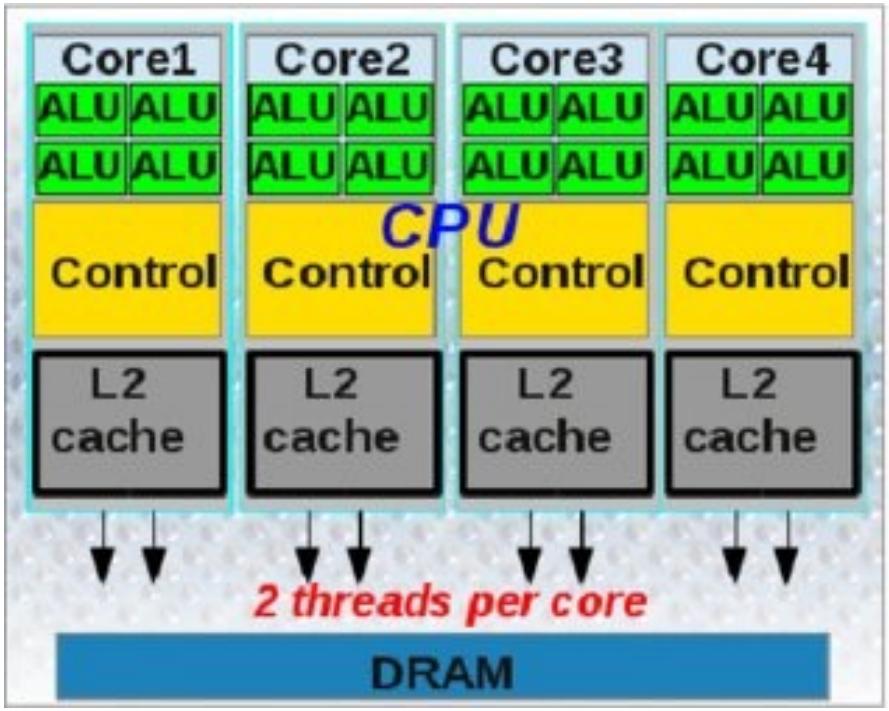
Learning Systems 2018:
Lecture 7 – Compilation

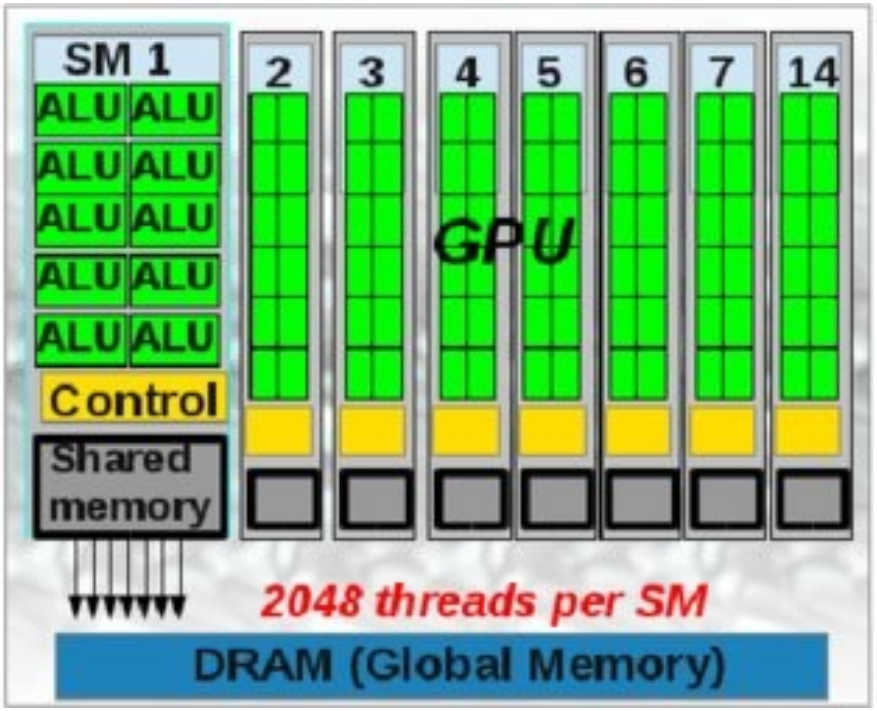
Crescat scientia; vita excolatur

Ian Foster and Rick Stevens
Argonne National Laboratory
The University of Chicago

Compilation

- Compilation targets: CPU, GPU, TPU
- Compiling deep neural networks:
  - TensorFlow XLA
  - TVM
  - nGraph



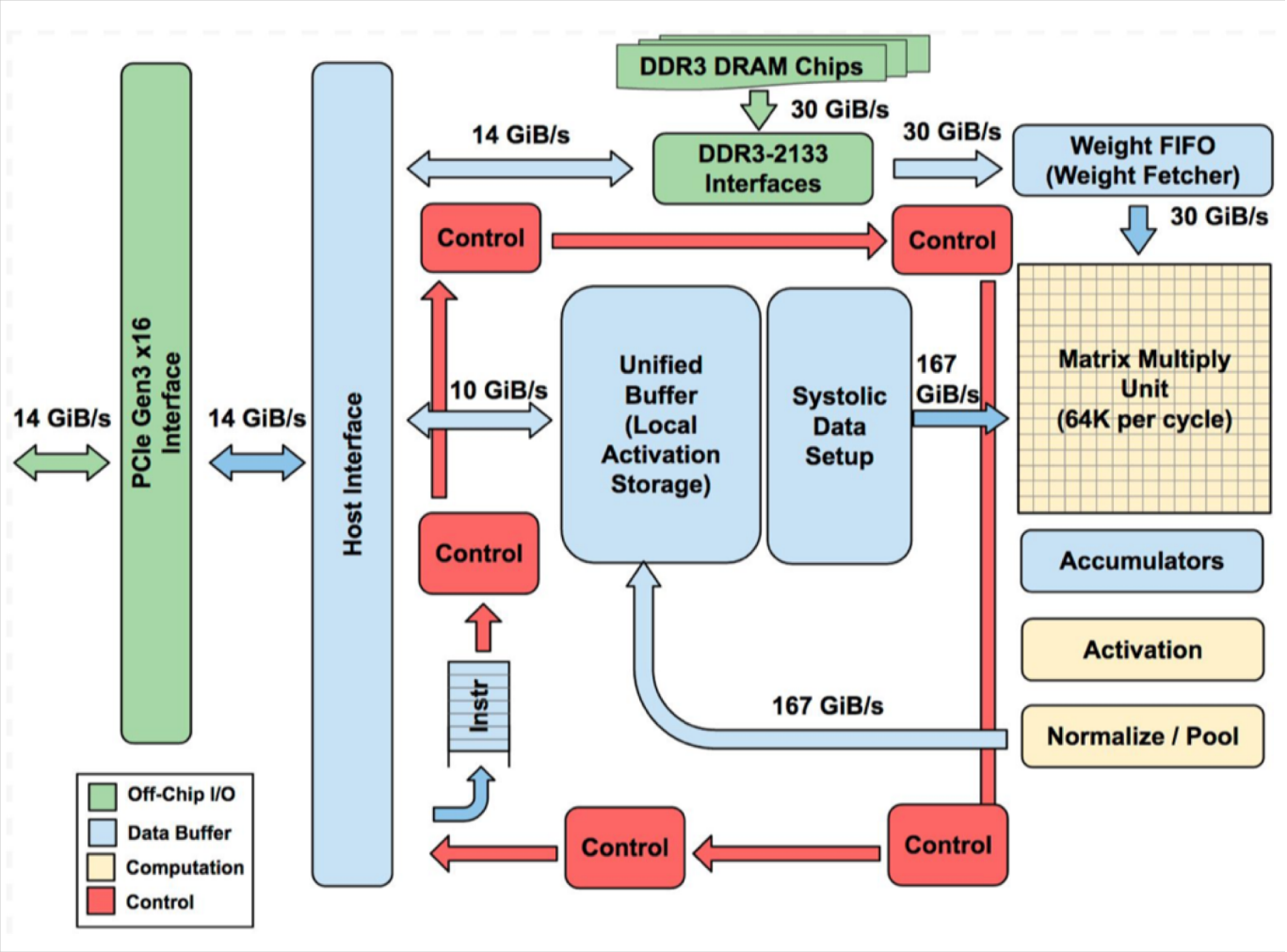


SM = Streaming Multiprocessor

Runtime system maps sets of threads (Warps) to SMs when required data are available

Google's Tensor Processing Unit

TPU Block Diagram. The main computation part is the Matrix Multiply unit in the upper right hand corner. Its inputs are the Weight FIFO and the Unified Buffer and its output is the Accumulators. The Activation Unit performs the nonlinear functions on the Accumulators, which go to the Unified Buffer.

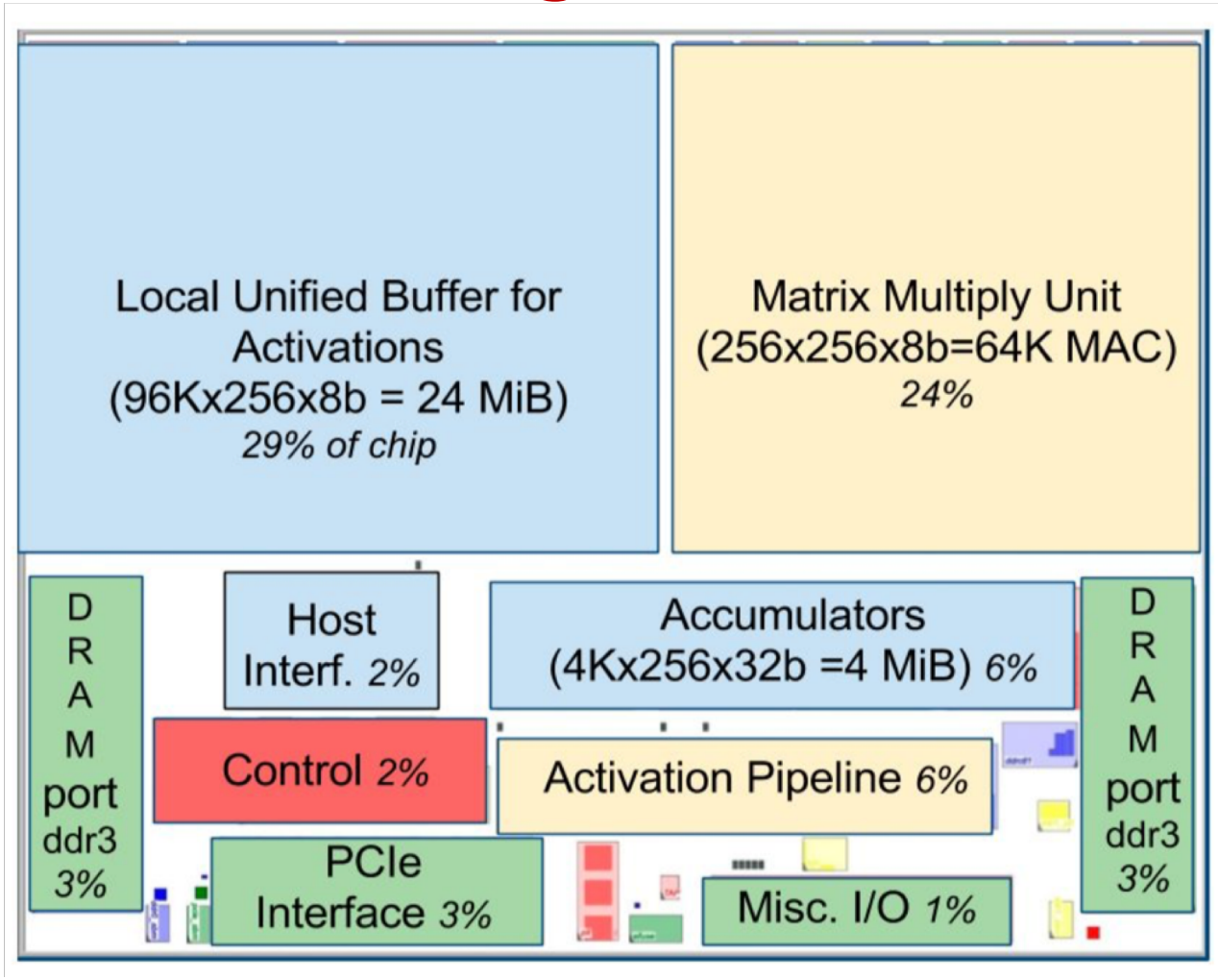


https://arxiv.org/pdf/1704.04760.pdf

Google's Tensor Processing Unit

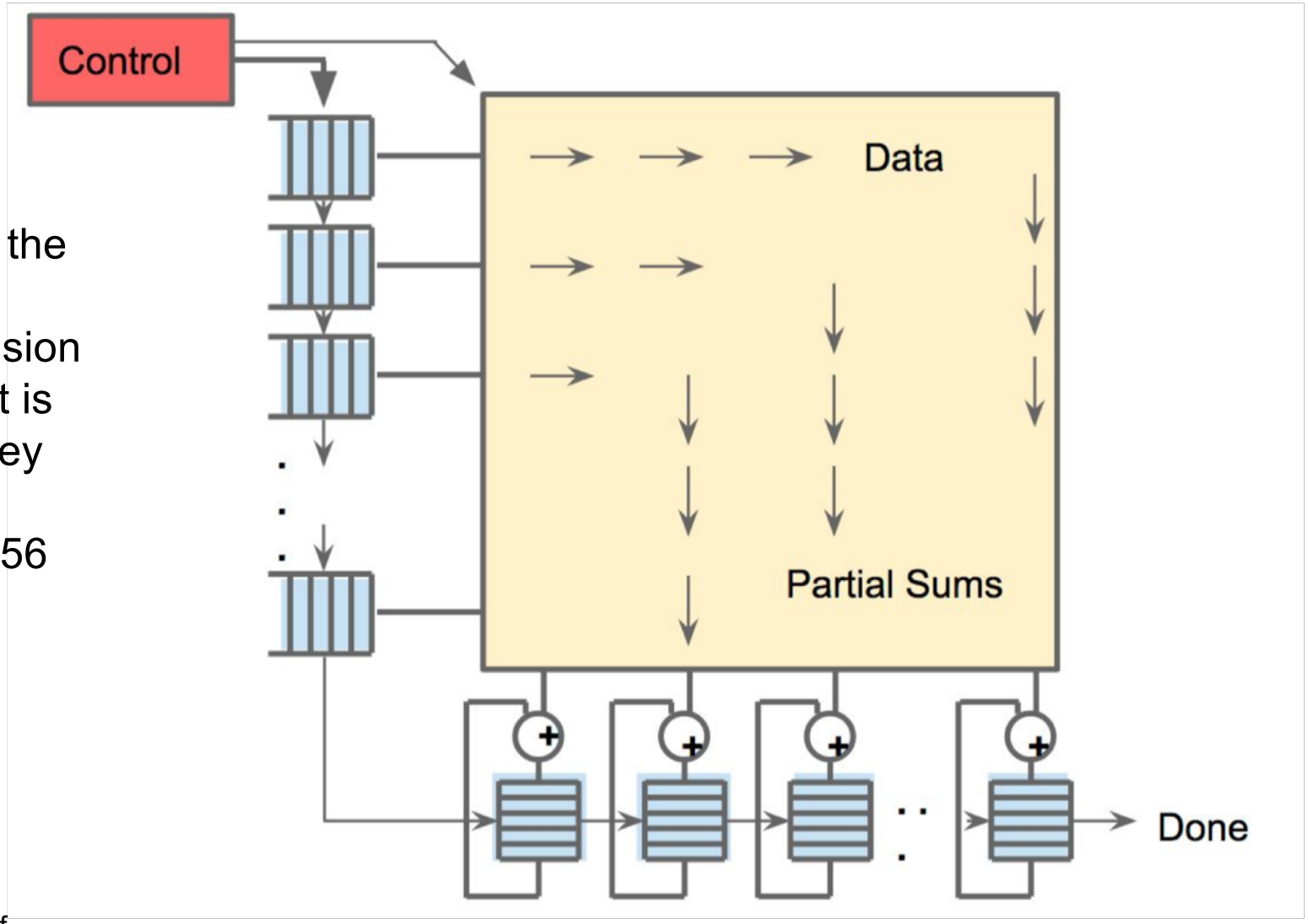
Floor Plan of TPU die. The data buffers are 37% of the die, compute is 30%, the medium (green) I/O is 10%, and the dark (red) control is just 2%. Control is much larger (and much more difficult to design) in a CPU or GPU

MAC = Multiply Accumulate Unit



https://arxiv.org/pdf/1704.04760.pdf

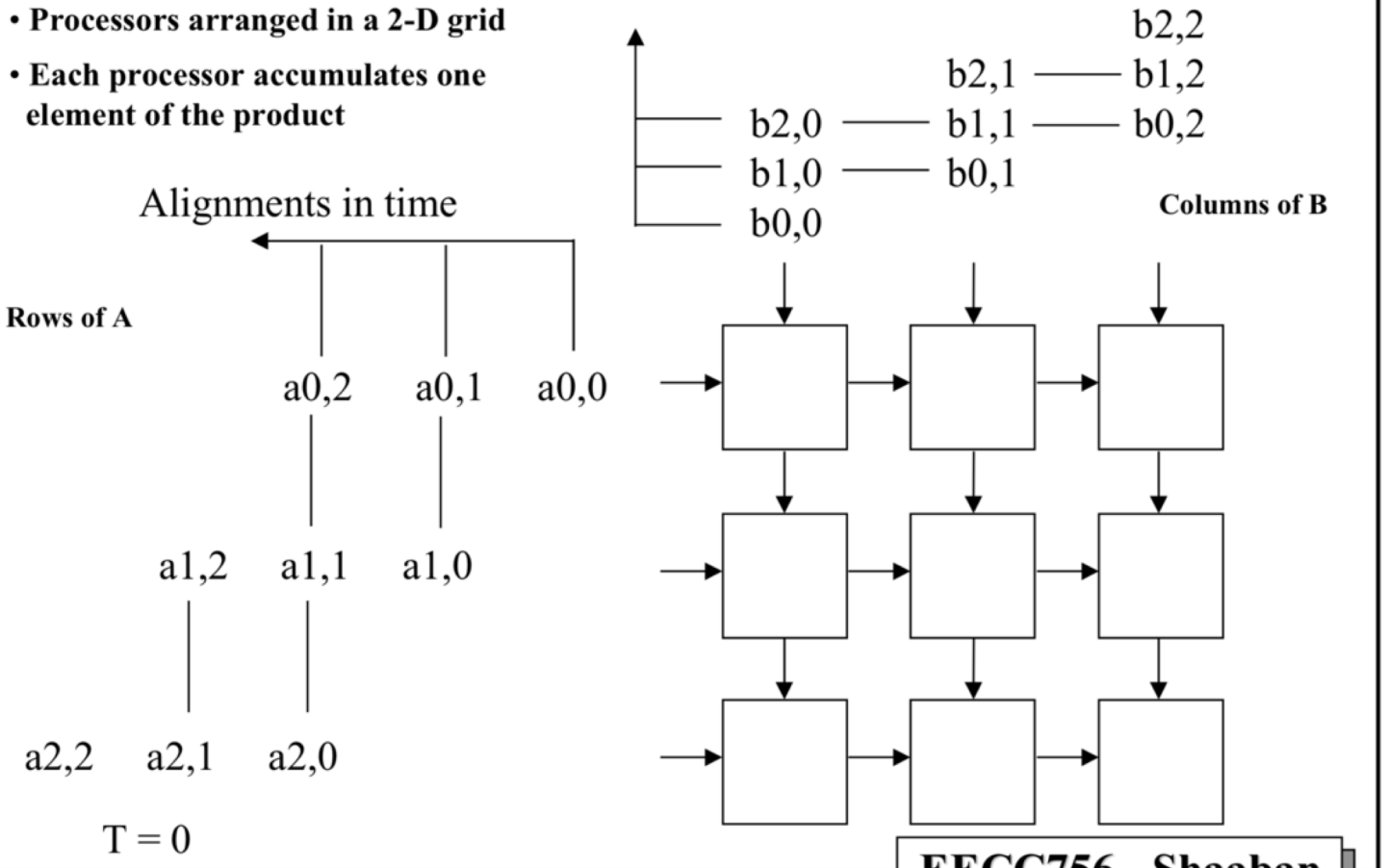
Systolic data flow of the Matrix Multiply Unit. Software has the illusion that each 256B input is read at once, and they instantly update one location of each of 256 accumulator RAMs.



https://arxiv.org/pdf/1704.04760.pdf

Systolic Array Example:
3x3 Systolic Array Matrix Multiplication

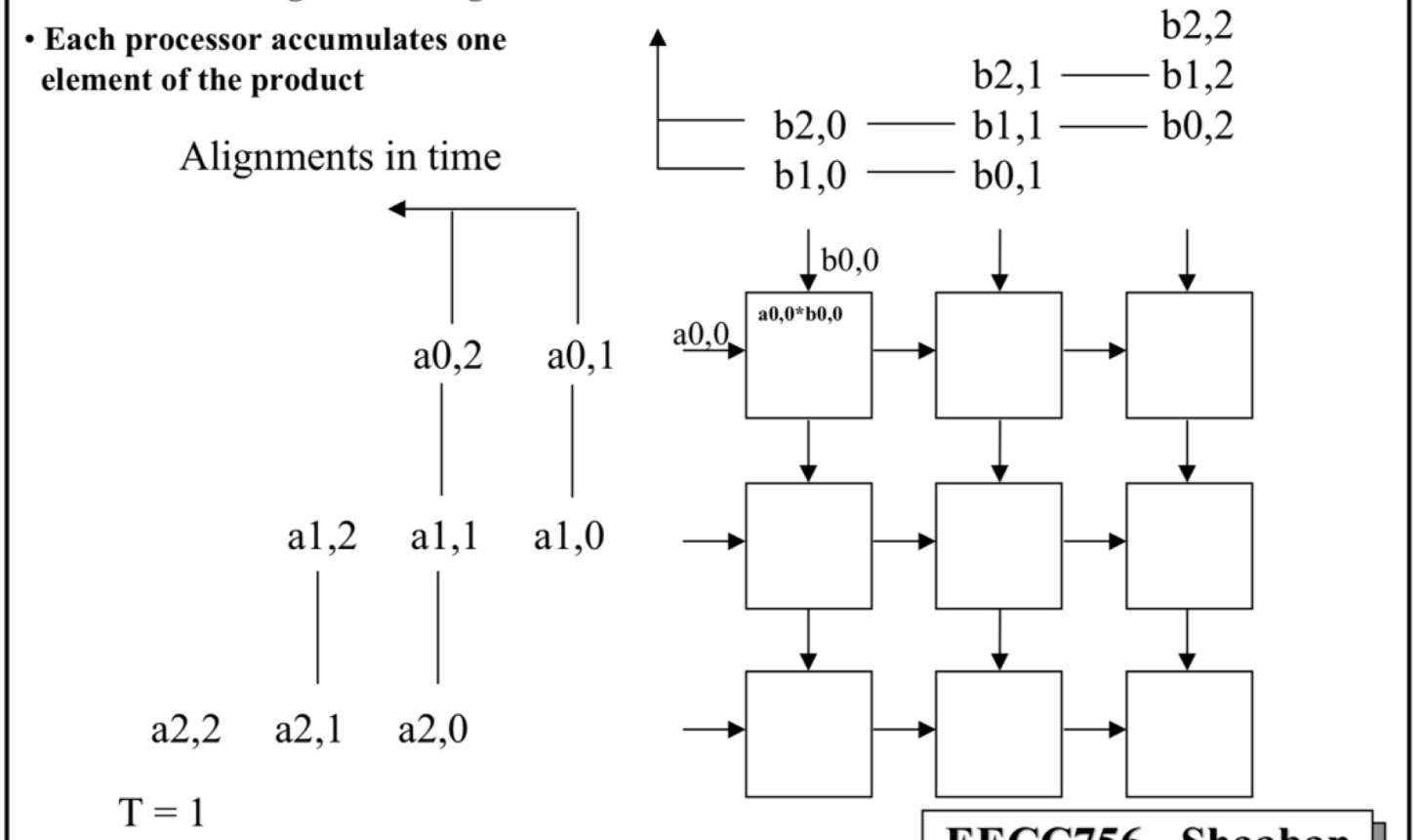
- **Processors arranged in a 2-D grid**
- **Each processor accumulates one element of the product**



Example source: http://www.cs.hmc.edu/courses/2001/spring/cs156/

Systolic Array Example:
3x3 Systolic Array Matrix Multiplication

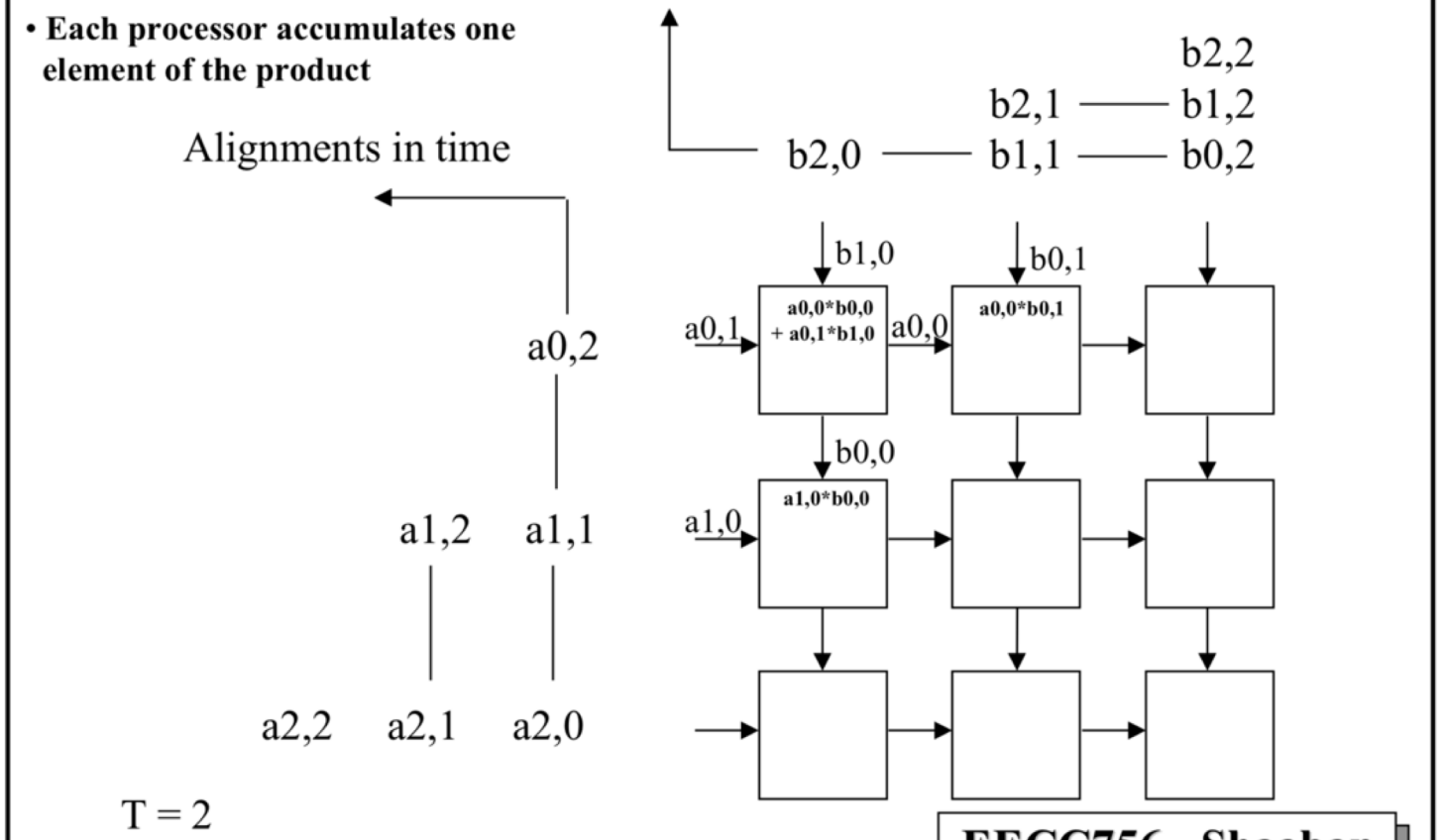
- **Processors arranged in a 2-D grid**
- **Each processor accumulates one element of the product**



Example source: http://www.cs.hmc.edu/courses/2001/spring/cs156/

Systolic Array Example:
3x3 Systolic Array Matrix Multiplication

- **Processors arranged in a 2-D grid**
- **Each processor accumulates one element of the product**

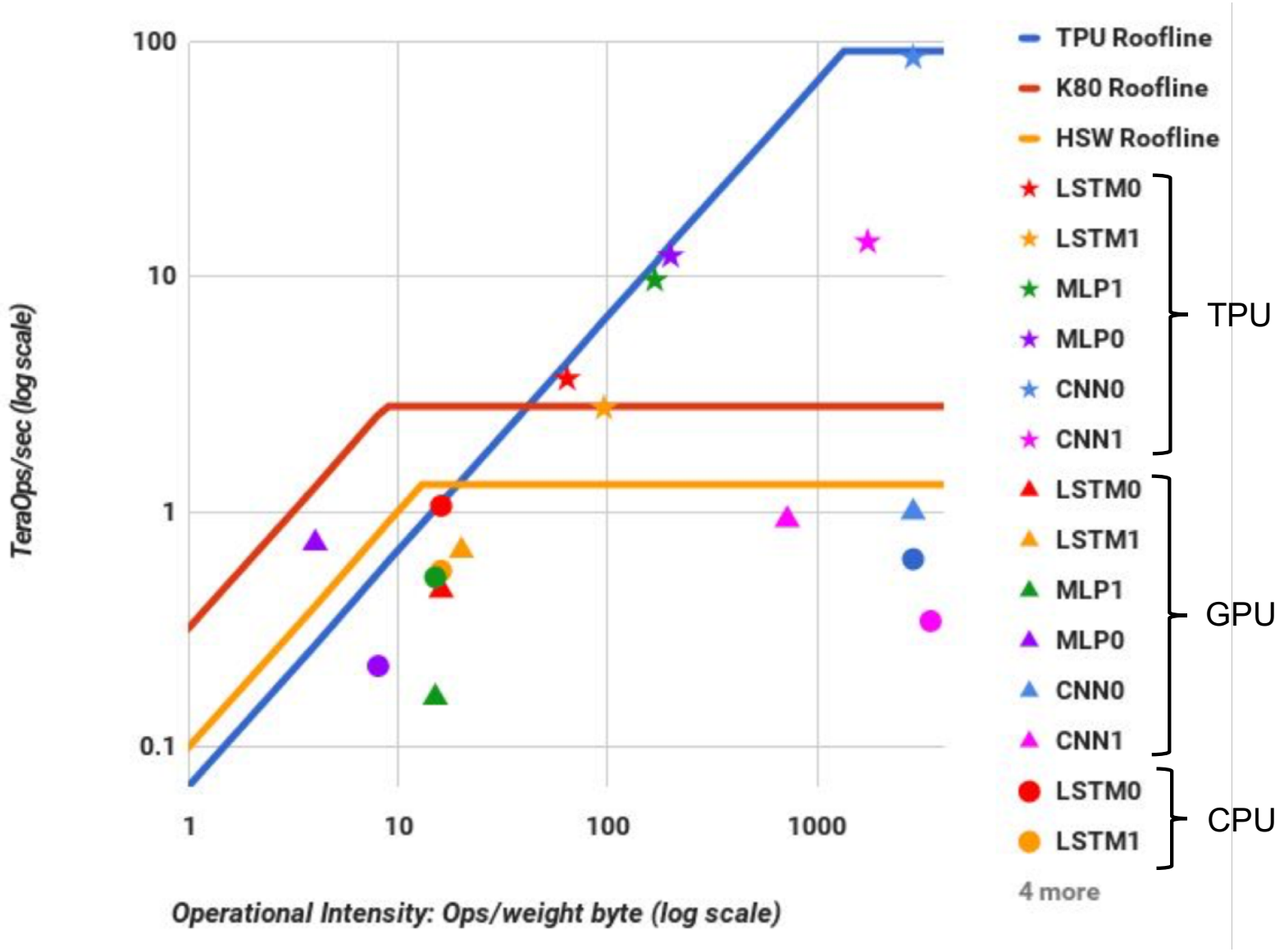


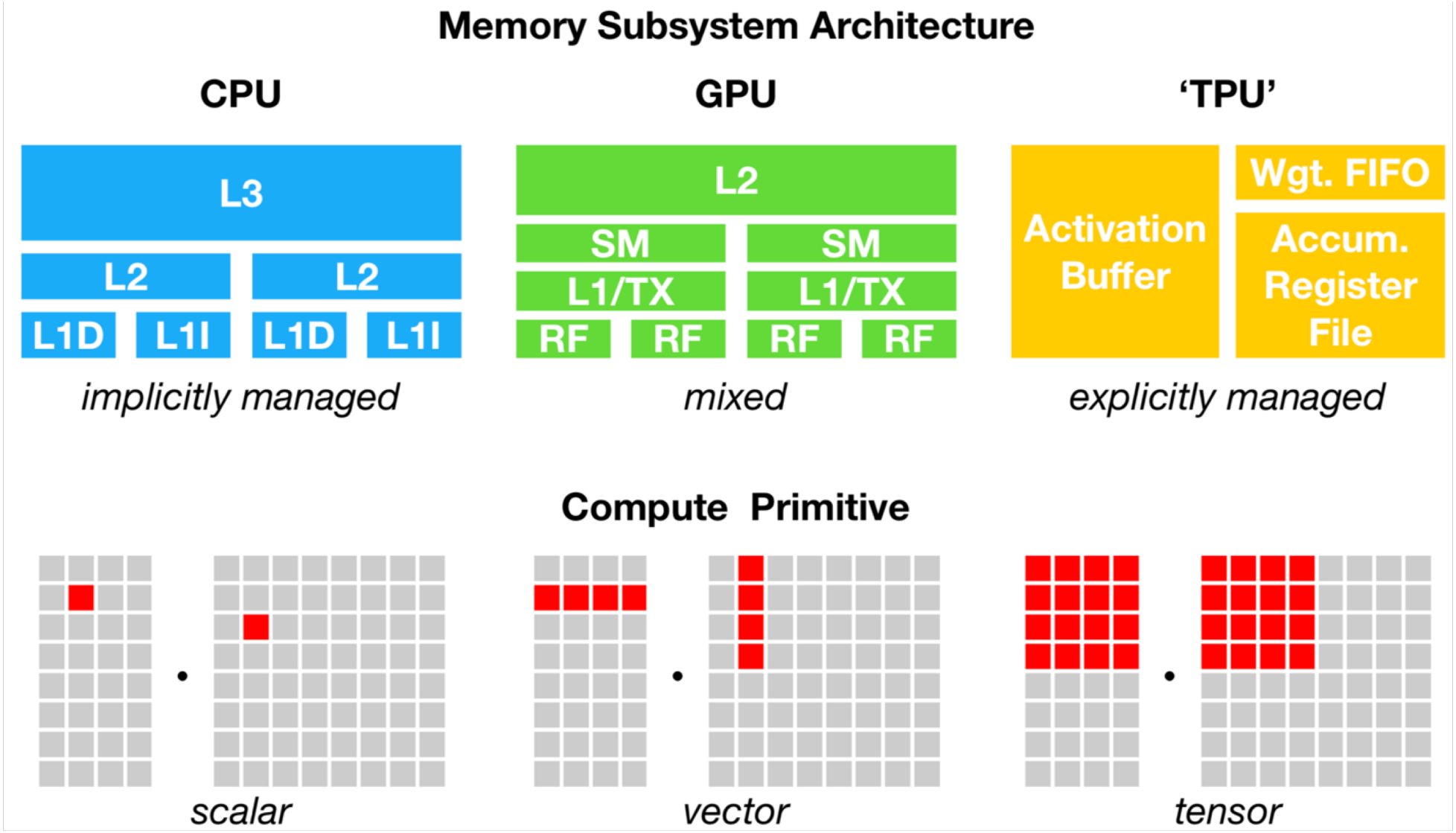
Example source: http://www.cs.hmc.edu/courses/2001/spring/cs156/

Model	*Die*										*Benchmarked Servers*				
	mm²	*nm*	*MHz*	*TDP*	*Measured*		*TOPS/s*		*GB/s*	*On-Chip Memory*	*Dies*	*DRAM Size*	*TDP*	*Measured*	
					Idle	*Busy*	8b	FP						*Idle*	*Busy*
Haswell E5-2699 v3	662	22	2300	145W	41W	145W	2.6	1.3	51	51 MiB	2	256 GiB	504W	159W	455W
NVIDIA K80 (2 dies/card)	561	28	560	150W	25W	98W	--	2.8	160	8 MiB	8	256 GiB (host) + 12 GiB x 8	1838W	357W	991W
TPU	NA*	28	700	75W	28W	40W	92	--	34	28 MiB	4	256 GiB (host) + 8 GiB x 4	861W	290W	384W

Table 2. Benchmarked servers use Haswell CPUs, K80 GPUs, and TPUs. Haswell has 18 cores, and the K80 has 13 SMX processors. Figure 10 has measured power. The low-power TPU allows for better rack-level density than the high-power GPU. The 8 GiB DRAM per TPU is Weight Memory. GPU Boost mode is not used (Sec. 8). SECDEC and no Boost mode reduce K80 bandwidth from 240 to 160. No Boost mode and single die vs. dual die performance reduces K80 peak TOPS from 8.7 to 2.8. (*The TPU die is ≤ half the Haswell die size.)

Performance on inference applications



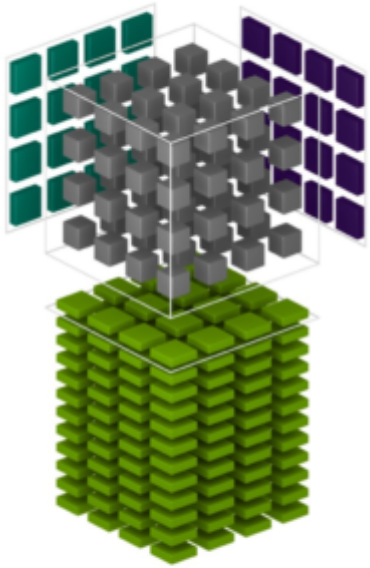


https://arxiv.org/pdf/1802.04799.pdf

Workloads not suited for TPUs

- Linear algebra programs that require frequent branching or are dominated element-wise by algebra. TPUs are optimized to perform fast, bulky matrix multiplication, so a workload that is not dominated by matrix multiplication is unlikely to perform well on TPUs compared to other platforms.
- Workloads that access memory in a sparse manner
- Workloads that require high-precision (e.g., double precision) arithmetic
- Neural network workloads that contain custom TensorFlow operations written in C++. Specifically, custom operations in the body of the main training loop are not suitable for TPUs.

GPUs fight back: NVIDIA Volta tensor cores



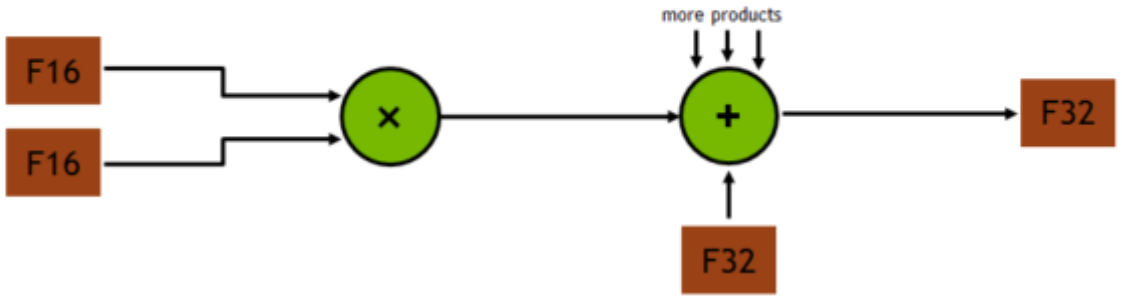
$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{matrix} \\ \\ \\ \\ \text{FP16 or FP32} \end{matrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} \begin{matrix} \\ \\ \\ \\ \text{FP16} \end{matrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix} \begin{matrix} \\ \\ \\ \\ \text{FP16 or FP32} \end{matrix}$$

FP16 storage/input

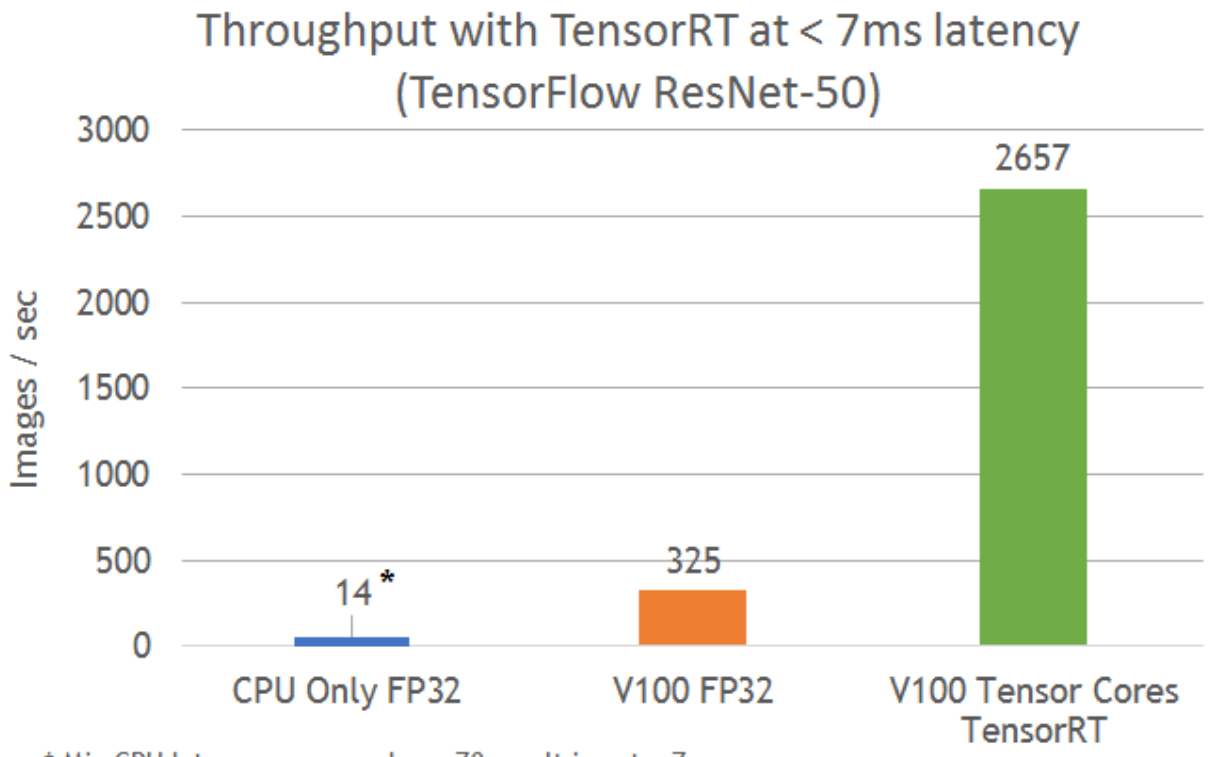
Full precision product

Sum with FP32 accumulator

Convert to FP32 result

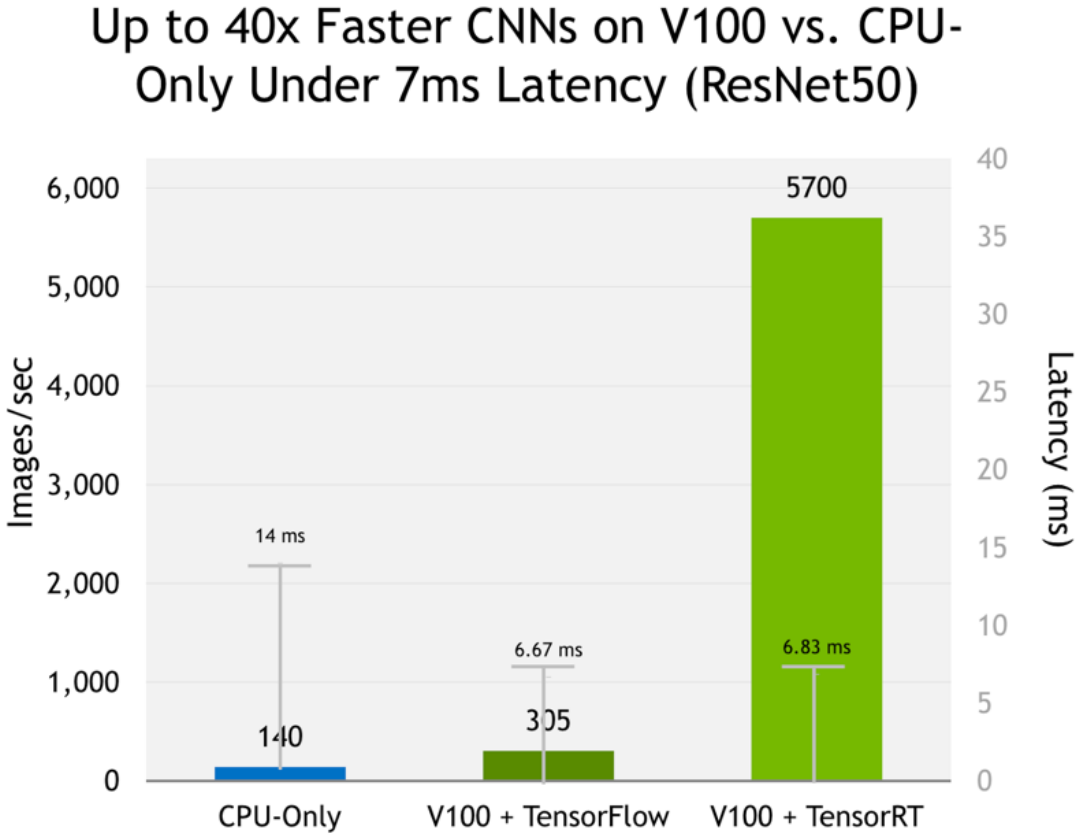


GPUs fight back: NVIDIA Volta tensor cores



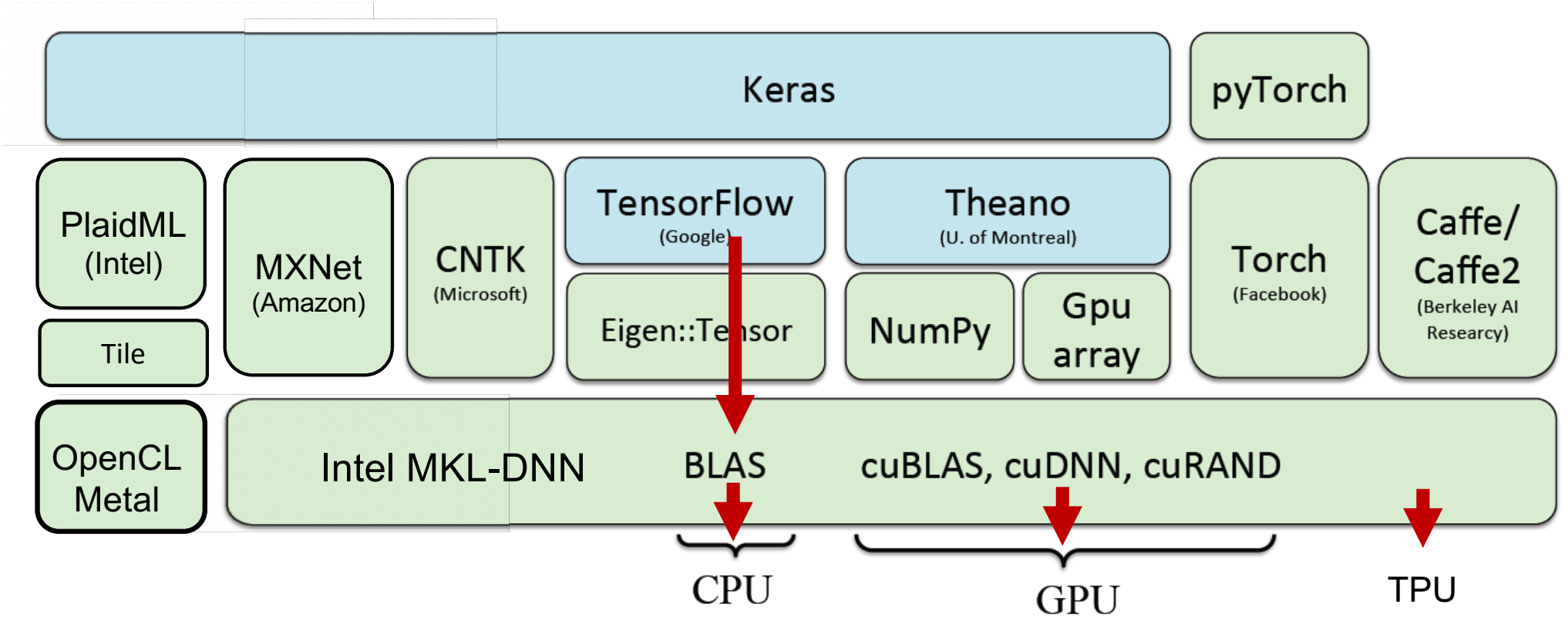
* Min CPU latency measured was 70 ms. It is not < 7 ms.
CPU: Skylake Gold 6140, 2.5GHz, Ubuntu 16.04; 18 CPU threads. Volta V100 SXM; CUDA (384.111; v9.0.176);
Batch sizes: CPU=1, V100_FP32=2, V100_TensorFlow_TensorRT=16 w/ latency=6ms

GPUs fight back: NVIDIA Volta tensor cores



Inference throughput (images/sec) on ResNet50. **V100 + TensorRT**: NVIDIA TensorRT (FP16), batch size 39, Tesla V100-SXM2-16GB, E5-2690 v4@2.60GHz 3.5GHz Turbo (Broadwell) HT On **V100 + TensorFlow**: Preview of volta optimized TensorFlow (FP16), batch size 2, Tesla V100-PCIE-16GB, E5-2690 v4@2.60GHz 3.5GHz Turbo (Broadwell) HT On. **CPU-Only:** Intel Xeon-D 1587 Broadwell-E CPU and Intel DL SDK. Score doubled to comprehend Intel's stated claim of 2x performance improvement on Skylake with AVX512.

Why compilation is important

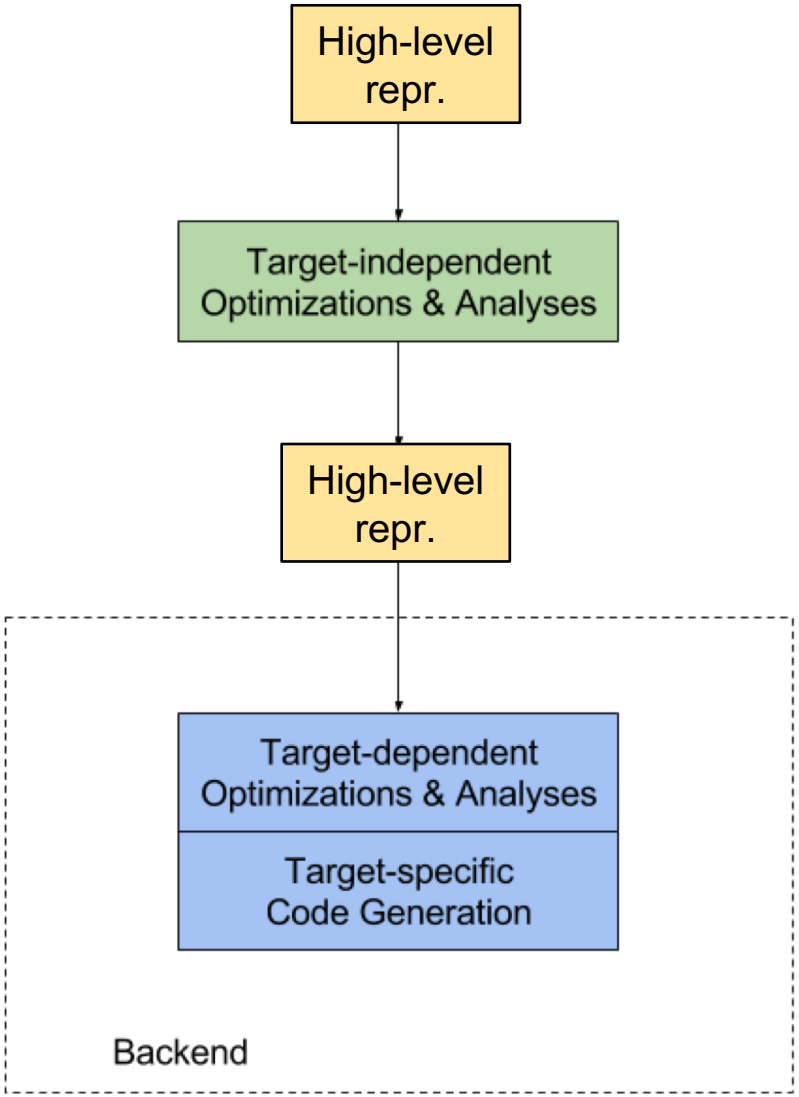


Why compilation is important

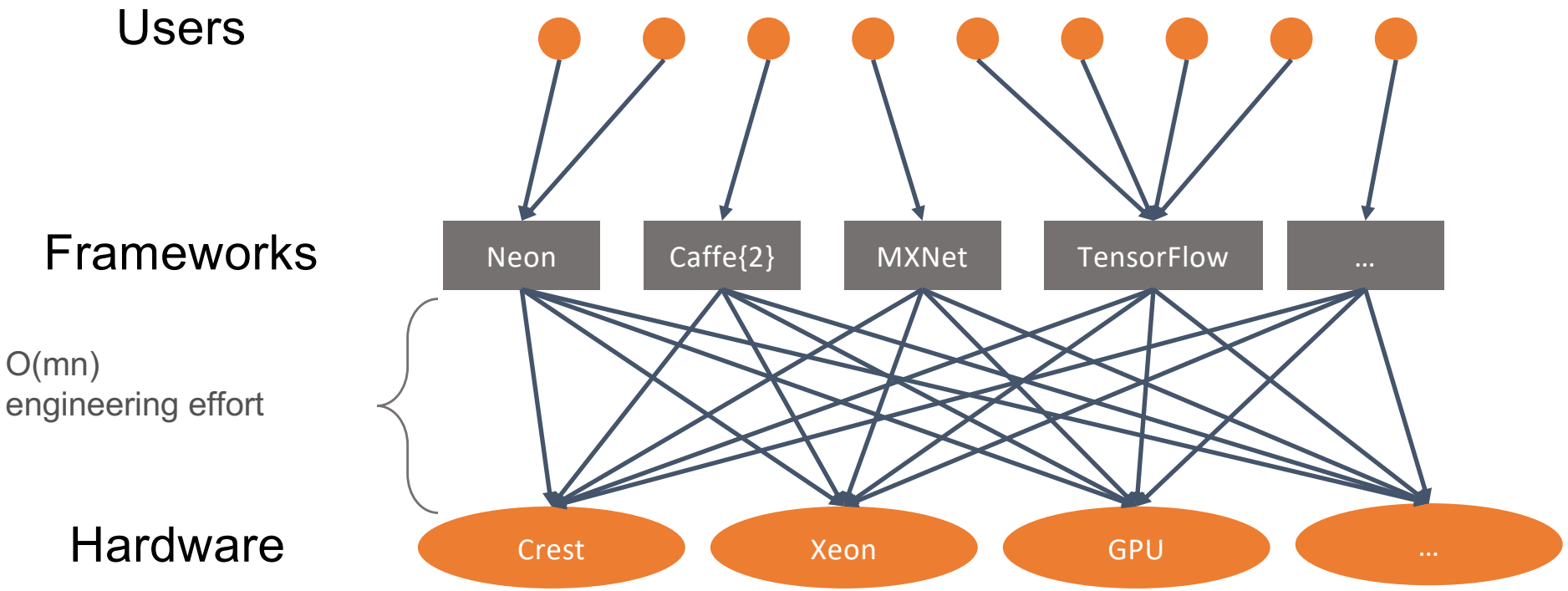
- “Traditional” approach
  - Neural networks are graphs
  - High-level language → graph → BLAS etc. kernels → CPU, GPU
- Advantages:
  - Leverage human effort optimizing linear algebra kernels
- Disadvantages:
  - Cannot leverage knowledge of tensor contents (cf sparsity)
  - Cannot “fuse” kernels
  - Odd-shaped tensors
  - Sparsity concerns
  - Variable resolution arithmetic
  - Gradients and differentiation

NN compilation: The basic idea

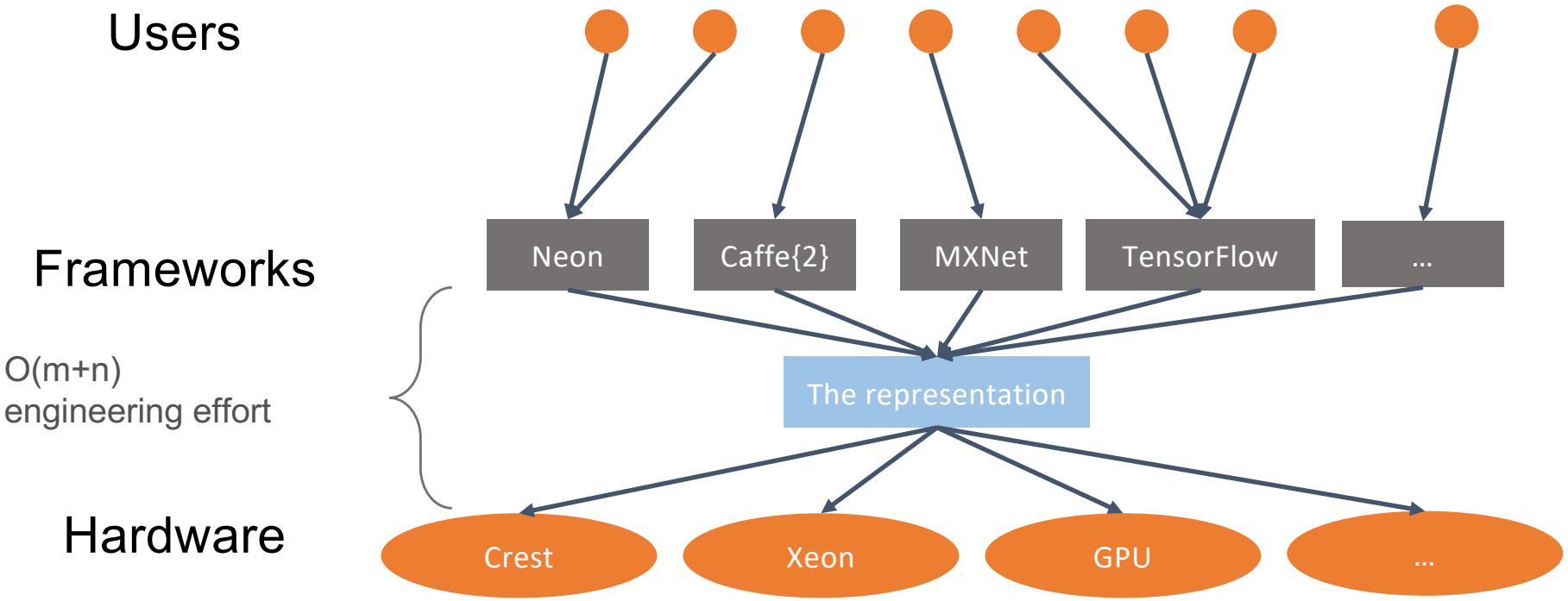
- Neural networks are programs!
- Define a good abstract, high-level representation that captures what is important for DNNs and target platforms
- Implement transformations on that representation
- Specialize for various use cases (e.g., training vs. inference)
- Specialize for various architectures
- Become rich and/or famous



Deep learning ecosystem - a many to many problem



Deep learning ecosystem - a many to many problem



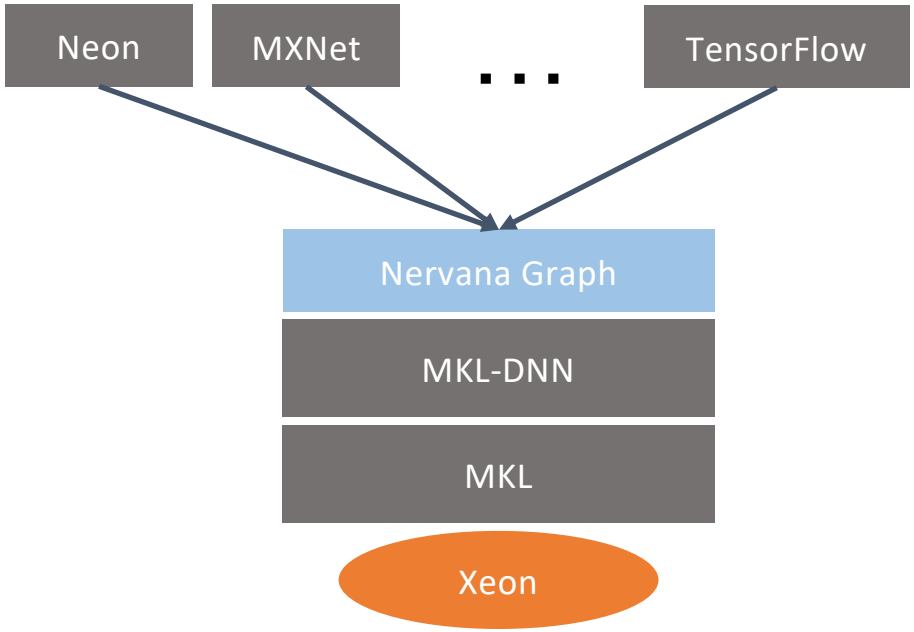
	Intel Nervana Graph	XLA	TVM	Halide	Tensor RT
Framework independence	✓		✓	✓	✓
Framework connectors	✓	✓ (only 1)	~	~	✓
Hardware independence	✓	✓	✓	✓	
Leverage existing tensor libraries	✓	✓			~
Inference and training	✓	✓*	✓	✓	
Production ready				✓	✓

	Intel Nervana Graph	XLA	TVM	Halide	Tensor RT
Framework independence	✓		✓	✓	✓
Framework connectors	✓	✓ (only 1)	~	~	✓
Hardware independence	✓	✓	✓	✓	
Leverage existing tensor libraries	✓	✓			~
Inference and training	✓	✓*	✓	✓	
Production ready				✓	✓

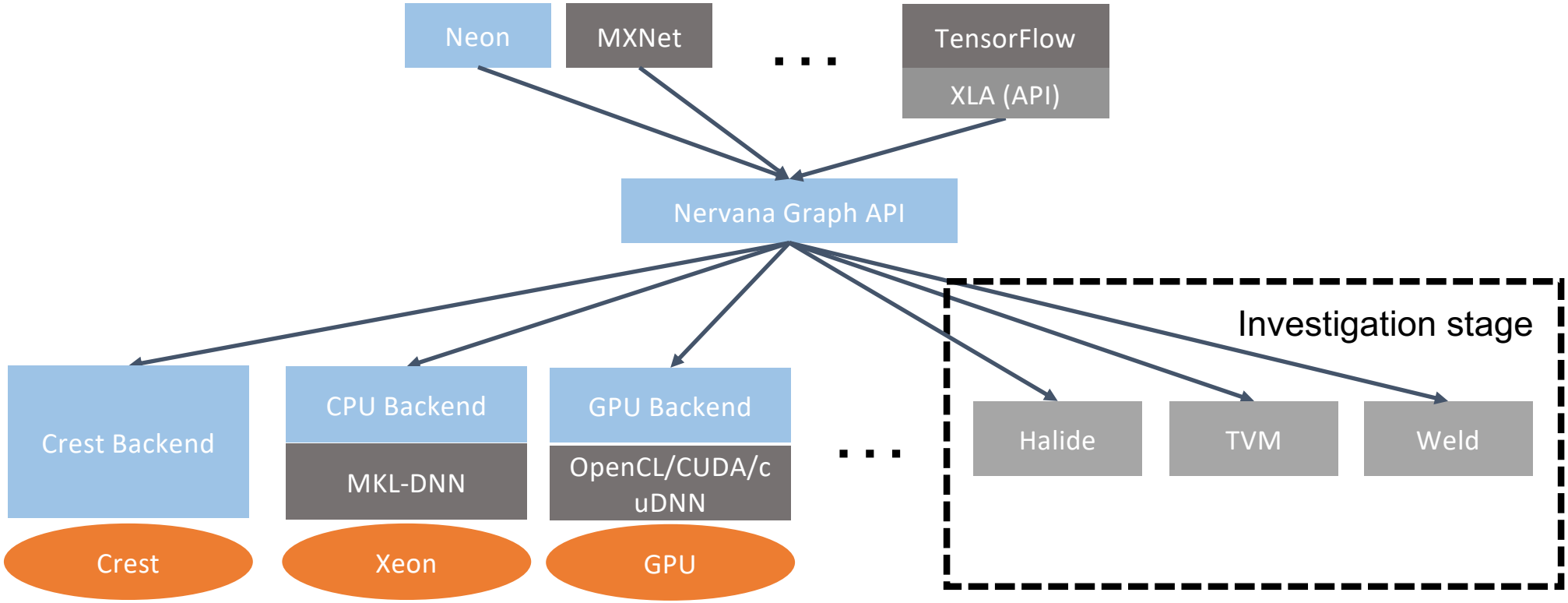
The Nervana Graph Project

- An intermediate representation (IR) for deep learning
- Compiler backends (transformers)
- Frontend “connectors”
- A reference frontend (neon)

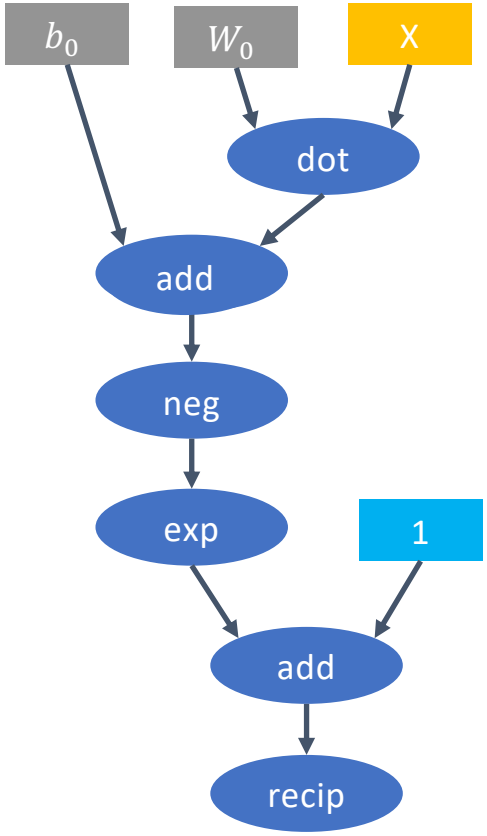
Nervana Graph – Just one more layer



Nervana Graph – The Ecosystem



Nervana Graph IR



- Dataflow graph
  - With support for control dependencies for side-effecting operations
- Rectangles are sources
  - Trainable (variable)
  - External (placeholder)
  - Constant
- Ellipses are operations
- Arrows show data inputs

Nervana Graph IR - Currently

- Try to strike a balance
  - Small enough to avoid ‘op creep’
  - Large enough to maintain performance
- Tensor math
  - Unary/binary element wise, Reductions
- Tensor manipulation (slicing, broadcasting)
- Control flow (parallel, sequential)
- Data mutation (Assignment)

Nervana Graph IR – Where we're going

- More control flow
  - Limited looping (while)
  - Iterate over selectable axes
- Reductions and maps
  - With user specified sub computations
- With the goal of:
  - Enable people to avoid writing x86 assembly, CUDA, etc and still get robust performance for new recurrent kernels and layer types

Nervana Graph Axes

- Tensor dimension management (dimshuffles, axis ordering, ...)
  - Pain point for end users
  - Difficult for device specific layout optimizations
- Nervana Graph introduces named Axes
  - Give meaningful names to axes for more meaning
  - Optional for frontends without support
  - Enables compiler to perform more static verification/analysis for the user

Graph construction example

- ```
  def sigmoid(x):
      return ng.reciprocal(ng.exp(-x) + 1)
  ```
- `X = ng.placeholder(axes=[ax.C, ax.N])`
- `Y = ng.placeholder(axes=[ax.N])`
- `W = ng.variable(axes=[ax.C], initial_value=0)`
- `b = ng.variable(axes=ax.C, initial_value=0)`
- `Y_hat = ng.sigmoid(ng.dot(W, X) + b)`
- `L = ng.cross_entropy(Y_hat, Y) / ng.batch_size(Y_hat)`
-
  ```

Working with the graph

- Find all variables that contribute to an Op
  - `Y_hat.variables()`
- Graph traversal and mutation
  - `op_list = ng.ordered_ops(cost)`
- Generate graph for the derivative of one Op with respect to another
  - `grad = ng.deriv(L, W)`
  - Uses reverse-mode autodiff backprop

Graph Transformation/Compilation

- Design inspired by the LLVM project
- Uses a series of passes to transform the graph from abstract tensor operations into an executable primitive
- Example Passes:
  - Arithmetic simplifications, eg: $\log(\exp(x)) \rightarrow x$
  - Dimension reduction and element layout
  - Storage planning
  - Maintain/exploit parallelism opportunities

Why not use existing compilers? (LLVM, ICC)

- Operations are primarily *tensor* operations
  - Tensor == Large multidimensional (often aliased) array
  - Fairly regular structure at this level
- Many optimizations at the tensor level
  - Horizontal fusion
  - Memory liveness for large tensors (rather than registers)
- Can still leverage these compilers within codegen of transformers
  - POC for C++ code gen and JIT using LLVM

How does this compare to CUDA, cuDNN, MKL-DNN?

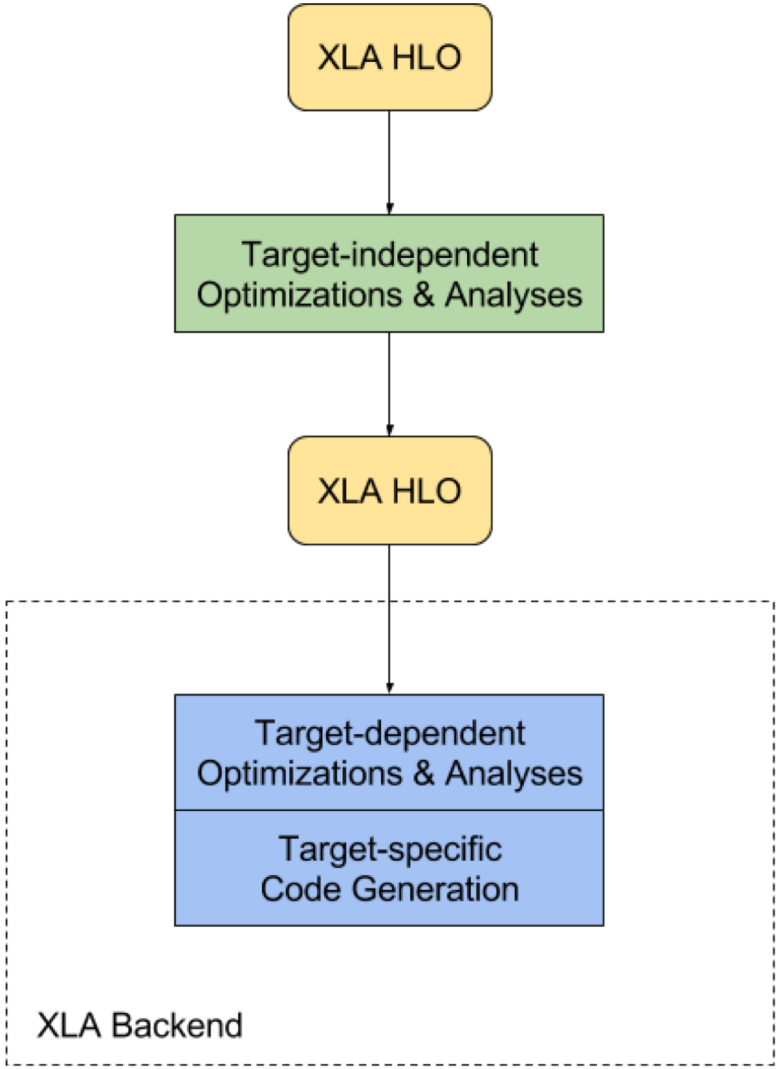
- CUDA, cuDNN, and MKL-DNN offer low levels of abstraction
  - ‘Raw’ matrix multiplies and convolutions
- Deep learning practitioners usually don’t need this amount of control
- Many ways to hurt performance when working at this level
  - Memory layout/allocation strategies
  - Operation fusion

	Intel Nervana Graph	XLA	TVM	Halide	Tensor RT
Framework independence	✓		✓	✓	✓
Framework connectors	✓	✓ (only 1)	~	~	✓
Hardware independence	✓	✓	✓	✓	
Leverage existing tensor libraries	✓	✓			~
Inference and training	✓	✓*	✓	✓	
Production ready				✓	✓


PipelineIO

TensorFlow XLA

1. Construct an XLA graph that encodes the expression we want to compute. The graph's nodes are XLA operations, and its edges represent the data flow between operations.
2. Ask XLA to create a "computation" based on this graph. This process JIT-compiles the graph into optimized native code for the chosen platform and returns a handle.
3. Use the computation handle and the input data to calculate the result.



Here is the part of the axpy sample code that constructs the graph (step 1):

Example: $\alpha x + y$

```
std::unique_ptr<xla::Literal> ComputeAxpyParameters(
    const xla::Literal& alpha, const xla::Literal& x,
    const xla::Literal& y) {
  // Get the singleton handle for an XLA client library and create a new
  // computation builder.
  xla::Client* client(xla::ClientLibrary::ClientLibraryOrDie());
  xla::ComputationBuilder builder(client, "axpy");

  // Build the actual XLA computation graph. It's a function taking
  // three parameters and computing a single output.
  auto param_alpha = builder.Parameter(0, alpha.shape(), "alpha");
  auto param_x = builder.Parameter(1, x.shape(), "x");
  auto param_y = builder.Parameter(2, y.shape(), "y");
  auto axpy = builder.Add(builder.Mul(param_alpha, param_x), param_y);
```

Here is the part that JIT-compiles the graph (step 2):

```
// We're done building the graph. Create a computation on the server.
util::StatusOr<std::unique_ptr<xla::Computation>> computation_status =
    builder.Build();
std::unique_ptr<xla::Computation> computation =
    computation_status.ConsumeValueOrDie();
```

Here is the part that runs the compiled code on the input (step 3):

```
  // Transfer the parameters to the server and get data handles that refer to
  // them.
  std::unique_ptr<xla::GlobalData> alpha_data =
      client->TransferToServer(alpha).ConsumeValueOrDie();
  std::unique_ptr<xla::GlobalData> x_data =
      client->TransferToServer(x).ConsumeValueOrDie();
  std::unique_ptr<xla::GlobalData> y_data =
      client->TransferToServer(y).ConsumeValueOrDie();

  // Now we have all we need to execute the computation on the device. We get
  // the result back in the form of a Literal.
  util::StatusOr<std::unique_ptr<xla::Literal>> result_status =
      client->ExecuteAndTransfer(
          *computation, {alpha_data.get(), x_data.get(), y_data.get()});
  return result_status.ConsumeValueOrDie();
}
```

Alternative version where alpha, x, and y are constants

```
std::unique_ptr<xla::Literal> ComputeAxpyConstants(
    float alpha, gtl::ArraySlice<float> x,
    gtl::ArraySlice<float> y) {
  // Get the singleton handle for an XLA client library and create a new
  // computation builder.
  xla::Client* client(xla::ClientLibrary::ClientLibraryOrDie());
  xla::ComputationBuilder builder(client, "axpy");

  auto constant_alpha = builder.ConstantR0<float>(alpha);
  auto constant_x = builder.ConstantR1<float>(x);
  auto constant_y = builder.ConstantR1<float>(y);
  auto axpy = builder.Add(builder.Mul(constant_alpha, constant_x), constant_y);

  // We're done building the graph. Tell the server to create a Computation from
  // it, and then execute this computation on the device, transferring the
  // result back as a literal.
  util::StatusOr<std::unique_ptr<xla::Computation>> computation_status =
      builder.Build();
  std::unique_ptr<xla::Computation> computation =
      computation_status.ConsumeValueOrDie();
  // No need to pass arguments into the computation since it accepts no
  // parameters.
  util::StatusOr<std::unique_ptr<xla::Literal>> result_status =
      client->ExecuteAndTransfer(*computation, {});
  return result_status.ConsumeValueOrDie();
}
```

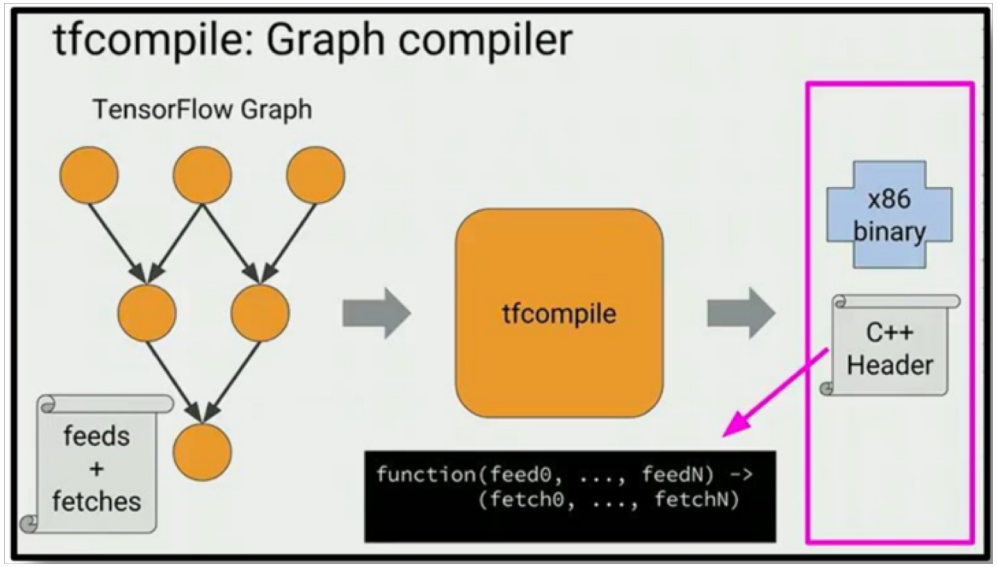
Controlling layout and padding

The `Layout` proto describes how an array is represented in memory. The `Layout` proto includes the following fields:

```
message Layout {
  repeated int64 minor_to_major = 1;
  repeated int64 padded_dimensions = 2;
  optional PaddingValue padding_value = 3;
}
```

Ahead-of-time (AOT) compiler: tfcompile

- Built on XLA framework
- Creates executable with minimal TensorFlow Runtime needed
- Includes only dependencies needed by subgraph computation
- Creates functions with feeds (inputs) and fetches (outputs)
- Packaged as cc_libary header and object files to link into your app
- Commonly used for mobile device inference graph
- Currently, only CPU x86-64 and ARM are supported - no GPU

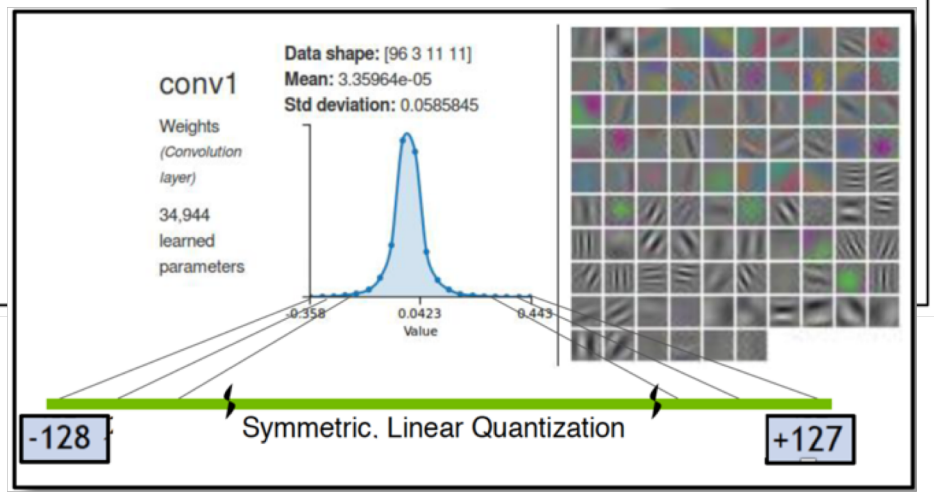


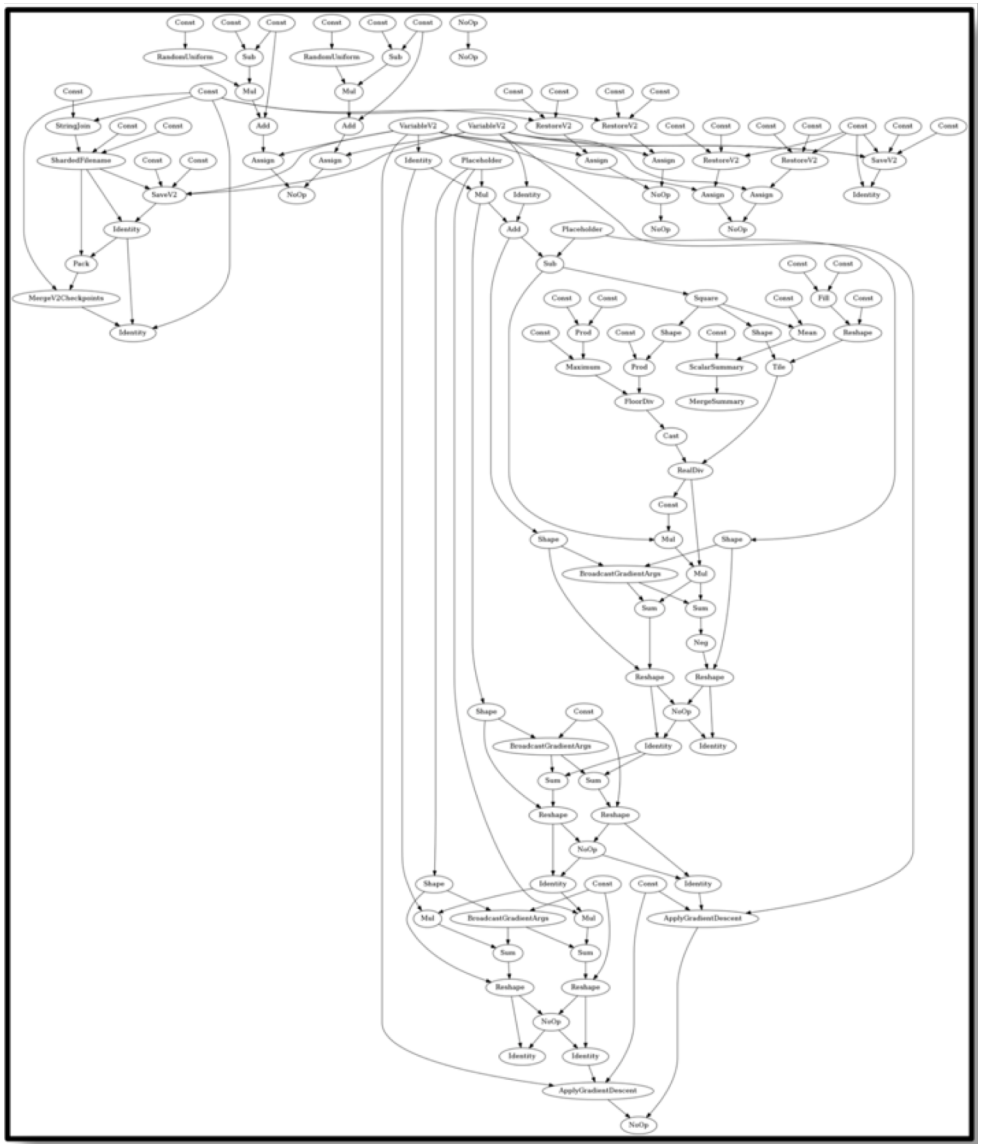
Graph Transformation Tool

- Post-training optimization to prepare for inference
- Remove training-only operations (checkpoint, drop out, logs)
- Remove unreachable nodes between given feed → fetch
- Fuse adjacent operators to improve memory bandwidth
- Fold final batch norm mean and variance into ariables
- Round weights/variables to improve compression (i.e., 70%)
- Quantize (FP32 → INT8) to speed up math operations

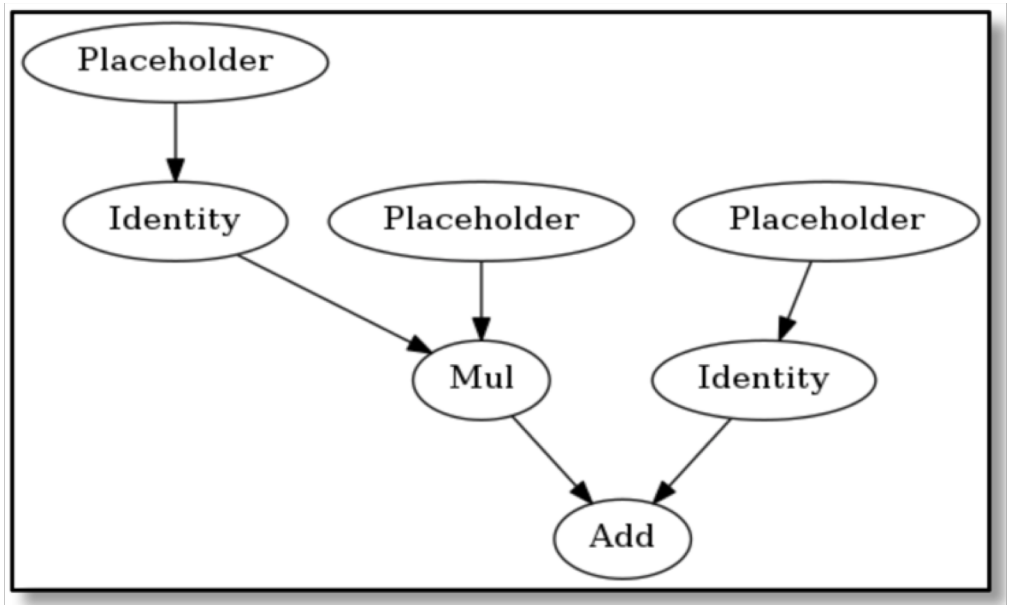
Graph Transformation Tool

```
transform_graph \
--in_graph=tensorflow_inception_graph.pb \ ← Original Graph
--out_graph=optimized_inception_graph.pb \ ← Transformed Graph
--inputs='Mul' \                           ← Feed (Input)
--outputs='softmax' \                      ← Fetch (Output)
--transforms='                             ← List of Transforms
strip_unused_nodes
remove_nodes(op=Identity, op=CheckNumerics)
fold_constants(ignore_errors=true)
fold_batch_norms
fold_old_batch_norms
quantize_weights
quantize_nodes'
```



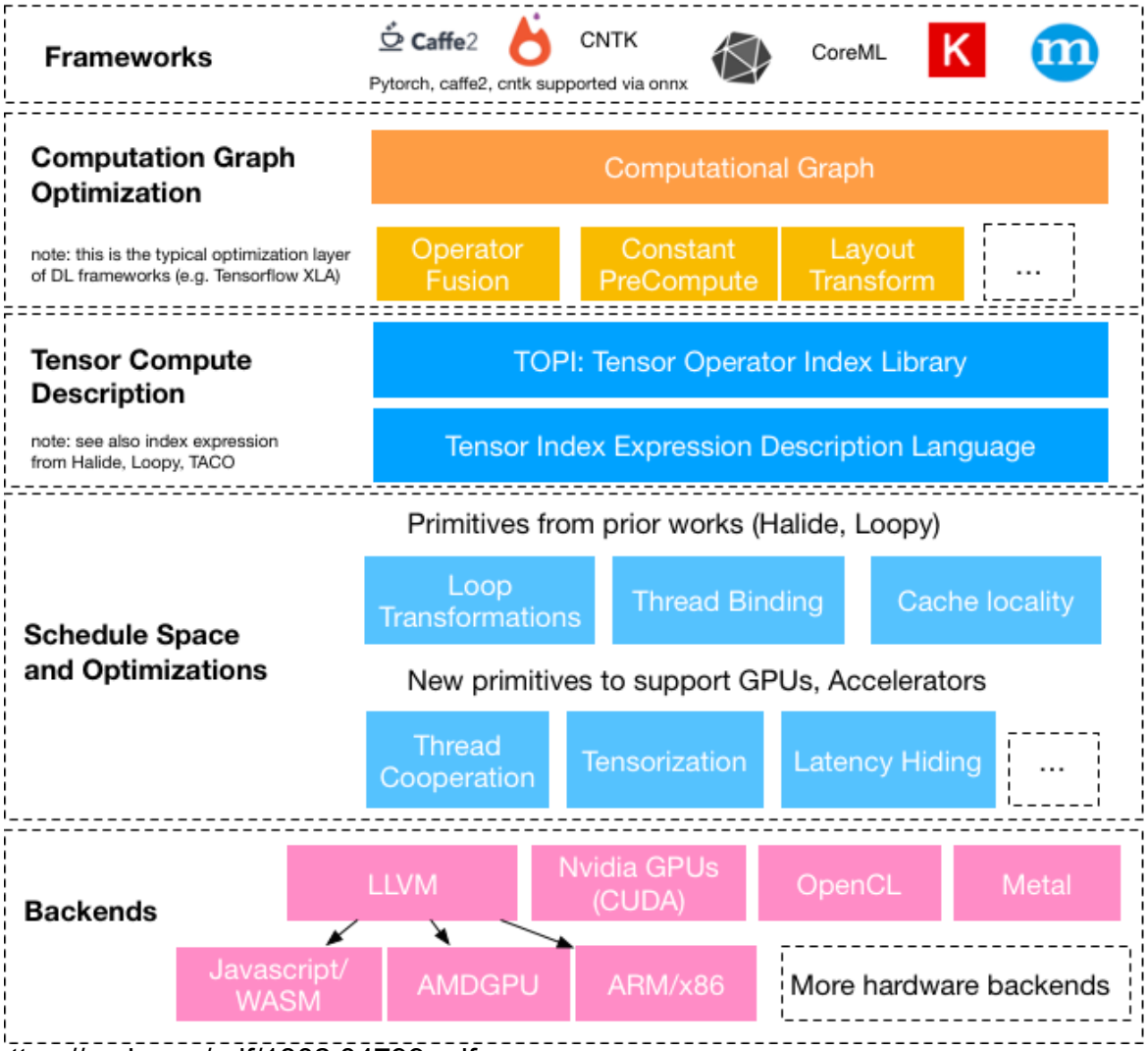


Graph Transformation Tool



	Intel Nervana Graph	XLA	TVM	Halide	Tensor RT
Framework independence	✓		✓	✓	✓
Framework connectors	✓	✓ (only 1)	~	~	✓
Hardware independence	✓	✓	✓	✓	
Leverage existing tensor libraries	✓	✓			~
Inference and training	✓	✓*	✓	✓	
Production ready				✓	✓

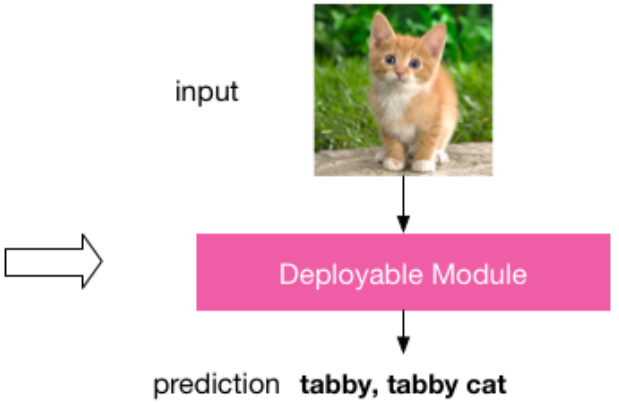
TVM Stack



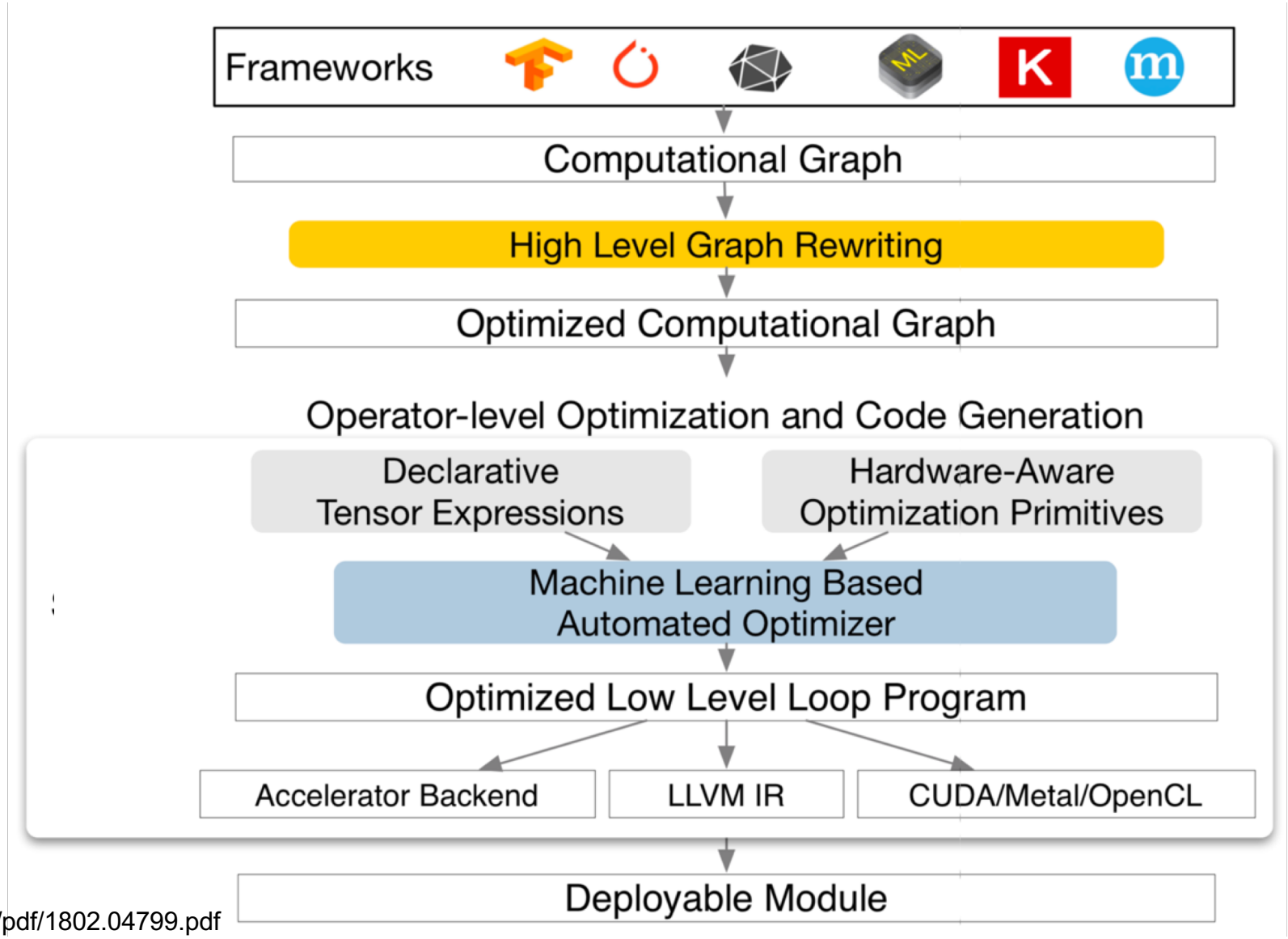
https://arxiv.org/pdf/1802.04799.pdf

Runtime: Lightweight and Cross Platform

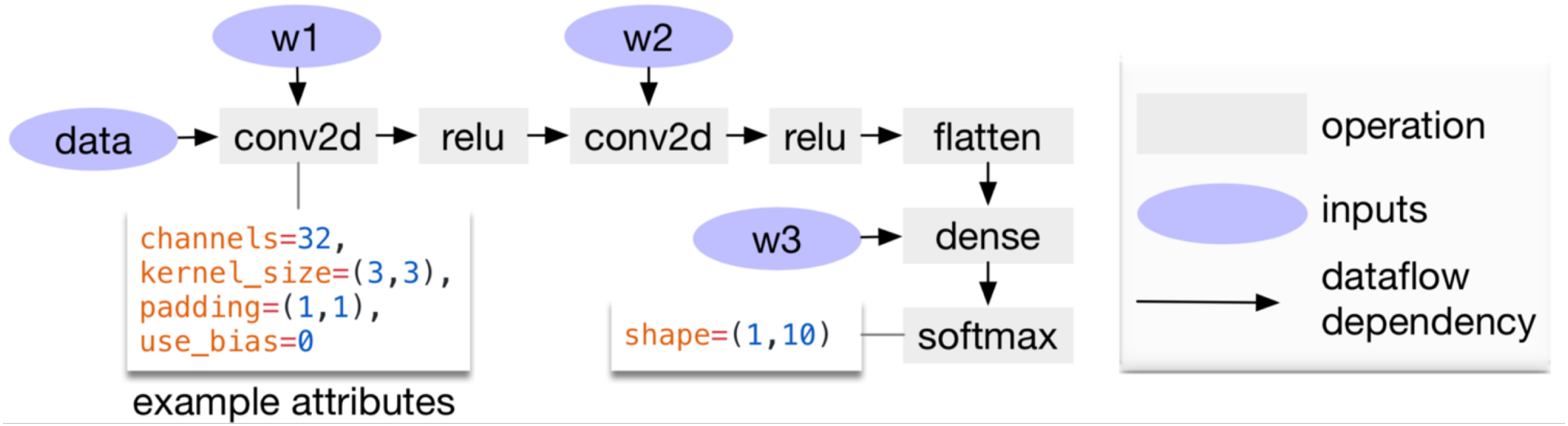
```
module = runtime.create(graph, lib, tvm.gpu(0))
module.set_input(**params)
module.run(data=data_array)
output = tvm.nd.empty(out_shape, ctx=tvm.gpu(0))
module.get_output(0, output)
```



Deploy Languages and Platforms



https://arxiv.org/pdf/1802.04799.pdf

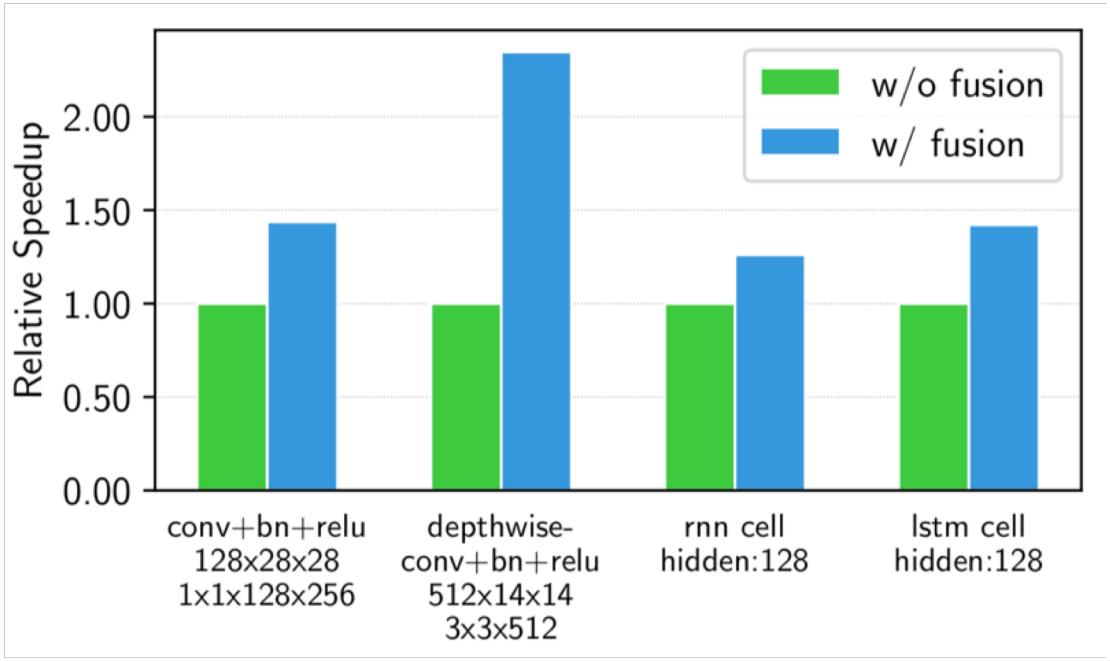


Example computational graph of a two-layer convolutional neural network. Each node in the graph represents an operation that consumes one or more tensors and produces one or more tensors. Tensor operations can be parameterized by attributes to configure their behavior (e.g., padding or strides).

https://arxiv.org/pdf/1802.04799.pdf

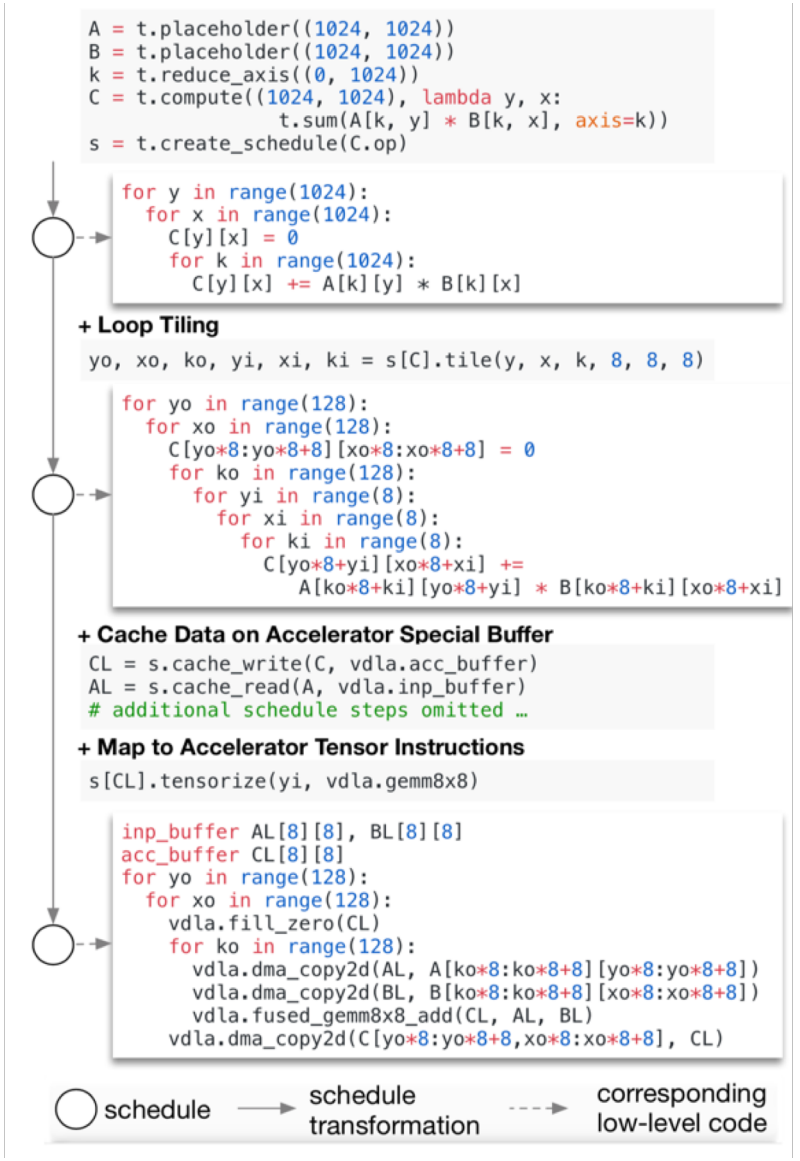
Example graph-level optimizations

- **Operator fusion**: fuse multiple small operations
- **Constant-folding**: pre-compute graph parts that can be determined statically,
- **Static memory planning**: pre-allocate memory to hold each intermediate tensor
- **Data layout transformations**: transform internal data layouts into back-end-friendly forms

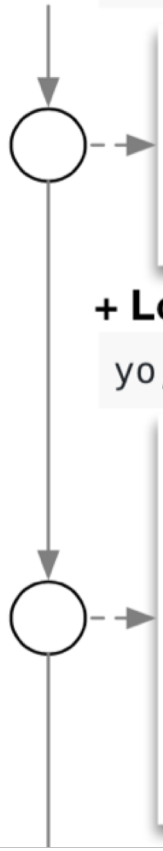


On NVIDIA Titan X

Example schedule transformations that optimize a matrix multiplication on a specialized accelerator



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



```
for y in range(1024):
  for x in range(1024):
    C[y][x] = 0
    for k in range(1024):
      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)
```

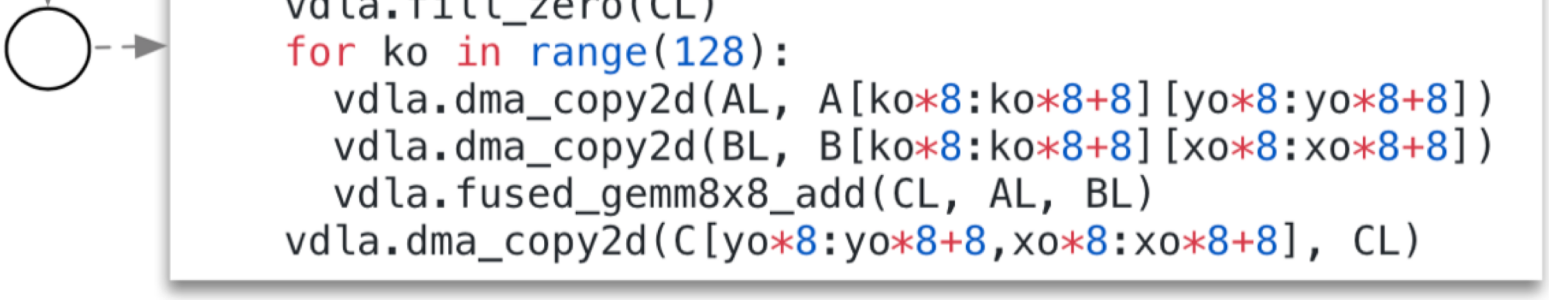
```
for yo in range(128):
  for xo in range(128):
    C[yo*8:yo*8+8][xo*8:xo*8+8] = 0
    for ko in range(128):
      for yi in range(8):
        for xi in range(8):
          for ki in range(8):
            C[yo*8+yi][xo*8+xi] +=
               A[ko*8+ki][yo*8+yi] * B[ko*8+ki][xo*8+xi]
```

+ Cache Data on Accelerator Special Buffer

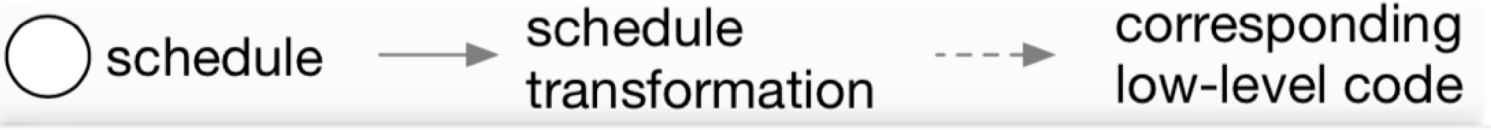
```
CL = s.cache_write(C, vdla.acc_buffer)
AL = s.cache_read(A, vdla.inp_buffer)
# additional schedule steps omitted …
```

+ Map to Accelerator Tensor Instructions

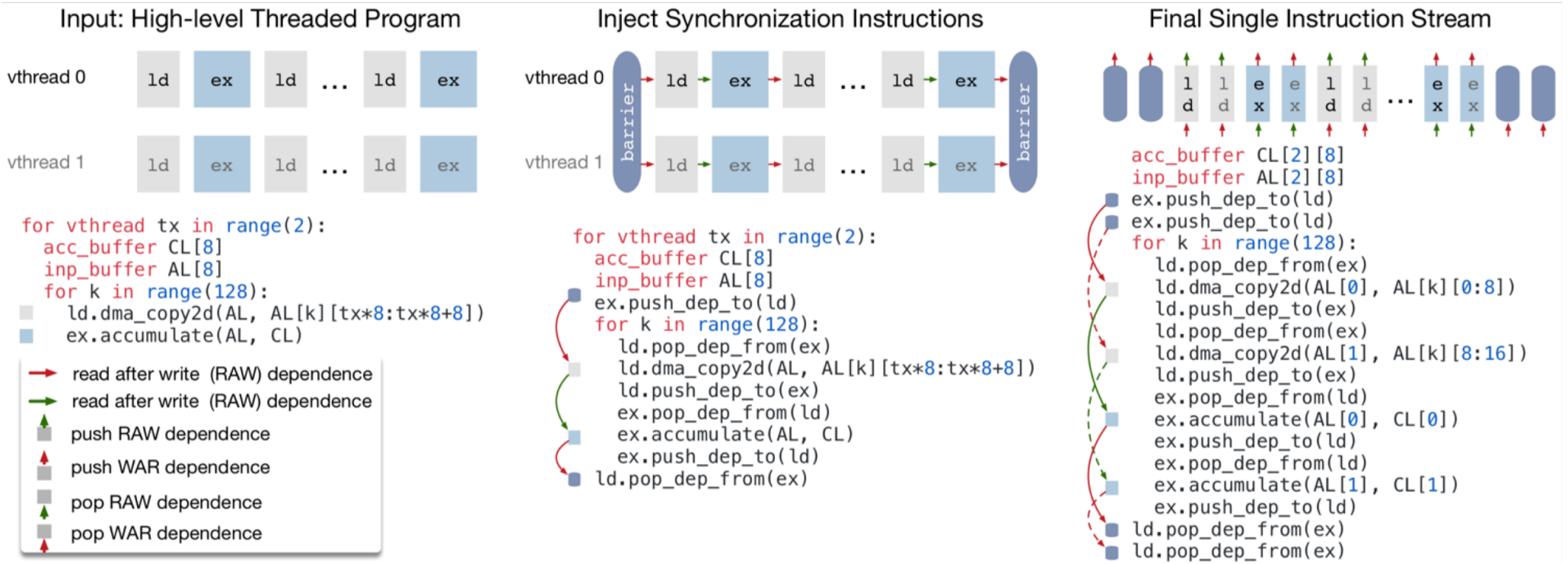
```
s[CL].tensorize(yi, vdla.gemm8x8)
```



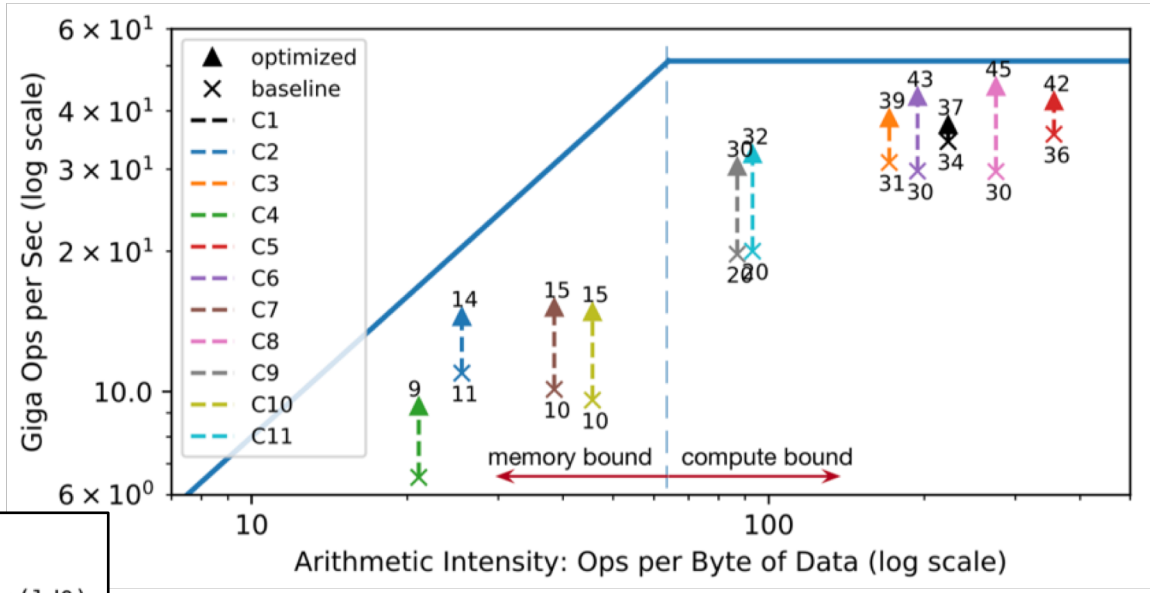
```
inp_buffer AL[8][8], BL[8][8]
acc_buffer CL[8][8]
for yo in range(128):
  for xo in range(128):
    vdla.fill_zero(CL)
    for ko in range(128):
      vdla.dma_copy2d(AL, A[ko*8:ko*8+8][yo*8:yo*8+8])
      vdla.dma_copy2d(BL, B[ko*8:ko*8+8][xo*8:xo*8+8])
      vdla.fused_gemm8x8_add(CL, AL, BL)
    vdla.dma_copy2d(C[yo*8:yo*8+8,xo*8:xo*8+8], CL)
```

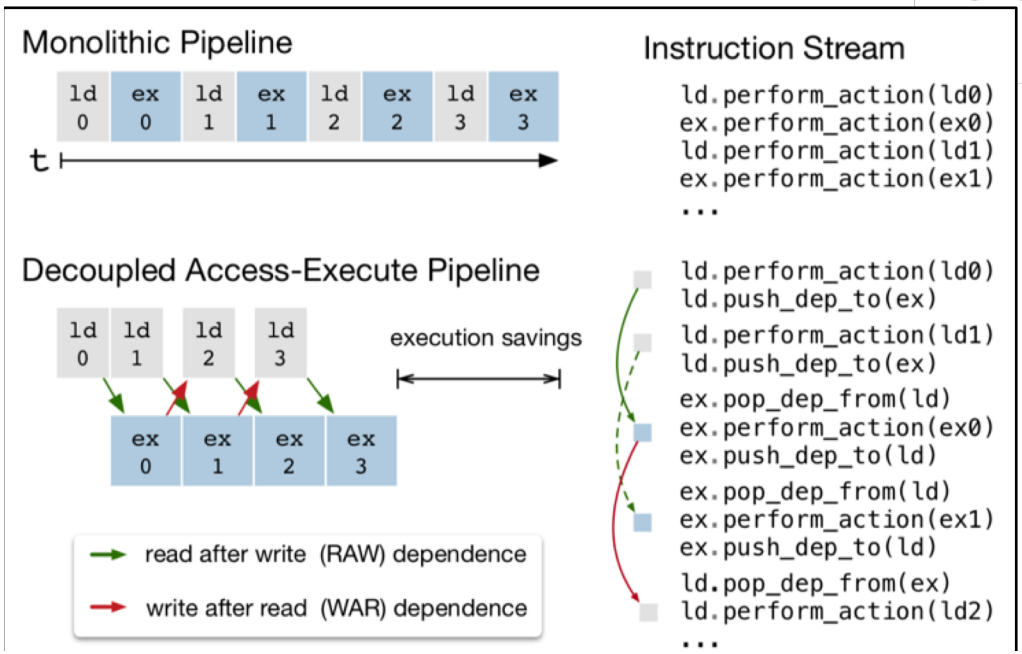


Thread lowering for decoupled access-execute architectures



Latency hiding





Roofline of an FPGA-based DL accelerator running ResNet inference, with latency hiding optimizations

	Intel Nervana Graph	XLA	TVM	Halide	Tensor RT
Framework independence	✓		✓	✓	✓
Framework connectors	✓	✓ (only 1)	~	~	✓
Hardware independence	✓	✓	✓	✓	
Leverage existing tensor libraries	✓	✓			~
Inference and training	✓	✓*	✓	✓	
Production ready				✓	✓

Halide

```
// Slice an affine matrix from the grid and
// transform the color
Expr gx = cast<float>(x)/sigma_s;
Expr gy = cast<float>(y)/sigma_s;
Expr gz =
  clamp(guide(x,y,n),0.f,1.f)*grid.channels();
Expr fx = cast<int>(gx);
Expr fy = cast<int>(gy);
Expr fz = cast<int>(gz);
Expr wx = gx-fx, wy = gy-fy, wz = gz-fz;
Expr tent =
  abs(rt.x-wx)*abs(rt.y-wy)*abs(rt.z-wz);
RDom rt(0,2,0,2,0,2);
Func affine;
affine(x,y,c,n) +=
  grid(fx+rt.x,fy+rt.y,fz+rt.z,c,n)*tent;
Func output;
Expr nci = input.channels();
RDom r(0, nci);
output(x,y,co,n) = affine(x,y,co*(nci+1)+nci,n);
output(x,y,co,n) +=
  affine(x,y,co*(nci+1)+r,n) * in(x,y,r,n);

// Propagate the gradients to inputs
auto d = propagate_adjoints(output, adjoints);
Func d_in = d(in);
Func d_guide = d(guide);
Func d_grid = d(grid);
```

Halide 24 lines

Runtime
64 ms (1 MPix)
165 ms (4 MPix)

```
xx = Variable(th.arange(0, w).cuda().view(1, -1).repeat(h, 1))
yy = Variable(th.arange(0, h).cuda().view(-1, 1).repeat(1, w))
gx = ((xx+0.5)/w) * gw
gy = ((yy+0.5)/h) * gh
gz = th.clamp(guide, 0.0, 1.0)*gd
fx = th.clamp(th.floor(gx - 0.5), min=0)
fy = th.clamp(th.floor(gy - 0.5), min=0)
fz = th.clamp(th.floor(gz - 0.5), min=0)
wx = gx - 0.5 - fx
wy = gy - 0.5 - fy
wx = wx.unsqueeze(0).unsqueeze(0)
wy = wy.unsqueeze(0).unsqueeze(0)
wz = th.abs(gz-0.5 - fz)
wz = wz.unsqueeze(1)
fx = fx.long().unsqueeze(0).unsqueeze(0)
fy = fy.long().unsqueeze(0).unsqueeze(0)
fz = fz.long()
cx = th.clamp(fx+1, max=gw-1);
cy = th.clamp(fy+1, max=gh-1);
cz = th.clamp(fz+1, max=gd-1)
fz = fz.view(bs, 1, h, w)
cz = cz.view(bs, 1, h, w)
batch_idx = th.arange(bs).view(bs, 1, 1, 1).long().cuda()
out = []
co = c // (ci+1)
for c_ in range(co):
  c_idx = th.arange((ci+1)*c_, (ci+1)*(c_+1)).view(\
            1, ci+1, 1, 1).long().cuda()
  a = grid[batch_idx, c_idx, fz, fy, fx]*(1-wx)*(1-wy)*(1-wz) + \
          grid[batch_idx, c_idx, cz, fy, fx]*(1-wx)*(1-wy)*(  wz) + \
          grid[batch_idx, c_idx, fz, cy, fx]*(1-wx)*(  wy)*(1-wz) + \
          grid[batch_idx, c_idx, cz, cy, fx]*(1-wx)*(  wy)*(  wz) + \
          grid[batch_idx, c_idx, fz, fy, cx]*(  wx)*(1-wy)*(1-wz) + \
          grid[batch_idx, c_idx, cz, fy, cx]*(  wx)*(1-wy)*(  wz) + \
          grid[batch_idx, c_idx, fz, cy, cx]*(  wx)*(  wy)*(1-wz) + \
          grid[batch_idx, c_idx, cz, cy, cx]*(  wx)*(  wy)*(  wz)
  o = th.sum(a[:, :-1, ...]*input, 1) + a[:, -1, ...]
  out.append(o.unsqueeze(1))
out = th.cat(out, 1)

out.backward(adjoints)
d_input = input.grad
d_grid = grid.grad
d_guide = guide.grad
```

PyTorch 42 lines

Runtime
1440 ms (1 MPix)
out of memory (4 MPix)

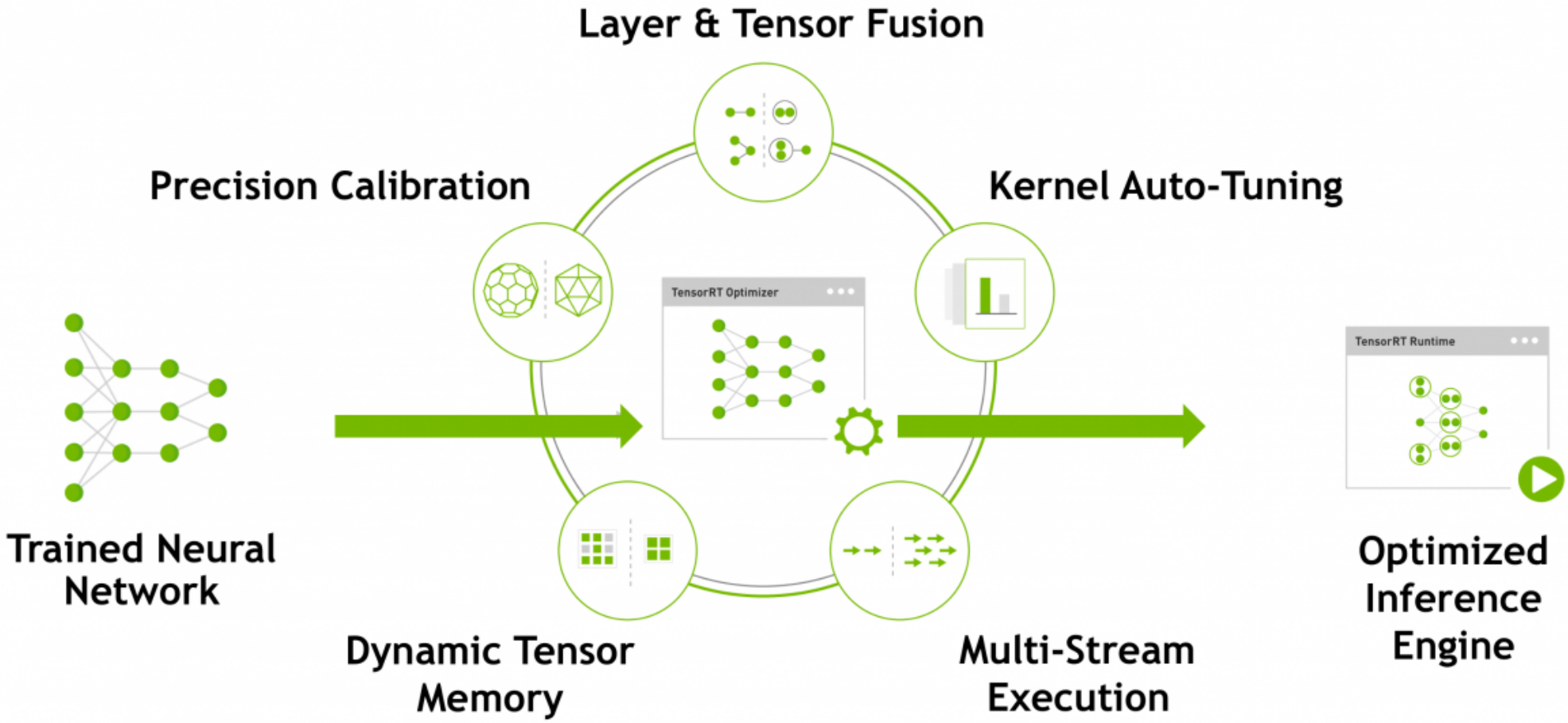
CUDA 308 lines

Runtime
430 ms (1 MPix)
2270 ms (4 MPix)

Pink = gradient computation

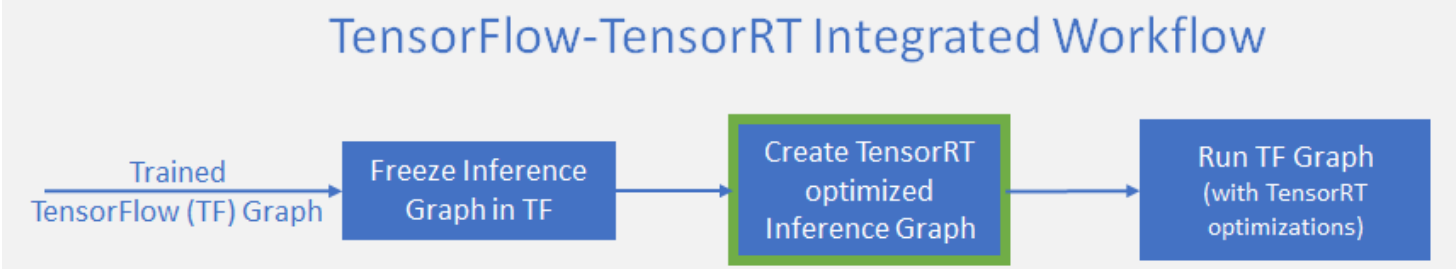
https://people.csail.mit.edu/tzumao/gradient_halide/

	Intel Nervana Graph	XLA	TVM	Halide	Tensor RT
Framework independence	✓		✓	✓	✓
Framework connectors	✓	✓ (only 1)	~	~	✓
Hardware independence	✓	✓	✓	✓	
Leverage existing tensor libraries	✓	✓			~
Inference and training	✓	✓*	✓	✓	
Production ready				✓	✓



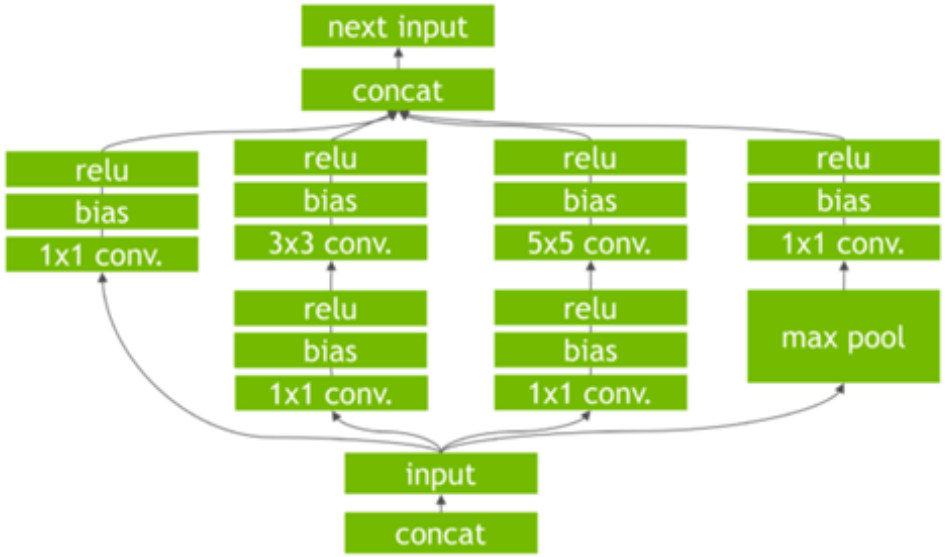
TensorRT optimizes trained neural network models to produce deployment-ready runtime inference engines

TensorRT optimizations

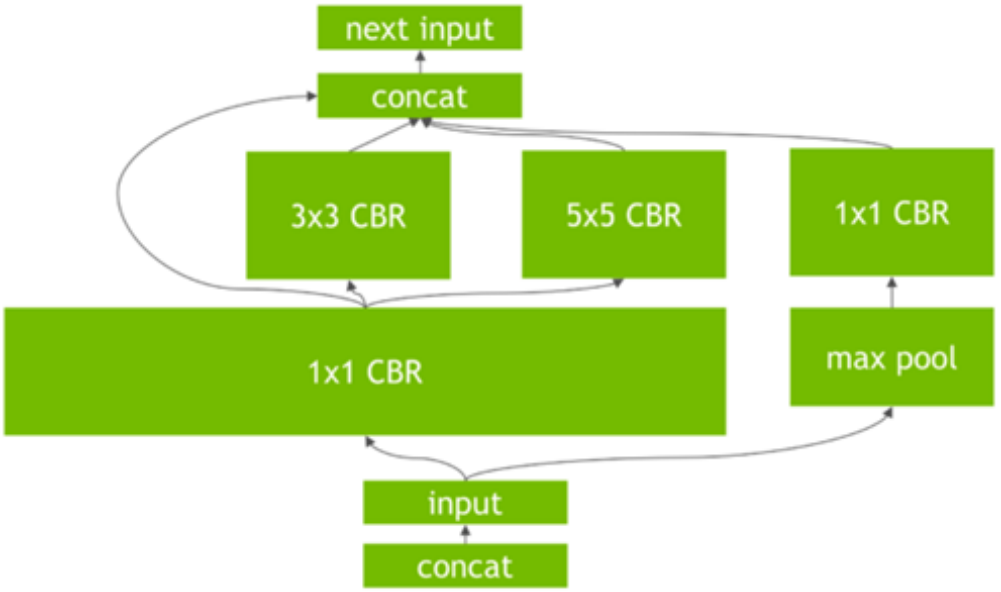


- Eliminate layers with unused output
- Fuse convolution, bias, and ReLU layers where possible
- Fuse horizontal layers that take the same source tensor and apply the same operations with similar parameters

Result: Speed up TensorFlow inference by 8x for low-latency runs of the ResNet-50 benchmark.



GoogLeNet Inception convolutional neural network with multiple convolutional and activation layers



TensorRT's vertical and horizontal layer fusion and layer elimination optimizations simplify module graph, reducing computation and memory overhead.

DLVM

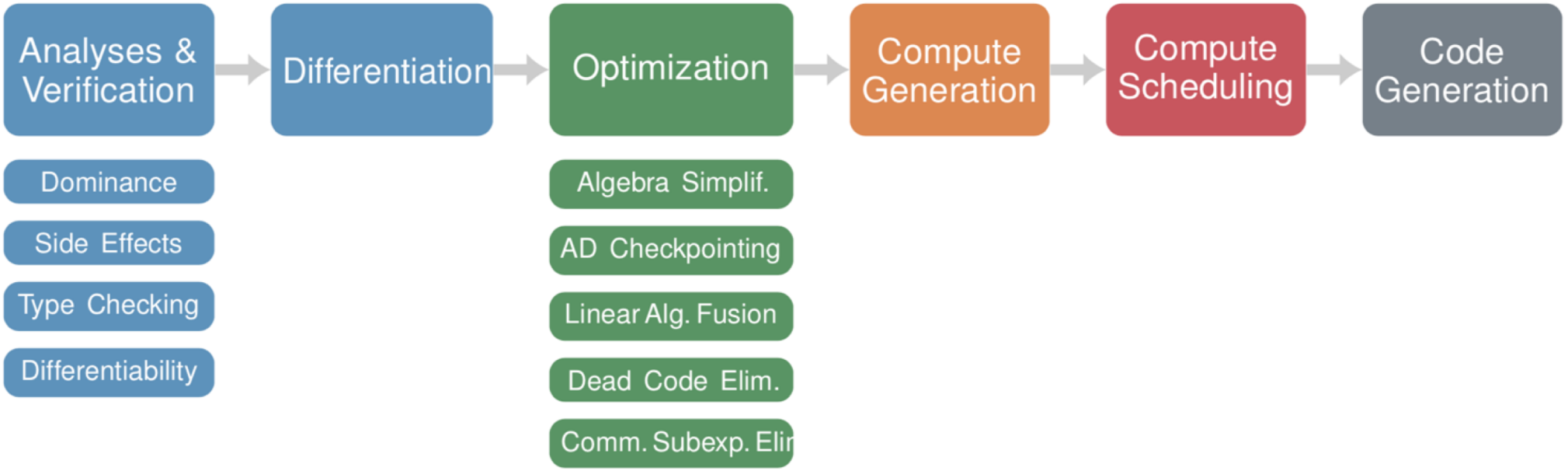


Figure 1: Compilation stages in the DLVM compilation pipeline.

https://openreview.net/forum?id=SJo1PLzCW

Domain-Specific Instructions

Kind	Example
Element-wise unary	`tanh %a: <10 x f32>`
Element-wise binary	`power %a: <10 x f32>, %b: 2: f32`
Dot	`dot %a: <10 x 20 x f32>, %b: <20 x 2 x f32>`
Concatenate	`concatenate %a: <10 x f32>, %b: <20 x f32> along 0`
Reduce	`reduce %a: <10 x 30 x f32> by add along 1`
Transpose	`transpose %m: <2 x 3 x 4 x 5 x i32>`
Convolution	`convolve %a: <…> kernel %b: <…> stride %c: <…> …`
Slice	`slice %a: <10 x 20 x i32> from 1 upto 5`
Random	`random 768 x 10 from 0.0: f32 upto 1.0: f32`
Select	`select %x: <10 x f64>, %y: <10 x f64> by %flags: <10 x bool>`
Compare	`greaterThan %a: <10 x 20 x bool>, %b: <1 x 20 x bool>`
Data type cast	`dataTypeCast %x: <10 x i32> to f64`

General-Purpose Instructions

Kind	Example
Function application	`apply %foo(%x: f32, %y: f32): (f32, f32) -> <10 x 10 x f32>`
Branch	`branch 'block_name(%a: i32, %b: i32)`
Conditional (if-then-else)	`conditional %cond: bool then 'then_block() else 'else_block()`
Shape cast	`shapeCast %a: <1 x 40 x f32> to 2 x 20`
Extract	`extract #x from %pt: $Point`
Insert	`insert 10: f32 to %pt: $Point at #x`
Allocate stack	`allocateStack $Point count 1`
Allocate heap	`allocateHeap $MNIST count 1`
Deallocate	`deallocate %x: *<10 x f32>`
Load	`load %ptr: *<10 x i32>`
Store	`store %x: <10 x i32> to %ptr: *<10 x i32>`
Copy	`copy from %src: *<10 x f16> to %dst: *<10 x f16> count 1: i64`

```
func @inference: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}
```

$$O = wX + b$$

```
func @inference: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}

[gradient @inference wrt 1, 2]
func @inference_grad: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>)
                    -> (<784 x 10 x f32>, <1 x 10 x f32>)
```

$$O = wX + b$$

Differentiation Pass

Canonicalizes every gradient function declaration in an IR module

```
func @inference: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}


func @inference_grad: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>)
                    -> (<784 x 10 x f32>, <1 x 10 x f32>) {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>



}
```

$$O = wX + b$$

Copy instructions from original function

```
func @inference: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}


func @inference_grad: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>)
                    -> (<784 x 10 x f32>, <1 x 10 x f32>) {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    %0.2 = transpose %x: <1 x 784 x f32>
    %0.3 = multiply %0.2: <1 x 784 x f32>, 1: f32
    return (%0.3: <1 x 10 x f32>, 1: f32): (<1 x 10 x f32>, f32)
}
```

$$O = wX + b$$

Generate adjoint code

```
func @inference: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}
```

$$O = wX + b$$

```
func @inference_grad: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>)
                    -> (<784 x 10 x f32>, <1 x 10 x f32>) {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    %0.2 = transpose %x: <1 x 784 x f32>
    %0.3 = multiply %0.2: <1 x 784 x f32>, 1: f32
    return (%0.3: <1 x 10 x f32>, 1: f32): (<1 x 10 x f32>, f32)
}
```

Algebra Simplification Pass

```
func @inference: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}
```

$$O = wX + b$$

```
func @inference_grad: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>)
                    -> (<784 x 10 x f32>, <1 x 10 x f32>) {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    %0.2 = transpose %x: <1 x 784 x f32>
    return (%0.2: <1 x 10 x f32>, 1: f32): (<1 x 10 x f32>, f32)
}
```

Dead Code Elimination Pass

```
func @inference: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}


func @inference_grad: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>)
                    -> (<784 x 10 x f32>, <1 x 10 x f32>) {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = transpose %x: <1 x 784 x f32>
    return (%0.0: <1 x 10 x f32>, 1: f32): (<1 x 10 x f32>, f32)
}
```

$$O = wX + b$$

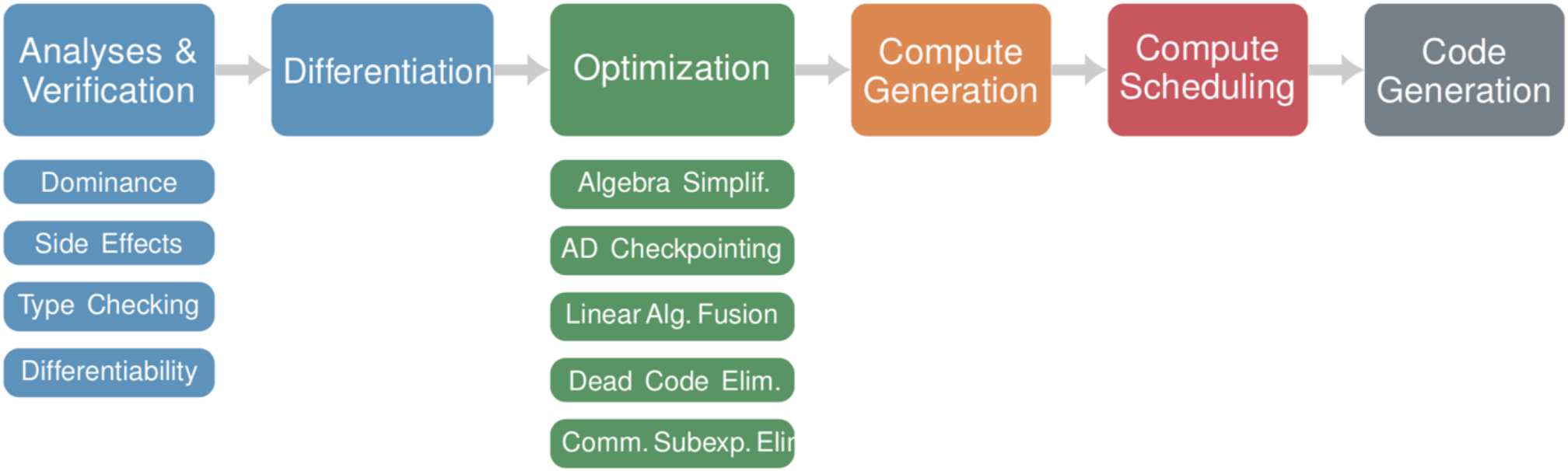


Figure 1: Compilation stages in the DLVM compilation pipeline.

Update: The authors of this project are no longer maintaining DLVM, but instead developing Swift for TensorFlow, a project providing first-class language and compiler support for machine learning in Swift.

https://openreview.net/forum?id=SJo1PLzCW

Summary

- Limitations of “DNNs are graphs” model and subsequent “translate graphs to kernel operations” approach
- Increasing interest in “DNNs are programs” perspective and use of mainstream compiler technology
- Still many areas of uncertainty and exploration, e.g.:
  - Nature of intermediate representation
  - Training vs. inference
  - Different network architectures
  - Target platforms