rule)
- Dependency Analysis
- Split subgraph
Grappler: TensorFlow Graph Optimizations

Graph: High-level IR

Not the only IR

https://web.stanford.edu/class/cs245/slides/TFGraphOptimizationsStanford.pdf
Why transformations at the graph level?

- **Pros:**
  - Many optimizations can be easier to discover and express as high-level graph transformations
    - Example: `Matmul(Transpose(x), y) => Matmul(x, y, transpose_x=True)`
  - Graph is backend independent (TF runtime, XLA, TensorRT, TensorFlow.js, ...)
  - Interoperable with TensorFlow supported languages (protocol buffer format)
  - Optimizations can be applied at **runtime** or **offline** using our **standalone tool**
  - Lots of **existing models** (TF Hub, Google production models) available for learning
  - Pragmatic: Helps the most existing TensorFlow users get better “out-of-the-box” performance

- **Cons:**
  - Rewrites can be tricky to implement correctly, because of loosely defined graph semantics
    - In-place ops, side-effects, control flow, control dependencies
  - Protocol buffer dependence increases binary size
  - Currently requires extra graph format conversions in TF runtime
Graph Simplifications

```
S = tf.shape(A)  # S = [2, 2]
B = tf.ones(S)
```

```
S = tf.constant([2, 2])
B = tf.constant([[1, 1], [1, 1]])
```

```
S = tf.constant([2, 2])
B = tf.ones(S)
```
Constant folding optimizer: SimplifyGraph()

- Removes trivial ops, e.g., identity Reshape, Transpose of 1-d tensors, Slice(x) = x, etc.
- Rewrites that enable further constant folding, e.g.
  - Constant propagation through Enter
  - Switch(pred=x, value=x) => propagate False through port0, True through port1
  - Partial constant propagation through IdentityN
- Arithmetic rewrites that rely on known shapes or inputs, e.g.
  - Constant push-down:
    - Add(c1, Add(x, c2)) => Add(x, c1 + c2)
    - ConvND(c1 * x, c2) => ConvND(x, c1 * c2)
  - Partial constfold:
    - AddN(c1, x, c2, y) => AddN(c1 + c2, x, y),
    - Concat([x, c1, c2, y]) = Concat([x, Concat([c1, c2]), y])
  - Operations with neutral & absorbing elements:
    - x * Ones(s) => Identity(x), if shape(x) == output_shape
    - x * Ones(s) => BroadcastTo(x, Shape(s)), if shape(s) == output_shape
    - Same for x + Zeros(s), x / Ones(s), x * Zeros(s) etc.
    - Zeros(s) - y => Neg(y), if shape(y) == output_shape
    - Ones(s) / y => Recip(y) if shape(y) == output_shape
Arithmetic optimizer:

- Arithmetic simplifications
  - Flattening: \( a+b+c+d \Rightarrow AddN(a, b, c, d) \)
  - Hoisting: \( AddN(x * a, b * x, x * c) \Rightarrow x * AddN(a+b+c) \)
  - Simplification to reduce number of nodes:
    - Numeric: \( x+x+x \Rightarrow 3*x \)
    - Logic: \( !(x > y) \Rightarrow x <= y \)

- Broadcast minimization
  - Example: \((\text{matrix1} + \text{scalar1}) + (\text{matrix2} + \text{scalar2}) \Rightarrow (\text{matrix1} + \text{matrix2}) + (\text{scalar1} + \text{scalar2})\)

- Better use of intrinsics
  - \( \text{Matmul}(\text{Transpose}(x), y) \Rightarrow \text{Matmul}(x, y, \text{transpose}_x=True) \)

- Remove redundant ops or op pairs
  - \( \text{Transpose}(\text{Transpose}(x, \text{perm}), \text{inverse}_\text{perm}) \)
  - \( \text{BitCast}(\text{BitCast}(x, \text{dtype}1), \text{dtype}2) \Rightarrow \text{BitCast}(x, \text{dtype}2) \)
  - Pairs of elementwise involutions \( f(f(x)) \Rightarrow x \) (Neg, Conj, Reciprocal, LogicalNot)
  - Repeated idempotent ops \( f(f(x)) \Rightarrow f(x) \) ( DeepCopy, Identity, CheckNumerics...)

- Hoist chains of unary ops at Concat/Split/SplitV
  - \( \text{Concat}([\text{Exp}(\text{Cos}(x)), \text{Exp}(\text{Cos}(y)), \text{Exp}(\text{Cos}(z))]) \Rightarrow \text{Exp}(\text{Cos}(\text{Concat}([x, y, z]))) \)
  - \( [\text{Exp}(\text{Cos}(y)) \text{ for } y \text{ in } \text{Split}(x)] \Rightarrow \text{Split}(\text{Exp}(\text{Cos}(x)), \text{num_splits}) \)
Layout optimizer

Node 4
- NHWC to NCHW
- Conv in NCHW
- NCHW to NHWC

Node 5
- BiasAdd in NHWC

Node 6
- Relu

Node 7
- MaxPool in NHWC

Node 8
- NHWC to NCHW
- Conv in NCHW
- NCHW to NHWC

Node 9
- BiasAdd in NHWC

Node 48
- NCHW to NHWC
- Conv in NCHW
- NCHW to NHWC

Node 48
- NCHW to NHWC
- Conv in NCHW
- NCHW to NHWC

https://web.stanford.edu/class/cs245/slides/TFGraphOptimizationsStanford.pdf
Remapper optimizer: Op fusion

- Replaces commonly occurring subgraphs with optimized fused "monolithic" kernels
  - Examples of patterns fused:
    - Conv2D + BiasAdd + <Activation>
    - Conv2D + FusedBatchNorm + <Activation>
    - Conv2D + Squeeze + BiasAdd
    - MatMul + BiasAdd + <Activation>

- Fusing ops together provides several performance advantages:
  - Completely eliminates Op scheduling overhead (big win for cheap ops)
  - Increases opportunities for ILP, vectorization etc.
  - Improves temporal and spatial locality of data access
    - E.g. MatMul is computed block-wise and bias and activation function can be applied while data is still "hot" in cache.

- A separate mechanism allows the TensorFlow compiler to cluster subgraphs and generate fused kernel code on-the-fly
TensorFlow 2.0: Eager Execution

Graph Execution

- Build graph
- `tf.Session`: owns all states
- `sess.run()`: run the graph

Eager Execution:

- Numpy-like
- PyTorch gain popularity because of eager execution
- `print(x)`
- Support for dynamic models using easy-to-use Python control flow
TensorFlow 2.0: Eager Execution

● Upside:
  ○ Fast debugging with immediate run-time errors and integration with Python tools
  ○ Support for dynamic models using easy-to-use Python control flow

● Downside:
  ○ Slow
    ■ Interpreting Python code
    ■ Fixed, unoptimized code path
    ■ Issue kernels one by one
    ■ No op fusion
    ■ No graph optimizations

● User friendly + Performance
  ○ `tf.function()` / `torch.jit.script()`
    ■ Trace Python code once for given input specs (function signature, e.g., dtype, shape)
    ■ Eager code -> Graph
TensorFlow: Data Parallel Training

- One 1000-element mini-batch == Ten 100-element mini-batches
- Easiest way to use multiple GPUs
  - Replicate the model across GPUs
  - Shard data across GPUs
  - Compute gradient on each GPU
  - Aggregate gradients
    - Sync: wait for slowest
    - Async: different semantics
      - Gradient of old parameters
      - Convergence?
Data Parallelism: Parameter Server

- **API:**
  - `ps.push(key, gradient)`
  - `ps.pull(key)`

- **Roles:**
  - **Server:** Key-value store; Merge gradient
  - **Worker:** Calculate gradient

- **Consistency Model**
  - **Sequential (Sync)**
  - **Eventual (Async)**
  - **Bounded Delay (tuneable)**

- **Server bottleneck:**
  - **High bandwidth demand**
  - **Synchronized burst**
  - **How to fix it? (Multi-server!)**

https://www.cs.cmu.edu/~muli/file(parameter_server_osdi14.pdf)
Data Parallelism: Parameter Server

- Multiple servers
  - Shard across Key space.
- How to deal with skewed key space (e.g., string as keys)?
- How to deal with server load imbalance?
- This reminds you of a paper...
Data Parallelism: Parameter Server

- **Multiple servers**
  - Shard across Key space.
  - Each server is responsible for a range of keys.
  - Chord?!
    - Load balancing of keys: hashing
    - Load balancing of servers: virtual nodes

**Uber Horovod: Challenges with PS**

- **Worker:PS ratio**
  - Single PS: bottleneck
  - One PS per worker: all-to-all, may saturate network switch

- **Integration with existing TensorFlow program**
  - Service discovery for PS and worker
  - Modify code to shard parameters explicitly

Data Parallelism: Collective Communication


Page 4-7,11-12,18-47

● Advantage:
  ○ The number of devices does not affect the latency
  ○ Bandwidth optimal
  ○ Interconnect topology aware
  ○ Minimal modification to code (allreduce)
Machine Learning Parallelism

● Data Parallelism
  ○ Small model; Large dataset;
  ○ Replicate model; Shard dataset; Sync update
  ○ Collective communication

● Model Parallelism
  ○ Large model: a model might require multiple devices
  ○ Pipeline parallelism
    ■ Partition a model into several stages
    ■ Less communication; More idle time
  ○ Operator parallelism
    ■ Partition an operator along some dimensions
    ■ More communication; Less idle time
  ○ Point-to-point communication
Pipeline Parallelism

- No pipeline: bubbles
- GPipe
  - Split a mini-batch as many “micro-batch”
  - Memory: linear to micro-batches
- PipeDream
  - Async update (1F1B)
  - Lose accuracy
Pipeline Parallelism

- No pipeline: bubbles
- GPipe
  - Split a mini-batch as many “micro-batch”
  - Memory: linear to micro-batches
- PipeDream
  - Async update (1F1B)
  - Lose accuracy
- PipeDream-Flush
  - Sync; Alternate Forward & Backward
  - Save memory: linear to pipeline stages
- Megatron-2 Virtual Pipeline
  - Place multiple stages on the same device
  - More communication; Less bubble
Operator Parallelism

- Alpa
  - [https://www.usenix.org/sites/default/files/conference/protected-files/osdi22_slides_zheng-lian min.pdf](https://www.usenix.org/sites/default/files/conference/protected-files/osdi22_slides_zheng-lian min.pdf)
  - Data + Pipeline + Operator parallelism
  - Two tier network topology
Model Serving (Inference)

- Latency constraint for real-time tasks
  - e.g., end-to-end latency < 10ms
- Multi-tenancy
  - e.g., multiple models on one GPU cluster
- Request rate fluctuation
  - Piecewise stationary + burst
- Hardware utilization
  - batching under latency constraint
- GPU cluster management
  - load balancing
  - horizontal scaling
Inference Characteristics on GPUs

- Very predictable execution latency
- Concurrent execution increases throughout but significantly sacrifices predictability
- Execution latency is linear to batch size
  - $\text{latency}(bs) := k \times bs + c$
  - $\text{throughput}(bs) := \frac{bs}{\text{latency}(bs)} \propto -1/bs$

Model Serving Systems

● Roles:
  ○ Client
  ○ Frontend servers
    ■ Accept client requests
    ■ Preprocessing (e.g., image decoding)
    ■ Forward request to backend
    ■ Postprocessing (e.g., index to label)
    ■ Send response back to client
  ○ Backend servers
    ■ Run models with GPU
  ○ Scheduler
    ■ Backend allocation
    ■ Model mapping
    ■ Execution plan
Model Serving System: Scheduling

● Schedule:
  ○ Which GPU to run this batch?
  ○ Which requests are included in this batch?
  ○ When to start running this batch?

● Distributed scheduling (Nexus [SOSP’19])
  ○ Request lifetime: Client -> Frontend -> Backend -> Frontend -> Client
  ○ Frontend, Backend -> Scheduler: stats
  ○ Scheduler -> Frontend: List of backends for round robin
  ○ Scheduler -> Backend: Duty cycle (list of model + batch size)
  ○ Backend: pick requests for the next batch; run DNN on GPU back-to-back
  ○ Scheduler, Frontend, Backend all make parts of scheduling decisions
Model Serving System: Scheduling

- **Schedule:**
  - Which GPU to run this batch?
  - Which requests are included in this batch?
  - When to start running this batch?

- **Distributed scheduling (Nexus [SOSP’19]):**
  - Scheduler, Frontend, Backend all make parts of scheduling decisions

- **Centralized scheduling (Clockwork [OSDI’20]):**
  - Client -> Frontend -> **Scheduler** -> Backend -> **Scheduler** -> Client
  - Scheduler can have precision control over backend execution
  - Frontend, Backend are simple, non-decision-making.
  - Scheduler on every request’s data path
    - Bottleneck! (Network bandwidth & CPU)
Model Serving System: Scheduling

- **Schedule:**
  - Which GPU to run this batch?
  - Which requests are included in this batch?
  - When to start running this batch?

- **Distributed scheduling (Nexus [SOSP’19]):**
  - Scheduler, Frontend, Backend all make parts of scheduling decisions

- **Centralized scheduling (Clockwork [OSDI’20]):**
  - Scheduler can have precision control over backend execution
  - Bottleneck! (Network bandwidth & CPU)

- **Centralized scheduling (Symphony [under review]):**
  - Scheduler only exchange metadata
  - Multi-core scalable scheduling algorithm
  - Better scheduling quality (bigger batch size, higher goodput under latency constraint)
Model Serving System: Scheduling

- **Notation:**
  - \( b \): batch size
  - \( l(b) \): latency of batch size \( b \)
  - \( N \): the number of GPUs

- **Variables:** \( b, N \)

- **Batching equations**
  - Total throughput > Request rate
    - \( N \cdot b/l(b) > RPS \)
  - Queuing delay + Execution < latency SLO
    - Non-coordinated: \( (1 + 1) \cdot l(b) < SLO \)
    - Coordinated: \( (1/N + 1) \cdot l(b) < SLO \)