# A Primer to ML Compilers

Aditya Ramesh

**Overview**
Compilers
Papers
Future

# ML Compilers

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class ToyMLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(8, 16)
        self.fc2 = nn.Linear(16, 4)
        self.fc3 = nn.Linear(4, 1)

    def forward(self, x):
        x = self.fc1(x)        # affine transform
        x = F.relu(x)          # nonlinearity
        x = self.fc2(x)
        x = F.relu(x)
        x = self.fc3(x)        # prediction
        return x

model = ToyMLP()
```

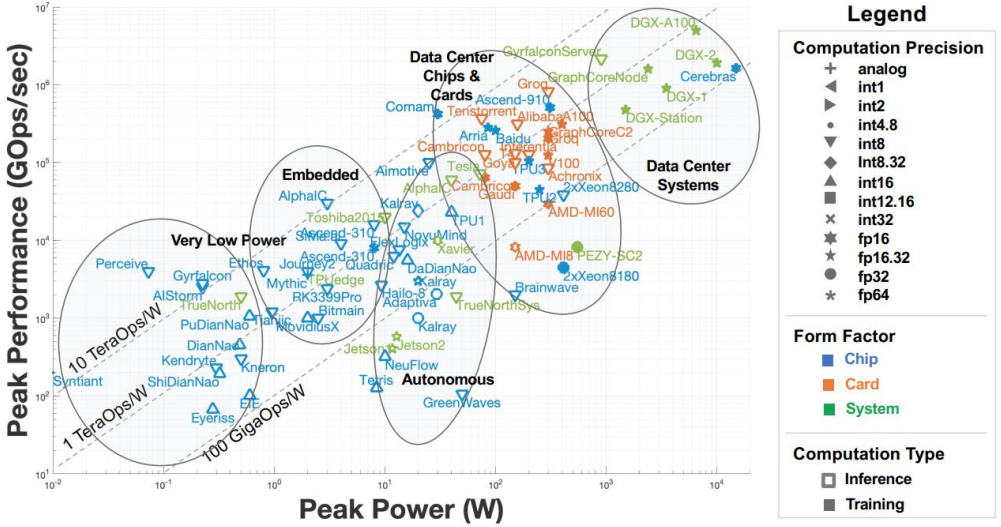
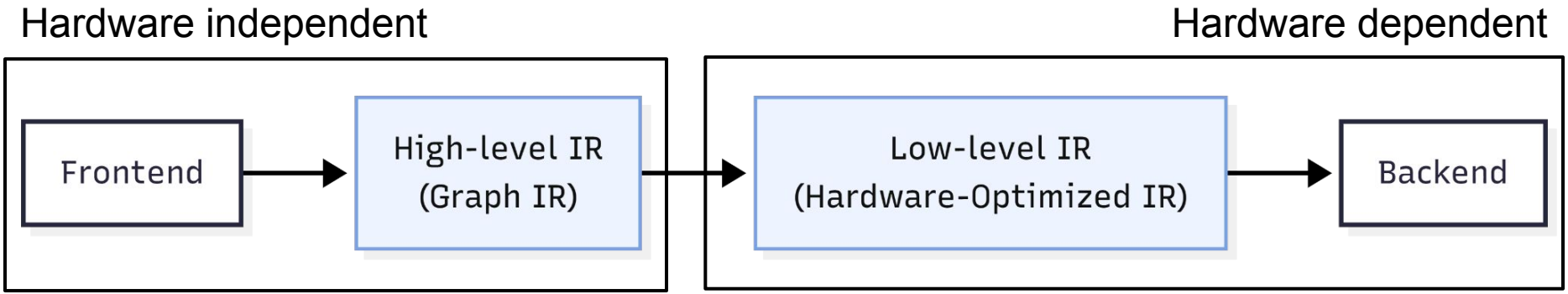



Fig. 2. Peak performance vs. power scatter plot of publicly announced AI accelerators and processors.

https://knowen-production.s3.amazonaws.com/uploads/attachment/file/5151/MLAccelerators_HPEC_2020_Reuther_CameraReady.pdf

# Common Architectures




https://arxiv.org/pdf/2002.03794

# High Level IR - Graph IR Representation

## DAG-based IRs

Nodes denote atomic DL operators
Edges denote tensors and dependencies

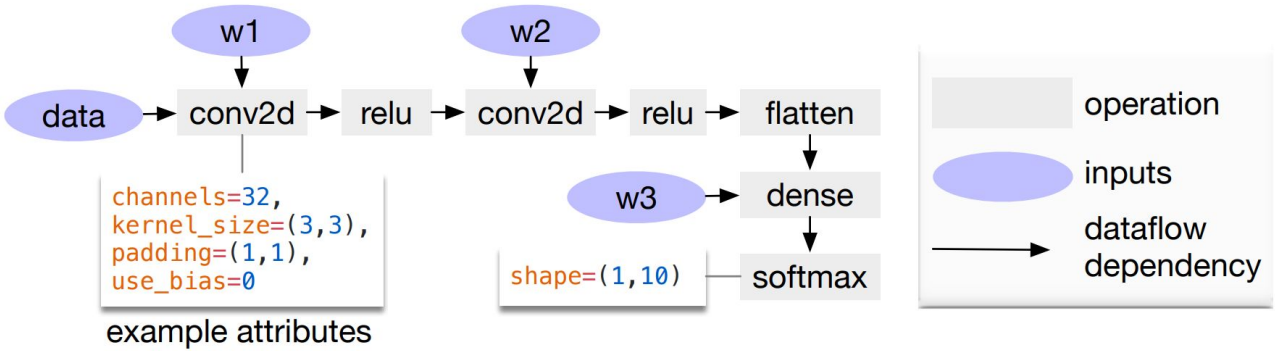



Figure from TVM: An Automated End-to-End Optimizing Compiler for Deep Learning

https://arxiv.org/pdf/2002.03794

# High Level IR - Graph IR Representation

**Let-binding based IRs**

- Defining an expression with *let* keyword
- Control flow easier
- *Relay IR* from TVM incorporates both Let-binding based IR and DAG-based IR

```
let x1 = op1(input)
let x2 = op2(x1)
let x3 = op3(x1, x2)
return x3
```

https://arxiv.org/pdf/2002.03794

# High Level IR - Implementation

**Tensors - Multi-dimensional Arrays**

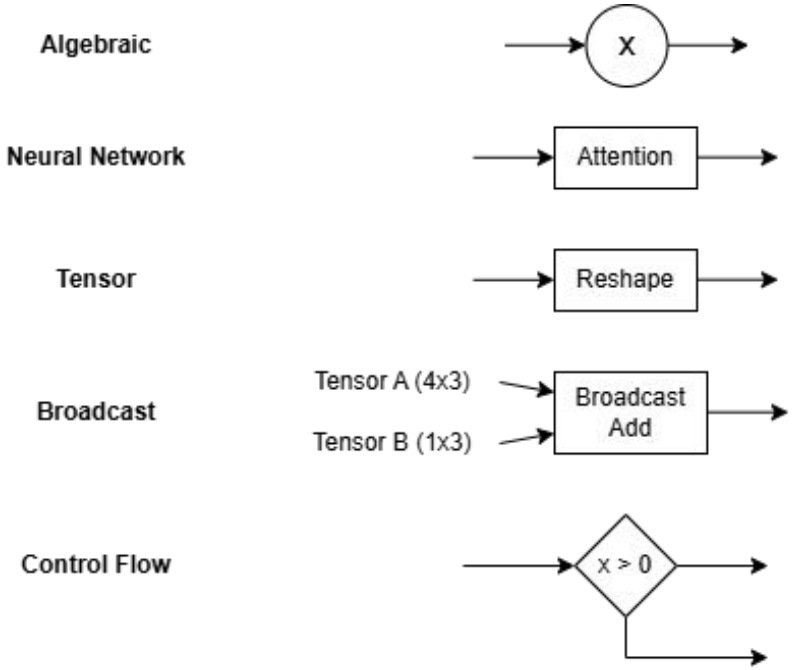
Represented by either ***memory pointers*** or ***placeholders***

**Placeholders** - variables with shape information that will be populated with data later

- Support unknown/dynamic shapes
- Aware of different data layouts
- Memory boundaries

https://arxiv.org/pdf/2002.03794

# High Level IR - Implementation




https://arxiv.org/pdf/2002.03794

# High Level IR - Optimizations

**DAG optimizations**

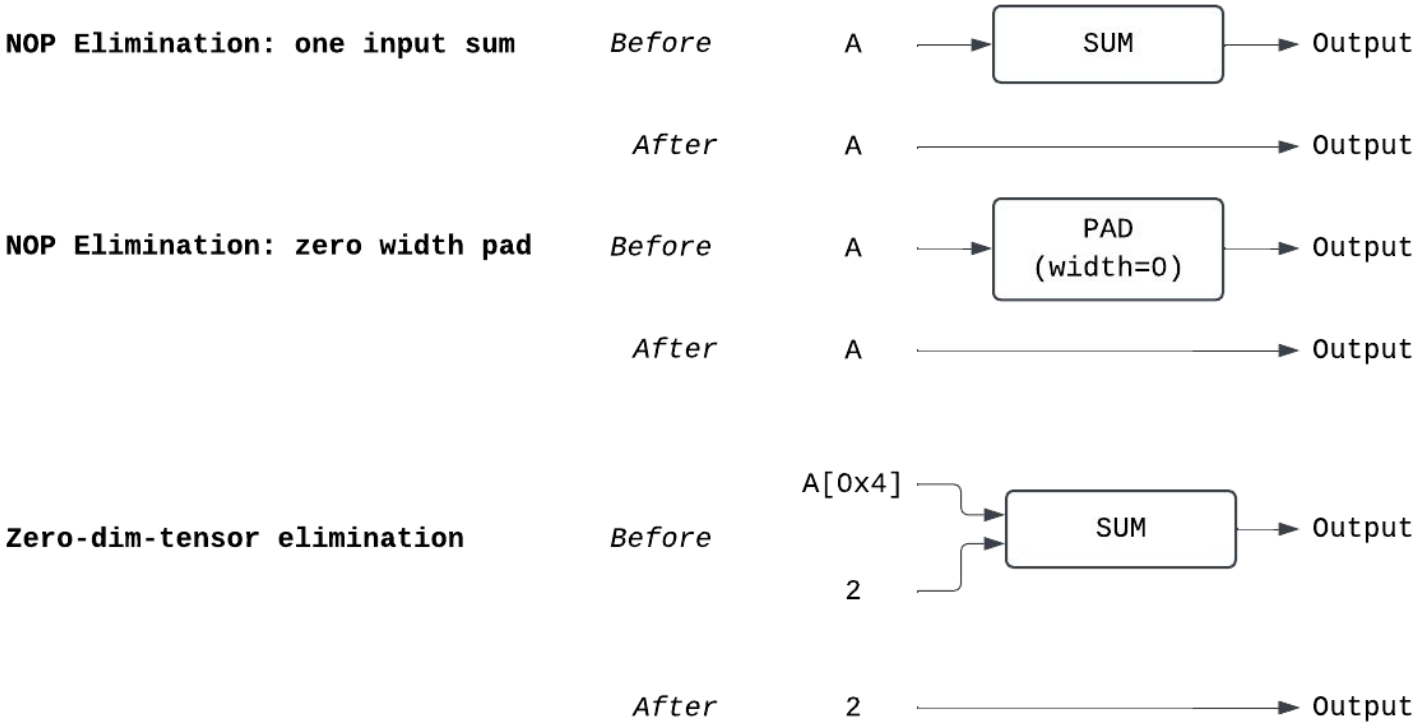
**Defined in *passes***

**Three levels**

- **Node level**
- **Block level**
- **Dataflow level**

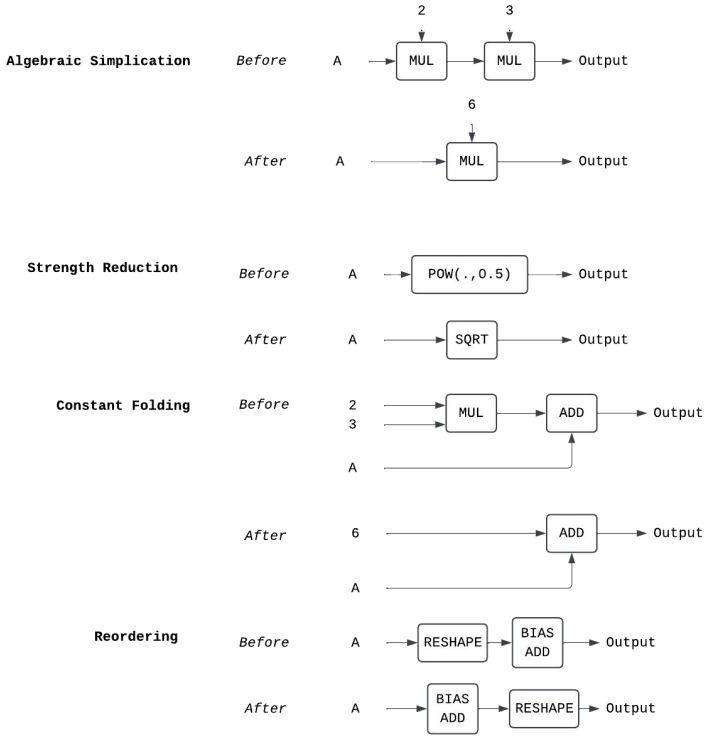
https://arxiv.org/pdf/2002.03794

# High Level IR - Node Level Optimizations




https://arxiv.org/pdf/2002.03794

# High Level IR - Block Level Optimizations




https://arxiv.org/pdf/2002.03794

# High Level IR - Block Level Optimizations

## Operator Fusion (TVM implementation)

### Classes

- injective (one-to-one map, e.g., add)
- reduction (e.g., sum)
- complexout-fusable (can fuse element-wise map to output, e.g., conv2d)
- opaque (cannot be fused, e.g., sort)

### Rules

- injective operators can be fused into another injective operator
- reduction operator can be fused with input injective operators
- complex-out-fusable can fuse injective operators to its output

## Operator sinking

https://arxiv.org/pdf/2002.03794

# High Level IR - Dataflow Optimizations

**Common sub-expression elimination (CSE)**

An expression *E* is a *common sub-expression* if the value of *E* is previously computed, and the value of *E* has not to be changed since previous computation

**Dead code elimination (DCE)**

A set of code is dead if its computed results or side-effects are not used.

**Static memory planning**

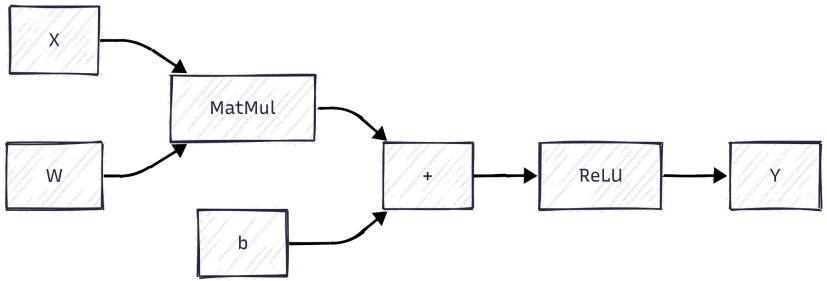
In-place vs. standard memory sharing

**Layout transformation**

Performance often dependent on tensor layout. Best tensor layout depends on target device.

https://arxiv.org/pdf/2002.03794

# Low Level IR - Implementation




```
for i in range(M):
    for j in range(N):
        acc = 0
        for k in range(K):
            acc += X[i, k] * W[k, j]
        acc = acc + b[j]
        if acc < 0:
            acc = 0
        Y[i, j] = acc
```

High Level IR

(Pseudo) Low Level IR

**High Level IR is *lowered* to Low Level IR**

https://arxiv.org/pdf/2002.03794

# Low Level IR - Implementation

```
m, n, h = t.var('m'), t.var('n'), t.var('h')
A = t.placeholder((m, h), name='A')
B = t.placeholder((n, h), name='B')
k = t.reduce_axis((0, h), name='k')
C = t.compute((m, n), lambda y, x:
    t.sum(A[k, y] * B[k, x], axis=k))
```

computing rule

result shape

TVM IR

```
x.permute(1,0) + x[2, :]
```

becomes

```
def inner_fn(index: List[sympy.Expr]):
    i1, i0 = index
    tmp0 = ops.load("x", i1 + i0*size1)
    tmp1 = ops.load("x", 2*size1 + i0)
    tmp2 = ops.add(tmp0, tmp1)
    return tmp2
```

Torch Inductor IR

```
declare {
 %input = weight float<8 x 28 x 28 x 1>,
     broadcast, 0.0
 %filter = weight float<16 x 5 x 5 x 1>,
     xavier, 25.0
 %filter0 = weight float<16>, broadcast,
     0.100
 %weights = weight float<10 x 144>, xavier,
     144.0
 %bias = weight float<10>, broadcast, 0.100
 %selected = weight index<8 x 1>
 ...
 %result = weight float<8 x 10>
}

program {
 %allo = alloc float<8 x 28 x 28 x 16>
 %conv = convolution [5 1 2 16] @out %allo,
     @in %input, @in %filter3, @in %bias0
 %allo0 = alloc float<8 x 28 x 28 x 16>
 %relu = max0 @out %allo0, @in %allo
 %allo1 = alloc index<8 x 9 x 9 x 16 x 2>
 %allo2 = alloc float<8 x 9 x 9 x 16>
 %pool = pool max [3 3 0] @out %allo2, @in
     %allo0, @inout %allo1
 ...
 %deal6 = dealloc @out %allo6
 %deal7 = dealloc @out %allo7
 %deal8 = dealloc @out %allo8
 %deal9 = dealloc @out %allo9
}
```

**Figure 3:** Unoptimized low-level Glow IR.

Glow IR

## Real examples of Low Level IR

https://arxiv.org/pdf/2002.03794

# Low Level IR - Implementation

**High Level IR lowered into Low Level IR**

**Optimizations and Autotuning**

**Low Level IR lowered to device code**

https://arxiv.org/pdf/2002.03794

# Low Level IR - Optimizations

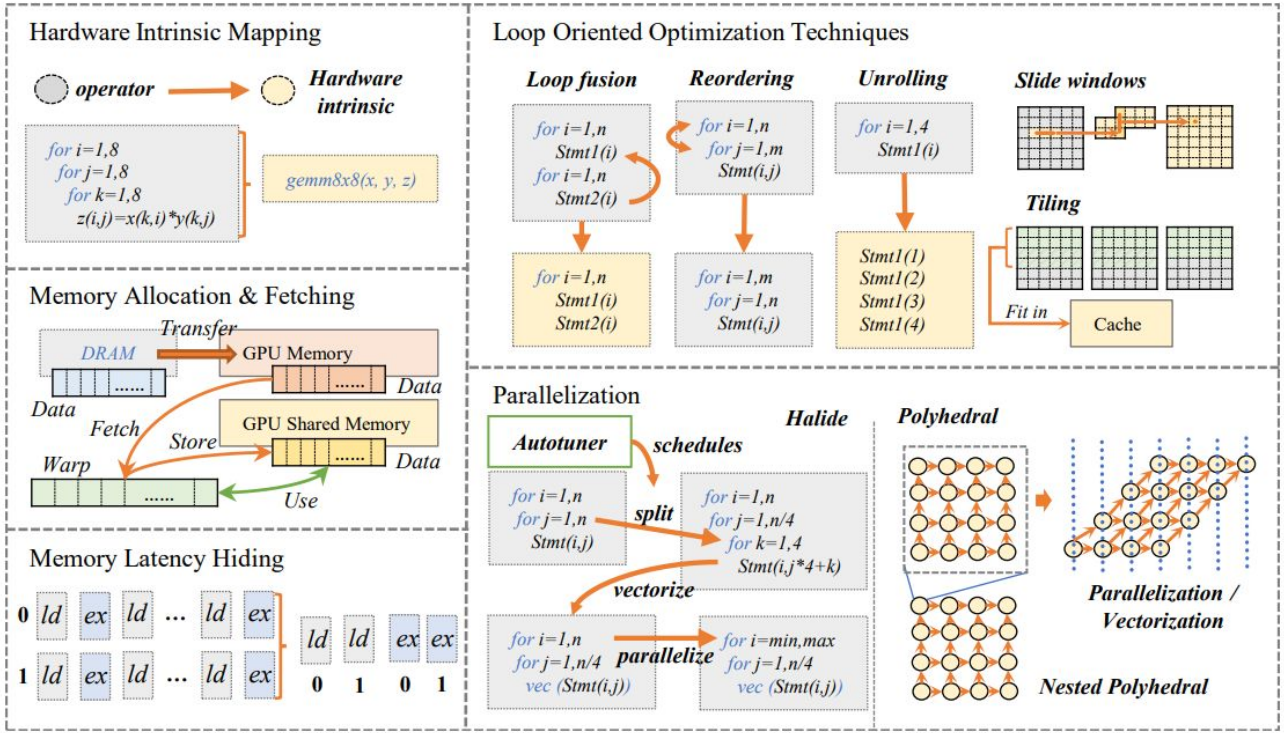



Fig. 4. Overview of hardware-specific optimizations applied in DL compilers.

https://arxiv.org/pdf/2002.03794

# Low Level IR - Autotuning

**Define input/outputs**
- Input: Target Device, register count, shared memory size
- Output: Tile size, grid size, loop ordering

**Cost model**
- Black box
- ML model
- Heuristics

**Search Method**
- Genetic Algorithm
- Simulated annealing algorithm
- Reinforcement learning

**Acceleration**
- Parallelization
- Configuration reuse

https://arxiv.org/pdf/2002.03794

# Low Level IR - Lower to device

**Kernel Libraries**

Intel - DNNL

NVIDIA - cuDNN

AMD - MIOpen

**Code Generation**

Triton, CUDA, CUTLASS, …

**LLVM IR**

https://arxiv.org/pdf/2002.03794

Overview
**Compilers**
Papers
Future


- Theano: A CPU and GPU Math Compiler in Python
- TensorFlow: A system for large-scale machine learning

# Theano (2010)

```python
import theano
from theano import tensor

# Declare two symbolic 2D arrays (matrices)
A = tensor.dmatrix("A")
B = tensor.dmatrix("B")

# Define a matrix multiplication (dot product) operation
C = tensor.dot(A, B)

# Create a function that computes the result of the matrix multiplication
f = theano.function([A, B], C)

# Sample matrices
A_val = [[1, 2], [3, 4]]
B_val = [[5, 6], [7, 8]]

# Evaluate the matrix multiplication
result = f(A_val, B_val)
print(result)
```

From Wikipedia

# Theano - Compilation

**Canonicalization** - *Simplify expression*

**Stabilization** - *Numerical stability*

**Specialization** - *Replace expressions*

**GPU Transfer** - *Replace CPU with GPU implementation*

**Code Generation** - *Produces and loads modules for the expression in the computation graph*

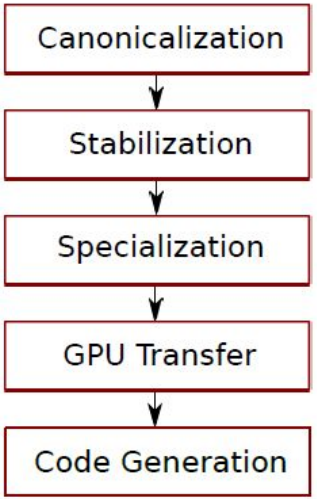



*Figure 8: The compilation pipeline for functions compiled for GPU. Functions compiled for the CPU omit the GPU transfer step. v2*

# Theano - Benchmark

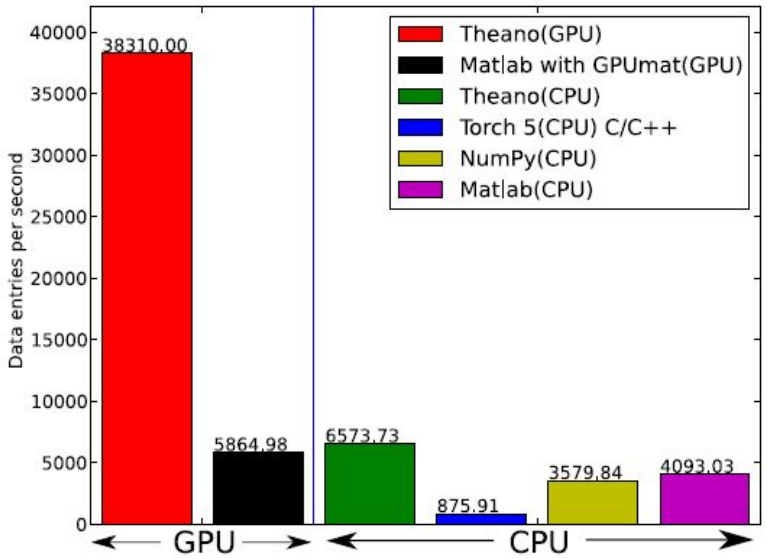



***Figure 5:*** *Fitting a multi-layer perceptron to simulated data with various implementations of stochastic gradient descent. These models have 784 inputs, 500 hidden units, a 10-way classification, and are trained 60 examples at a time.*

# TensorFlow

Represents ML program as a DAG

- Nodes = operations
- Edges = tensors
- Stateful Nodes = variables, queues, checkpoint ops, etc.

Makes the communication explicit, enables distributed and heterogeneous execution

# TensorFlow - Graph

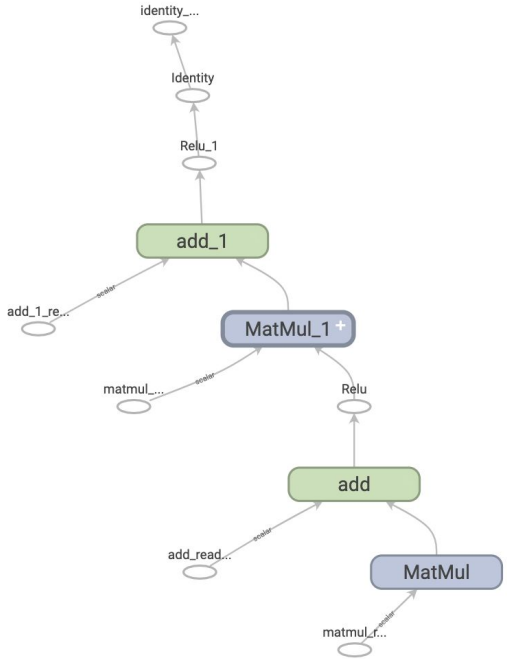
**Operators** and **Tensors** as nodes and edges

Graph level optimizations

Operations have multiple kernel implementation for different devices

In a heterogeneous distributed environment, we can place different operators on different devices (CPU, GPU, TPU)

Enabled by stateful variables and queues



# TensorFlow - Placement

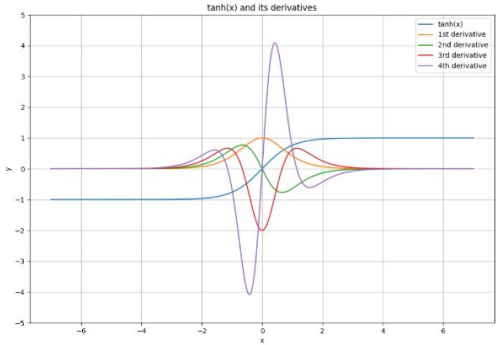
Device Placement Optimization with Reinforcement Learning

A Hierarchical Model for Device Placement

Placeto: Learning Generalizable Device Placement Algorithms for Distributed Machine Learning


- AutoGrad (Ch. 4)
- Chainer: A Deep Learning Framework for Accelerating the Research Cycle
- PyTorch: An Imperative Style, High-Performance Deep Learning Library
- Glow: Graph Lowering Compiler Techniques for Neural Networks
- PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation

# AutoGrad (2015)




```
> from autograd import elementwise_grad as egrad  # for functions that vectorize over inputs
> import matplotlib.pyplot as plt
> x = np.linspace(-7, 7, 700)
> plt.plot(x, tanh(x),
.          x, egrad(tanh)(x),                                  # first  derivative
.          x, egrad(egrad(tanh))(x),                           # second derivative
.          x, egrad(egrad(egrad(tanh)))(x),                    # third  derivative
.          x, egrad(egrad(egrad(egrad(tanh))))(x),)            # fourth derivative
> plt.show()
```

Designed to make gradients effortless

# AutoGrad - Computation Graph

Introduced **Reverse-mode differentiation**

Transform computation graph *F* to gradient function *F'*

Constructing Computation Graph F

1) Direct specification of graph (Theano)
2) Source code inspection (interpreter)
3) Monitoring function execution (AutoGrad)

# AutoGrad - Computation Graph

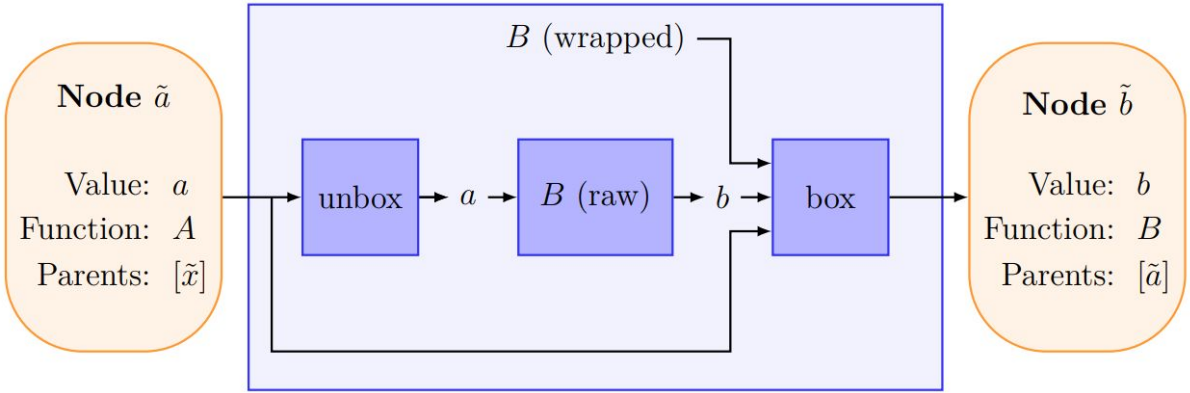



**Figure 4.1:** Primitive function wrapping. When called with a `Node` as an argument, the wrapped version of the primitive function extracts the `Node`'s value and passes it to the underlying primitive function. It boxes the result in a new `Node`, along with pointers to the original argument and the function itself.

Given a node in the tape, we can recursively trace through the graph!

# AutoGrad - Applying Backward Pass

Backward pass - applying reverse accumulation mode differentiation procedure

Create a separate tape for each variable we want the gradient with respect to, each node keeps track of the tape it belongs to, each node inherits tapes from their parents

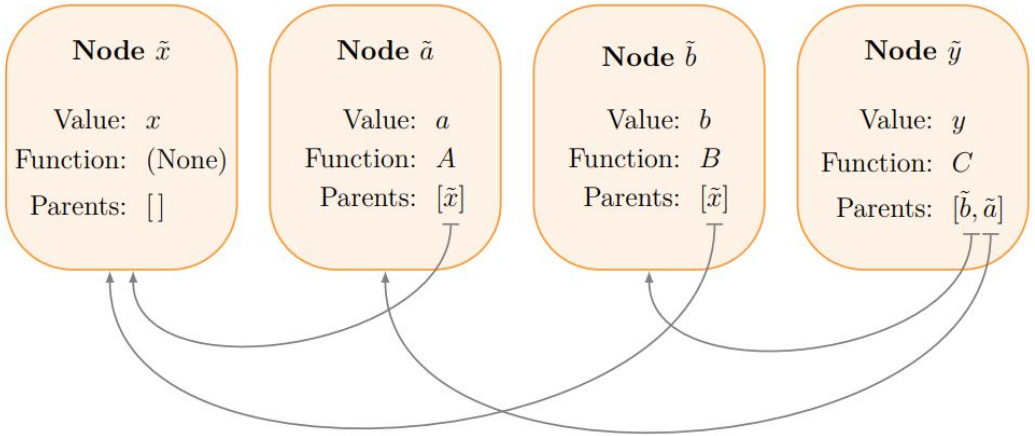



**Figure 4.2:** Representation of computation graph as set of linked nodes. Each node records the function that produces it, and the arguments given to that function, the node's parents. This allows the entire ancestry of any node to be determined.

# AutoGrad - Computation Graph

| | |
|---|---|
| Operators | +, −, *, /, (−), **, %, <, <=, ==, !=, >=, > |
| Basic math functions | exp, log, square, sqrt, sin, cos, tan, sinh, cosh, tanh, sinc, abs, fabs, logaddexp, logaddexp2, absolute, reciprocal, exp2, expm1, log2, log10, log1p, arcsin, arccos, arctan, arcsinh, arccosh, arctanh, rad2deg, degrees, deg2rad, radians |
| Complex numbers | real, imag, conj, angle, fft, fftshift, ifftshift, real_if_close |
| Array reductions | sum, mean, prod, var, std, max, min, amax, amin |
| Array reshaping | reshape, ravel, squeeze, diag, roll, array_split, split, vsplit, hsplit, dsplit, expand_dims, flipud, fliplr, rot90, swapaxes, rollaxis, transpose, atleast_1d, atleast_2d, atleast_3d |
| Linear algebra | dot, tensordot, einsum, cross, trace, outer, det, slogdet, inv, norm, eigh, cholesky, sqrtm, solve_triangular |
| Other array operations | cumsum, clip, maximum, minimum, sort, msort, partition, concatenate, diagonal, truncate_pad, tile, full, triu, tril, where, diff, nan_to_num, vstack, hstack |
| Probability functions | t.pdf, t.cdf, t.logpdf, t.logcdf, multivariate_normal.logpdf, multivariate_normal.pdf, multivariate_normal.entropy, norm.pdf, norm.cdf, norm.logpdf, norm.logcdf, dirichlet.logpdf, dirichlet.pdf |
| Special functions | polygamma, psi, digamma, gamma, gammaln, rgamma, multigammaln, j0, y0, j1, y1, jn, yn, erf, erfc |

**Table 4.1:** Primitives with gradients implemented in Autograd.

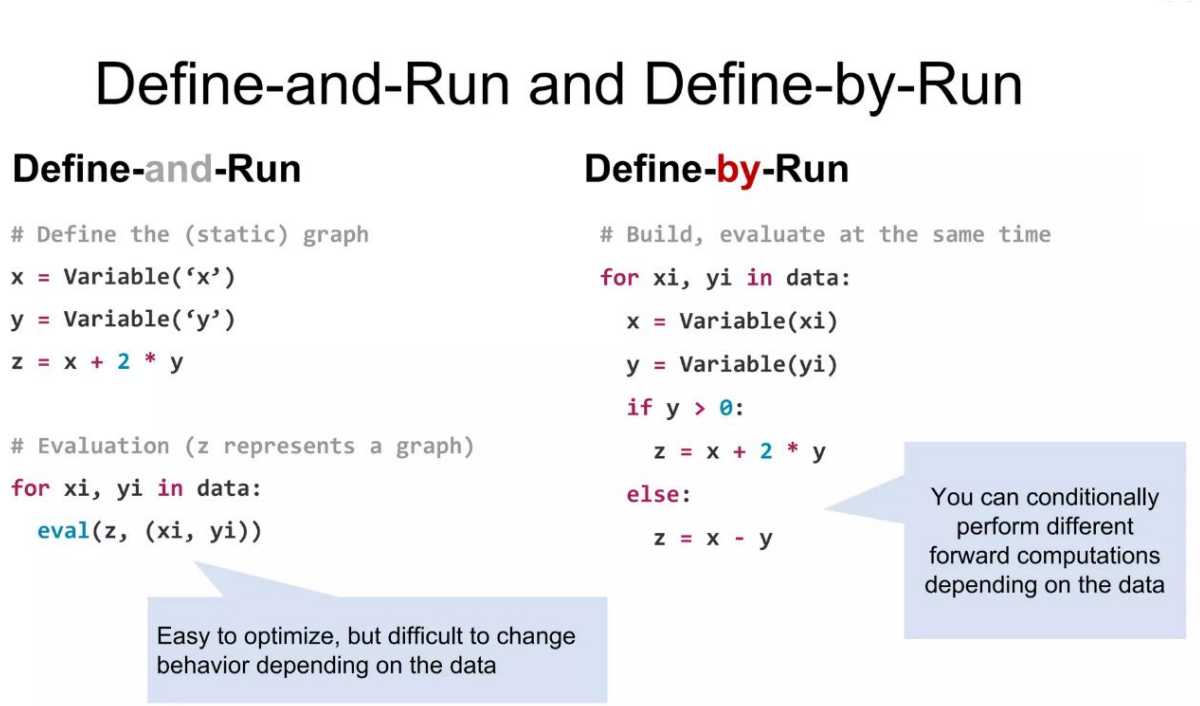
Requires supporting gradients of many primitives!

# Chainer (2019)

**Problem:** Many deep learning frameworks use DSLs, are hard to debug, can’t use standard programming debuggers.

**Solution:** Chainer builds the computation graph through the execution trace!

# Chainer (2019)




Source

# Chainer - Object Oriented Models

Compositionality is important!

In Define-by-Run paradigm, model must be bound to parameters

Can’t just use functions anymore, resolved with object-oriented programming

First introduced by Chainer in 2015

```
class Linear(Link):
    def __init__(self, n_in, n_out):
        with self.init_scope():
            self.W = Parameter(HeNormal(),
                               (n_out, n_in))
            self.b = Parameter(0, (n_out,))

    def forward(self, x):
        return x @ self.W.T + self.b

class MultiLayerPerceptron(Chain):
    def __init__(self, n_in, n_hid, n_out):
        with self.init_scope():
            self.l1 = Linear(n_in, n_hid)
            self.l2 = Linear(n_hid, n_out)

    def forward(self, x):
        h = relu(self.l1(x))
        return self.l2(h)
```

**Figure 2: Examples of model definitions by object-oriented programming.**

# Chainer - Backpropagation

Memory-Efficient Backpropagation

- Global Memory Usage Reduction
  - Reference-counting garbage collection
  - Subgraph is released immediately once rendered unreachable
- Local Memory Usage Reduction
  - Some nodes (ex: tanh) do not require input to compute gradient
    - Explicitly set with retain_inputs and retain_outputs
    - Creates the need to separate Variable from VariableNode, otherwise data will always be referenced and can not be freed by the graph!

# Chainer - CuPy

```
import numpy as np
x = np.array([1, 2])
l2 = np.linalg.norm(x)
```

(a) NumPy

```
import cupy as cp
x = cp.array([1, 2])
l2 = cp.linalg.norm(x)
```

(b) CuPy

**Figure 3: Examples using NumPy and CuPy.**

CuPy was developed as the backend for Chainer

# PyTorch (2019)

**Problem:** Dynamic eager execution is useful but leads to low performance

**Solution: PyTorch replaces some core ideas from Chainer**

1) C++ core
2) Separate control and data flow
3) Custom caching tensor allocator
4) Multiprocessing

# PyTorch - C++ Core

*Libtorch* library implements tensor data structure, GPU and CPU operators, automatic differentiation system, gradient formulas implemented in C++

Control flow is handled in Python

Computation is handled in C++ for CPU.

For GPUs, CUDA kernels are invoked and queued on the GPUs hardware FIFO using CUDA stream

# PyTorch - Custom caching tensor allocator

Operators dynamically allocate an output tensor

But using cudaFree blocks the caller until all queued GPU work is complete!

PyTorch created a custom allocator

- Incrementally build a cache of CUDA memory
- Reassign it later without using CUDA APIs
- One-pool-per-stream design assumption

# Glow (2019)

```
for (...) A[i] = 3;
for (...) A[i] = 4;
return A[0];
```

**Figure 1:** Compilers struggle to analyze and optimize this code when the two loops come from two different nodes in the dataflow graph.

**Glow** is an abbreviation of **Graph Lowering**

General compilers do not optimize ML model code well

# Glow - High-level IR

Strongly typed dataflow graph

Graph-level optimization
- Dead Code Elimination
- Constant Propagation and Folding
- Graph rewriting

High-level operator is then lowered into low-level linear algebra nodes

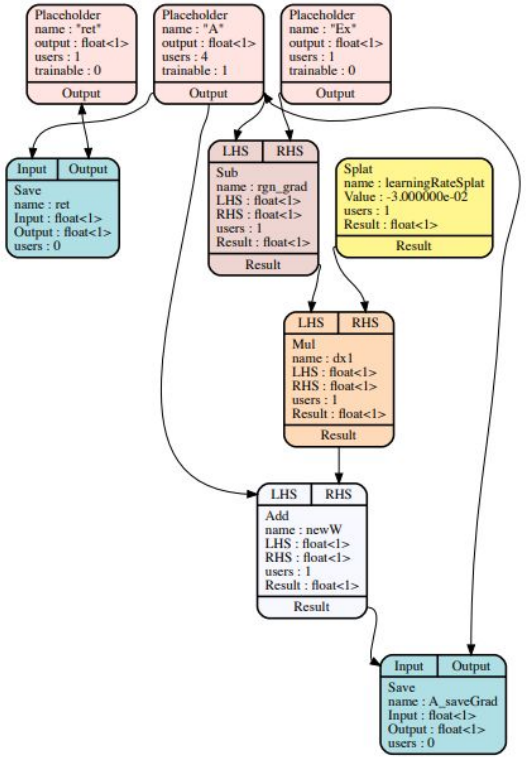



**Figure 2:** A lowered compute graph in Glow's high-level IR, representing a regression of $A$, automatically differentiated by Glow.

# Glow - Low-level IR

IRGen - IR Generation

Two memory regions: “declare” (global) and “program” (local)

Instruction operand qualified with

- @in
- @out
- @inout

Optimiztions like copy elimination or buffer sharing

```
declare {
 %input = weight float<8 x 28 x 28 x 1>,
     broadcast, 0.0
 %filter = weight float<16 x 5 x 5 x 1>,
     xavier, 25.0
 %filter0 = weight float<16>, broadcast,
     0.100
 %weights = weight float<10 x 144>, xavier,
     144.0
 %bias = weight float<10>, broadcast, 0.100
 %selected = weight index<8 x 1>
 ...
 %result = weight float<8 x 10>
}

program {
 %allo = alloc float<8 x 28 x 28 x 16>
 %conv = convolution [5 1 2 16] @out %allo,
     @in %input, @in %filter3, @in %bias0
 %allo0 = alloc float<8 x 28 x 28 x 16>
 %relu = max0 @out %allo0, @in %allo
 %allo1 = alloc index<8 x 9 x 9 x 16 x 2>
 %allo2 = alloc float<8 x 9 x 9 x 16>
 %pool = pool max [3 3 0] @out %allo2, @in
     %allo0, @inout %allo1
 ...
 %deal6 = dealloc @out %allo6
 %deal7 = dealloc @out %allo7
 %deal8 = dealloc @out %allo8
 %deal9 = dealloc @out %allo9
}
```

**Figure 3:** Unoptimized low-level Glow IR.

# Glow - Lowering and Instruction Lifetime

```
BB.newInstr("AvgPool")
  .addOperand("Dest", OperandKind::Out)
  .addOperand("Src", OperandKind::In)
  .addMember(MemberType::VectorUnsigned, "Kernels")
  .addMember(MemberType::VectorUnsigned, "Strides")
  .addMember(MemberType::VectorUnsigned, "Pads")
  .autoIRGen()
  .autoVerify(VerifyKind::SameElementType,
              {"Dest", "Src"})
  .addGradientInstr({"Dest"}, {"Dest", "Src"});
```

**Figure 4:** Example class-gen for the Average Pool instruction.

1. The graph is either loaded via the graph loader (from ONNX or Caffe2 format), or constructed via the C++ interface.
2. The graph is differentiated if needed.
3. The graph is optimized.
4. Linear algebra node lowering takes place.
5. Additional rounds of optimizations occur, both target independent and target specific.
6. The graph is scheduled into a linear sequence of nodes that minimizes memory usage.
7. IRGen converts the low-level graph into instructions.
8. Low-level IR optimizations are performed.
9. Backend-specific optimizations and code generation are performed.

From the Glow paper

# Glow - CPU Backend

Glow could call BLAS or Eigen to implement DL operators

These have runtime checks and optimized implementation are dependent on specific tensor sizes, address of buffer, pointer aliases.
Low-level IR has this information, libraries do not!

Glow implements a small standard library that is compiled into LLVM bitcode.

Glow also runs operator stacking optimization

Low-level IR gets lowered into LLVM IR

LLVM IR optimizations take place

machine code is emitted

# PyTorch 2 (2024)

TorchDynamo - Graph Compilation

TorchInductor - Compiler

```python
import torch
import torch.nn as nn

class TinyMLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(16, 32)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(32, 8)

    def forward(self, x):
        return self.fc2(self.relu(self.fc1(x)))

model = TinyMLP()

@torch.compile
def run_model(x):
    return model(x)

x = torch.randn(4, 16)
y = run_model(x)

print(y.shape)
```

# TorchInductor - Decomposition

```
log2_scale = 1 / math.log(2)

@register_decomposition(torch.ops.aten.log2)
def log2(x):
  return torch.log(x) * log2_scale
```

Simplifies graph operators into a simpler set of operators

TorchInductor has 191 decompositions!

# TorchInductor - Lowering

```
def inner_fn_buf0(index):
  i0, i1 = index
  tmp0 = ops.load("arg0_1", i0 * s1 + i1)
  tmp1 = ops.log(tmp0)
  tmp2 = ops.constant(1.4426950408889634, torch.float32)
  tmp3 = ops.mul(tmp1, tmp2)
  return tmp3

buf0_ir = TensorBox(StorageBox(ComputedBuffer(
  name='buf0',
  layout=FixedLayout('cuda', torch.float32,
                     size=[s0, s1], stride=[s1, 1]),
  data=Pointwise(inner_fn=inner_fn_buf0,
                 ranges=[s0, s1], ...))))
```

**Figure 2.** TorchInductor IR for `torch.log2` on a 2D tensor.

```
ops.load
ops.store
ops.reduction
ops.index_expr
ops.masked
ops.rand
ops.mul
ops.add
```

TorchInductor has lowers for 433 PyTorch operators

# TorchInductor - Scheduling

Determines which operators get fused, order kernel run in, memory planning

Every buffer in the IR converted into a subclass of **`BaseSchedulerNode`**

- **`SchedulerNode`**
- **`ExternKernelSchedulerNode`**
- **`NopKernelSchedulerNode`**
- **`FusedSchedulerNode`**

# TorchInductor - Scheduling

```
scheduler.can_fuse(node1, node2)
scheduler.score_fusion(node1, node2)
```

Until no additional fusion remains

1) Find all fusion opportunities
2) Score and sort fusion opportunities
3) Check if fusion is legal and apply it

# TorchInductor - Triton Code Generation

Maps TorchInductor's IR to Triton kernels

```
def inner_fn_buf0(index):
  i0, i1 = index
  tmp0 = ops.load("arg0_1", i0 * s1 + i1)
  tmp1 = ops.log(tmp0)
  tmp2 = ops.constant(1.4426950408889634, torch.float32)
  tmp3 = ops.mul(tmp1, tmp2)
  return tmp3

buf0_ir = TensorBox(StorageBox(ComputedBuffer(
  name='buf0',
  layout=FixedLayout('cuda', torch.float32,
                     size=[s0, s1], stride=[s1, 1]),
  data=Pointwise(inner_fn=inner_fn_buf0,
                 ranges=[s0, s1], ...))))
```

**Figure 2.** TorchInductor IR for `torch.log2` on a 2D tensor.

```
@pointwise(...)
@triton.jit
def kernel(in_ptr0, out_ptr0, xnumel, XBLOCK : tl.constexpr):
    xoffset = tl.program_id(0) * XBLOCK
    xindex = xoffset + tl.arange(0, XBLOCK)[:]
    xmask = xindex < xnumel
    x0 = xindex
    tmp0 = tl.load(in_ptr0 + x0, xmask)
    tmp1 = tl.log(tmp0)
    tmp2 = 1.4426950408889634
    tmp3 = tmp1 * tmp2
    tl.store(out_ptr0 + x0, tmp3, xmask)
```

**Figure 3.** Generated Triton code for Figure 2.


- ➔ Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines
- ➔ TVM: An Automated End-to-End Optimizing Compiler for Deep Learning

# Halide (2013)

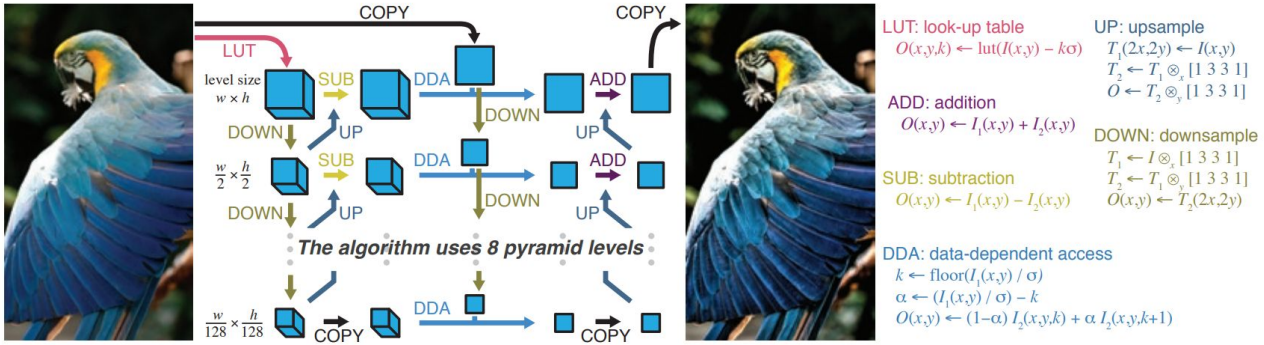



**Figure 1.** Imaging pipelines employ large numbers of interconnected, heterogeneous stages. Here we show the structure of the local Laplacian filter [3, 22], which is used for a variety of tasks in photographic post-production. Each box represents intermediate data, and each arrow represents one or more functions that define that data. The pipeline includes horizontal and vertical stencils, resampling, data-dependent gathers, and simple pointwise functions.

**Problem - Adobe has deep, wide image pipelines needing to be optimized on many different hardware.**

Current tools require handwriting C, C++, CUDA kernels, infeasible!

**Solution - Halide DSL and compiler**

# Halide - Motivation

```
// Build Gaussian pyramid
for (int l = 1; l < LEVELS; l++) {
    Image blur_x(gaussian[l-1].width, gaussian[l-1].height);
    Image blur_y(gaussian[l-1].width, gaussian[l-1].height);

    for (int y = 0; y < gaussian[l-1].height; y++) {
        for (int x = 0; x < gaussian[l-1].width; x++) {
            blur_x(x, y) =
                gaussian[l-1](x-1, y) +
                3 * gaussian[l-1](x, y) +
                3 * gaussian[l-1](x+1, y) +
                gaussian[l-1](x+2, y);
        }
    }

    for (int y = 0; y < gaussian[l-1].height; y++) {
        for (int x = 0; x < gaussian[l-1].width; x++) {
            blur_y(x, y) =
                blur_x(x, y-1) +
                3 * blur_x(x, y) +
                3 * blur_x(x, y+1) +
                blur_x(x, y+2);
        }
    }

    gaussian[l] = Image(gaussian[l-1].width / 2, gaussian[l-1].height / 2);
```

```
// Build Laplacian pyramid
for (int l = 0; l < LEVELS - 1; l++) {
    Image up(gaussian[l].width, gaussian[l].height);

    for (int y = 0; y < gaussian[l].height; y++) {
        for (int x = 0; x < gaussian[l].width; x++) {
            up(x, y) = gaussian[l+1](x / 2, y / 2);
        }
    }

    laplacian[l] = Image(gaussian[l].width, gaussian[l].height);

    for (int y = 0; y < gaussian[l].height; y++) {
        for (int x = 0; x < gaussian[l].width; x++) {
            laplacian[l](x, y) = gaussian[l](x, y) - up(x, y);
        }
    }
}

laplacian[LEVELS - 1] = gaussian[LEVELS - 1];
```

```
// For each pyramid level, do data-dependent lookup / interpolation
for (int l = 0; l < LEVELS; l++) {
    output_laplacian[l] = Image(laplacian[l].width, laplacian[l].height);

    for (int y = 0; y < laplacian[l].height; y++) {
        for (int x = 0; x < laplacian[l].width; x++) {
            float intensity = gaussian[l](x, y) * (BINS - 1);

            int k = floor(intensity);
            float alpha = intensity - k;

            float low  = lookup_table(k,     laplacian[l](x, y));
            float high = lookup_table(k + 1, laplacian[l](x, y));

            output_laplacian[l](x, y) =
                (1.0f - alpha) * low + alpha * high;
        }
    }
}
```

**C++ Implementation of local Laplacian filters has dozens of loop nests and hundreds of lines of code!**

**Different hardware will have different performance based on loop structure**

# Halide - DSL Algorithm

```
UniformImage in(UInt(8), 2)
Var x, y
Func blurx(x,y) =  in(x-1,y) +    in(x,y) +    in(x+1,y)
Func out(x,y) = blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)
```

```
UniformImage in(UInt(8), 2)
RDom r(0..in.width(), 0..in.height()), ri(0..255)
Var x, y, i
Func histogram(i) = 0; histogram(in(r.x, r.y))++
Func cdf(i) = 0; cdf(ri) = cdf(ri-1) + histogram(ri)
Func out(x, y) = cdf(in(x, y))
```

**Halide decouples the algorithm from the strategy**

# Halide - Scheduling Motivation

- When and where should the value at each coordinate in each function be computed?
- Where should they be stored?
- How long are values cached and communicated across multiple consumers, and when are they independently recomputed by each?

Choices that don’t affect the computed value but affect performance

Three axis:

producer-consumer locality

parallelism

redundant recomputation of shared values

# Halide - Scheduling Motivating Example

## Two stage pipeline

```
blurx(x,y) = in(x-1,y) + in(x,y) + in(x+1,y)

out(x,y) = blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)
```

# Halide - Scheduling Motivating Example

**Breadth-first schedule**

```
alloc blurx[2048][3072]
for each y in 0..2048:
  for each x in 0..3072:
    blurx[y][x] = in[y][x-1] + in[y][x] + in[y][x+1]
alloc out[2046][3072]
for each y in 1..2047:
  for each x in 0..3072:
    out[y][x]=blurx[y-1][x] + blurx[y][x] + blurx[y+1][x]
```

**Fused schedule**

```
alloc out[2046][3072]
for each y in 1..2047:
  for each x in 0..3072:
    alloc blurx[-1..1]
    for each i in -1..1:
      blurx[i]= in[y-1+i][x-1]+in[y-1+i][x]+in[y-1+i][x+1]
    out[y][x] = blurx[0] + blurx[1] + blurx[2]
```

**Sliding window schedule**

```
alloc out[2046][3072]
alloc blurx[3][3072]
for each y in -1..2047:
  for each x in 0..3072:
    blurx[(y+1)%3][x]=in[y+1][x-1]+in[y+1][x]+in[y+1][x+1]
    if y < 1: continue
    out[y][x] = blurx[(y-1)%3][x]
              + blurx[ y % 3 ][x]
              + blurx[(y+1)%3][x]
```

# Halide - Scheduling

Scheduling Choice Space
- What granularity to compute and store in?
- Within those grains, what order to traverse?

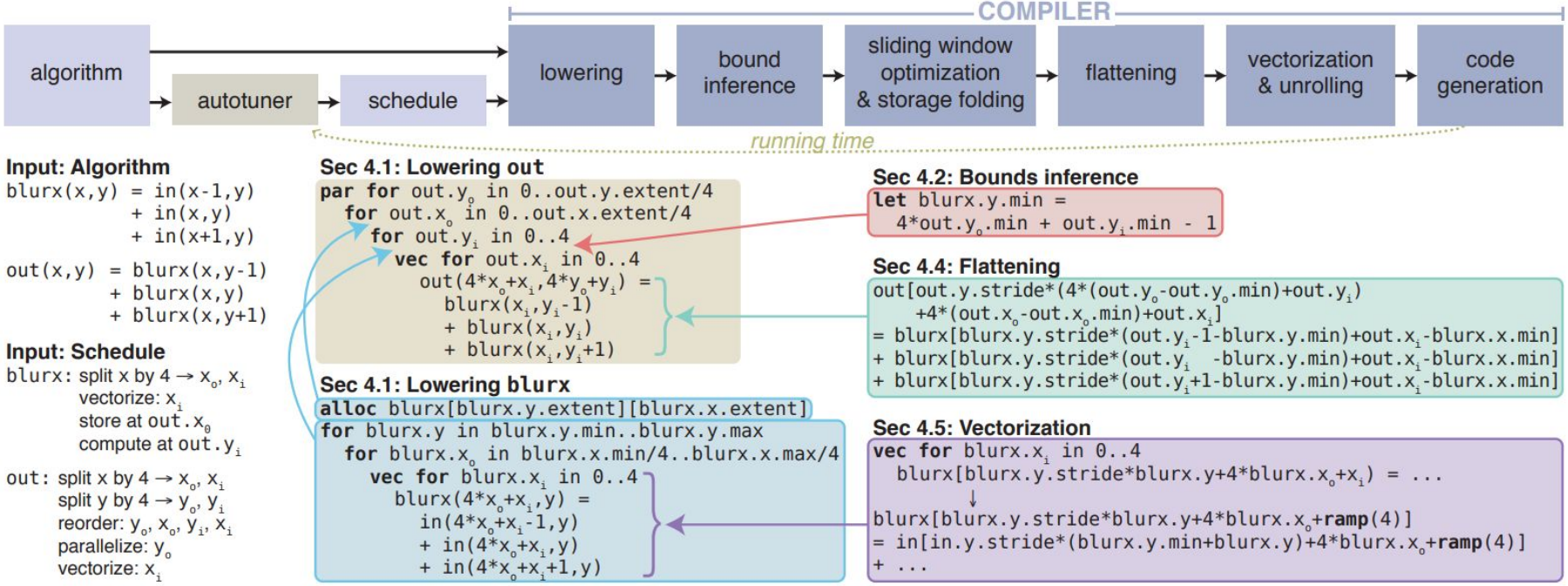
Domain Order
- Dimension traversed sequentially or in parallel
- Unrolled or vectorized
- Column vs. Row major
- Splitting loop
  - Recursively creates more choices

Call Schedule
- breadth-first schedule
- fused schedule
- sliding window schedule

# Halide

```
Func blur_3x3(Func input) {
  Func blur_x, blur_y;
  Var x, y, xi, yi;

  // The algorithm - no storage or order
  blur_x(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;
  blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1))/3;

  // The schedule - defines order, locality; implies storage
  blur_y.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
  blur_x.compute_at(blur_y, x).vectorize(x, 8);

  return blur_y;
}
```

# Halide - Compiler



# TVM (2018)

**Problem: Most ML compilers take high level graph level optimizations, then require manual tuning for operator level implementation**

Even if backend is supported,

- Avoid graph optimizations that yield unsupported operators
- We may have unoptimized operators

As more unique devices come out, this will become infeasible to support!

# TVM (2018)

**Solution: TVM uses ideas from Halide and refactors them for ML compilers!**

1) Tensor Expression Language
2) Automated Program Optimization
3) Graph Rewriter

# TVM

```
import tvm as t
# Use keras framework as example, import model
graph, params = t.frontend.from_keras(keras_model)
target = t.target.cuda()
graph, lib, params = t.compiler.build(graph, target, params)
```

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

**`graph`**
Final optimized computational graph

**`lib`**
Generated operators

**`params`**
Module parameters

# TVM - Graph Rewriter

**Operator Fusion**

**Constant Folding**

**Static Memory Planning**

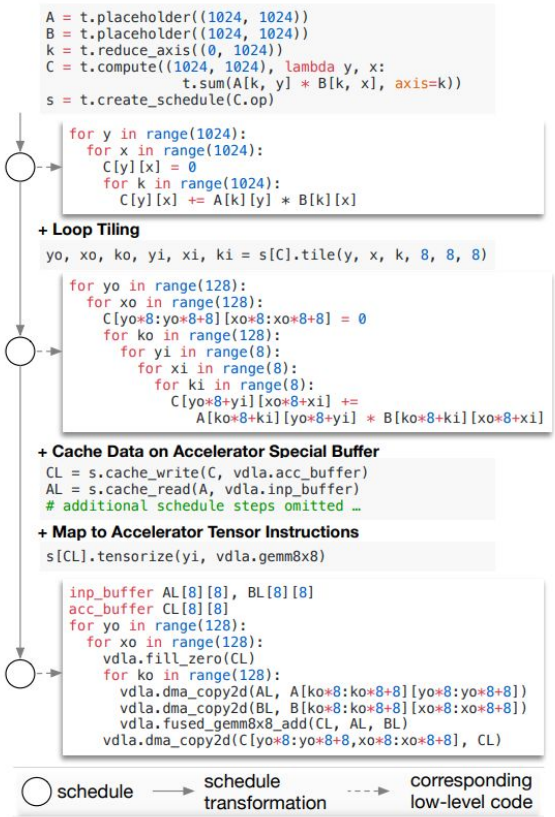
**Data Layout Transformations**

Convert data layouts to utilize target device hardware characteristics

**Creates unique graph operators not supported by kernel libraries!**

# TVM - Tensor Expression Language

- Decoupling of compute and scheduling
- Internal data structure to track loop structure
- Incrementally apply schedule transformation
- Need to support schedule primitives for different target devices



# TVM - Nested Parallelism with Cooperation

```
for thread_group (by, bx) in cross(64, 64):
  for thread_item (ty, tx) in cross(2, 2):
    local CL[8][8] = 0
    shared AS[2][8], BS[2][8]
    for k in range(1024):
      for i in range(4):
        AS[ty][i*4+tx] = A[k][by*64+ty*8+i*4+tx]
      for each i in 0..4:
        BS[ty][i*4+tx] = B[k][bx*64+ty*8+i*4+tx]
      memory_barrier_among_threads()
      for yi in range(8):
        for xi in range(8):
          CL[yi][xi] += AS[yi] * BS[xi]
      for yi in range(8):
        for xi in range(8):
          C[yo*8+yi][xo*8+xi] = CL[yi][xi]
```

All threads cooperatively load AS and BS in different parallel patterns

Barrier inserted automatically by compiler

Nested Parallelism (shared-nothing nested parallelism)

Introduces memory scopes for thread cooperation

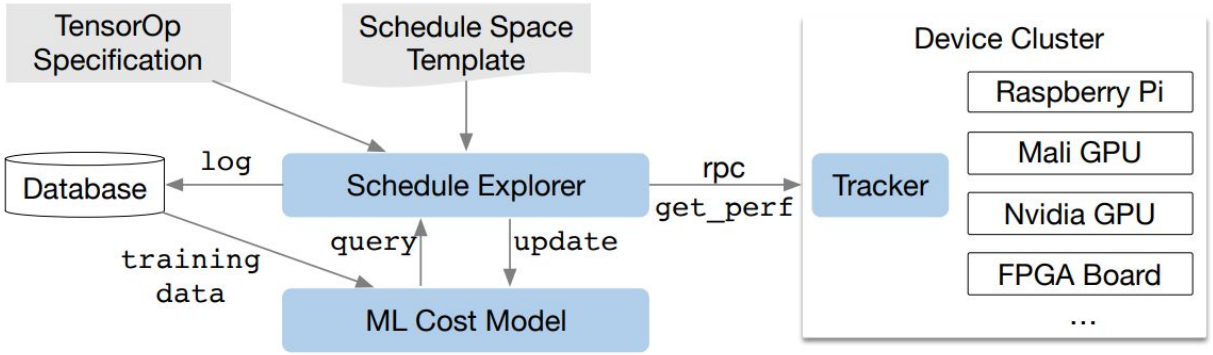
# TVM - Tensorization

```
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)
```

declare behavior

lowering rule to generate hardware intrinsics to carry out the computation

# TVM - Autotuning




- Gradient Boosting model (XGBoost)
- Enumerate through each configuration, return top-k -> too slow
- Random walk through configuration space until cost model predicts lower cost

Overview
Compilers
**Papers**
Future

# Papers

TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions

Relay: A New IR for Machine Learning Frameworks

Ansor: Generating High-Performance Tensor Programs for Deep Learning

Relax: Composable Abstractions for End-to-End Dynamic Machine Learning

Overview
Compilers
Papers
**Future**

# Mega-Kernels

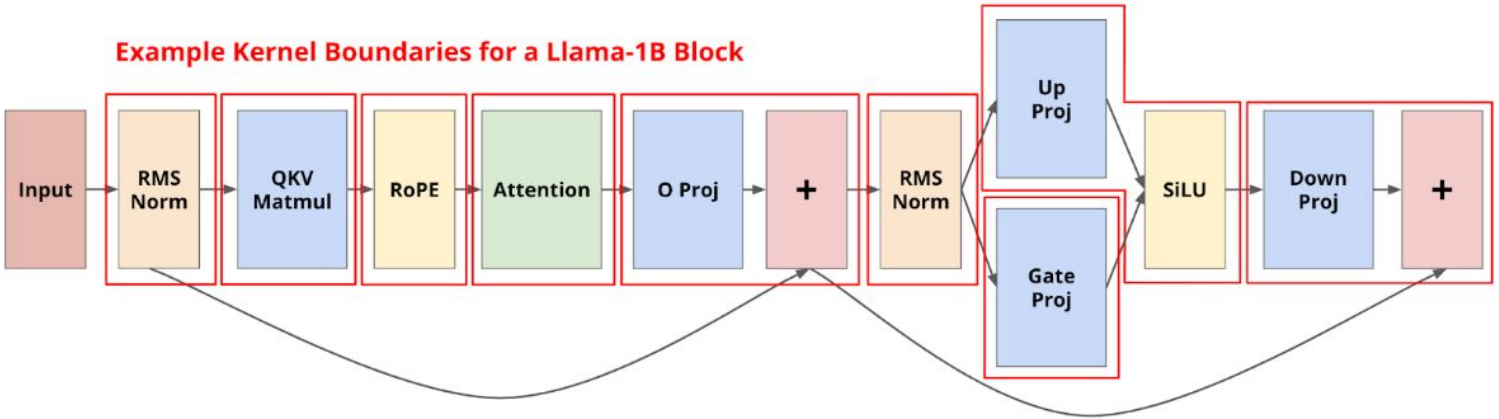
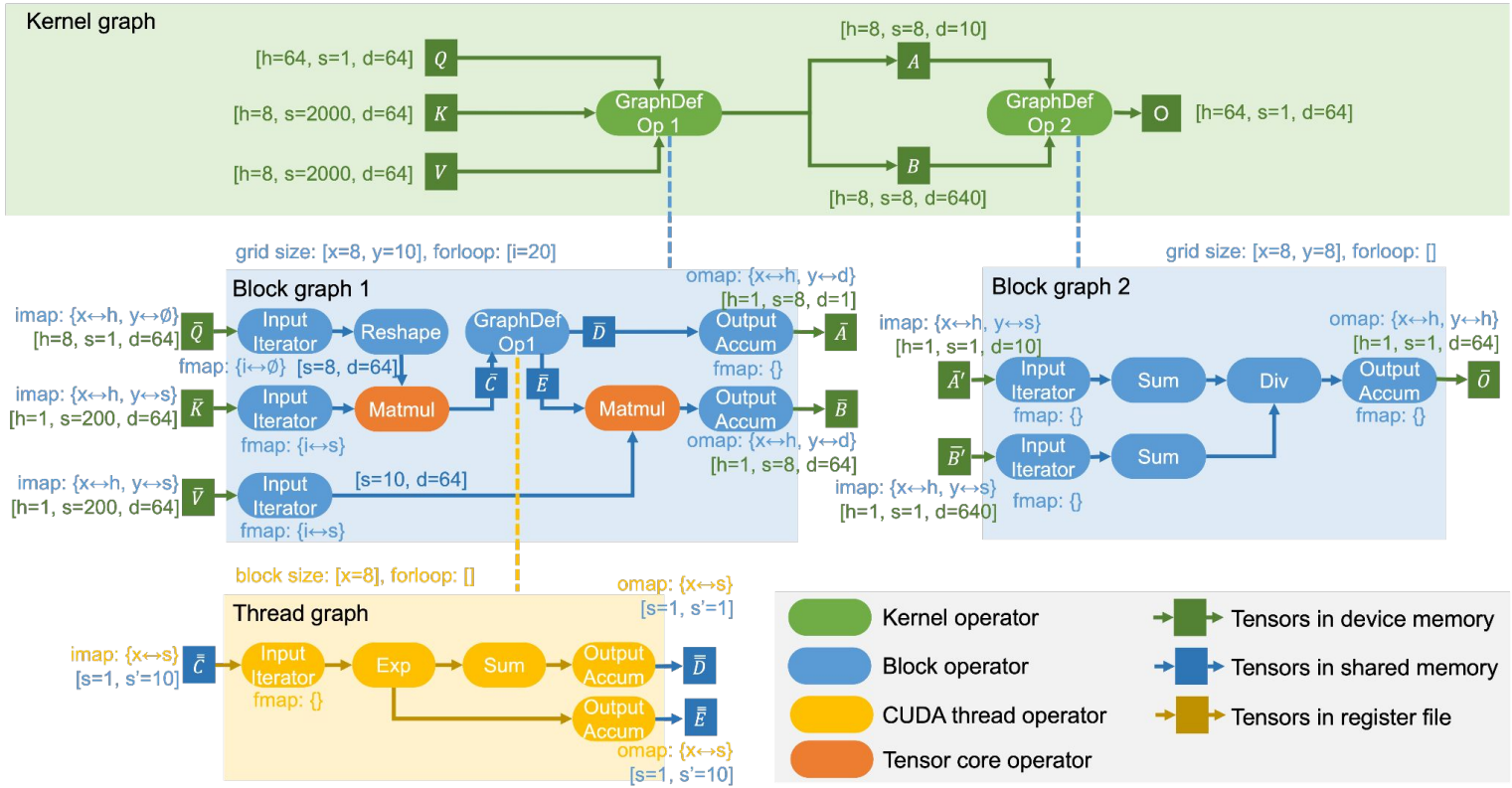
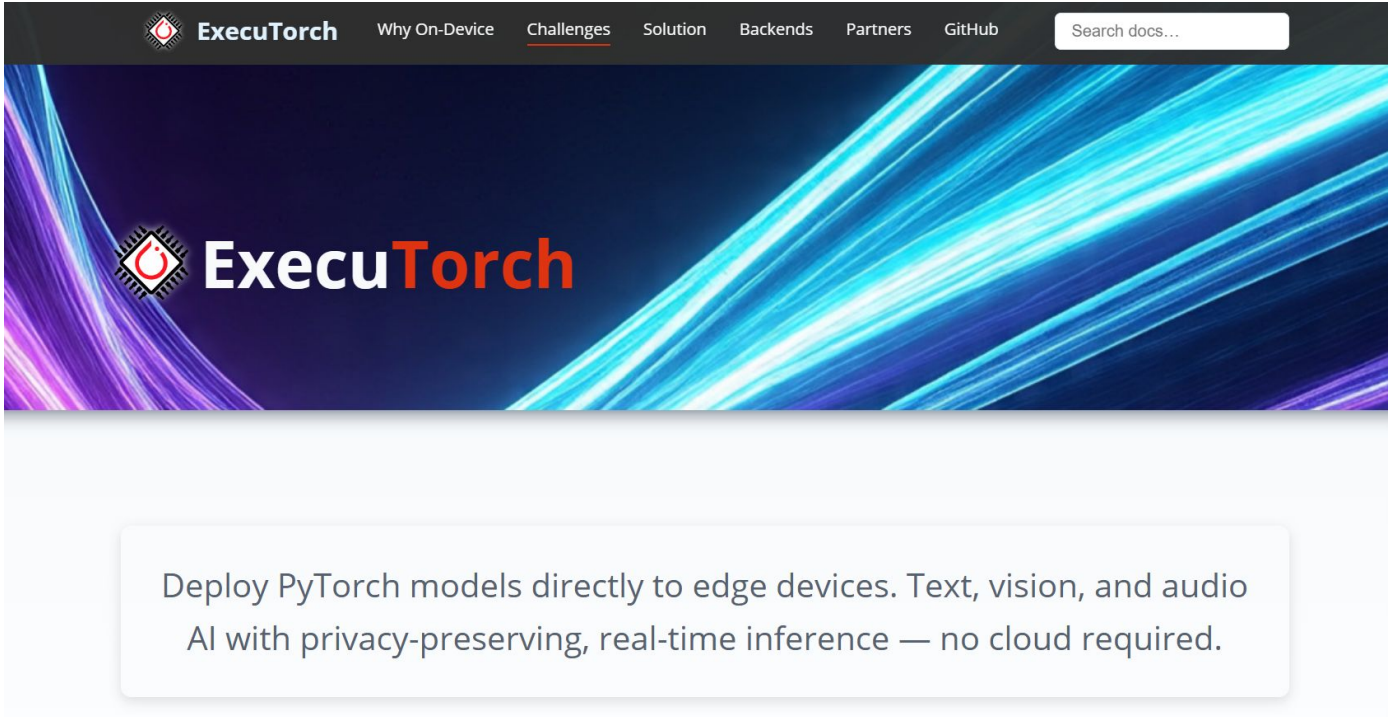



*Figure 2: An example set of kernel boundaries for the Llama-1B transformer block. Red boxes delineate the work done by individual kernels.*

# Mirage


Kernel graph
[h=64, s=1, d=64]
Q
[h=8, s=2000, d=64]
K
[h=8, s=2000, d=64]
V
GraphDef Op 1
[h=8, s=8, d=10]
A
B
[h=8, s=8, d=640]
GraphDef Op 2
O
[h=64, s=1, d=64]
grid size: [x=8, y=10], forloop: [i=20]
Block graph 1
imap: {x↔h, y↔∅}
[h=8, s=1, d=64]
Input Iterator
Reshape
fmap: {i↔∅} [s=8, d=64]
GraphDef Op1
D
Output Accum
fmap: {}
omap: {x↔h, y↔d}
[h=1, s=8, d=1]
Ā
imap: {x↔h, y↔s}
[h=1, s=200, d=64]
K̄
Input Iterator
Matmul
fmap: {i↔s}
C̄
Ē
Matmul
Output Accum
omap: {x↔h, y↔d}
[h=1, s=8, d=64]
B̄
imap: {x↔h, y↔s}
[h=1, s=200, d=64]
V̄
Input Iterator
[s=10, d=64]
fmap: {i↔s}
grid size: [x=8, y=8], forloop: []
Block graph 2
imap: {x↔h, y↔s}
[h=1, s=1, d=10]
Ā′
Input Iterator
fmap: {}
Sum
Div
Output Accum
fmap: {}
omap: {x↔h, y↔h}
[h=1, s=1, d=64]
Ō
B̄′
Input Iterator
fmap: {}
Sum
imap: {x↔h, y↔s}
[h=1, s=1, d=640]
block size: [x=8], forloop: []
Thread graph
imap: {x↔s}
[s=1, s'=10]
C̄
Input Iterator
fmap: {}
Exp
Sum
Output Accum
D̄
Output Accum
Ē
omap: {x↔s}
[s=1, s'=1]
omap: {x↔s}
[s=1, s'=10]
Kernel operator
Block operator
CUDA thread operator
Tensor core operator
Tensors in device memory
Tensors in shared memory
Tensors in register file

# ExecuTorch


ExecuTorch
Why On-Device
Challenges
Solution
Backends
Partners
GitHub
Search docs...
ExecuTorch
Deploy PyTorch models directly to edge devices. Text, vision, and audio AI with privacy-preserving, real-time inference — no cloud required.