AI-Systems

Deep Learning

Compilers

Some content has been borrowed from:
• Simon Mo’s Lecture (Ai-Sys SP19)
• UW-CSE 599W Systems for ML Class

Joseph E. Gonzalez
Co-director of the RISE Lab
jegonzal@cs.berkeley.edu
Deep Learning Execution Model

- DL frameworks execute the network by running one operator at a time
  - May **optimize choice** of operator implementation
  - Each operator reads input and produces new output

- Issues?

Example (Conv Op): `volta_scudnn_winograd_1 28x128_ldg1_ldg4_relu_tile 148t_nt_v1`

Image from [http://torch.ch/blog/2016/02/04/resnets.html](http://torch.ch/blog/2016/02/04/resnets.html)
Issues with operator at a time execution model

- Interpreted execution
- Multiple scans of data
  - Potentially large temp. memory requirements
- Need optimized implementations of operators
  - Difficult to build new ops.
  - Difficult to target new hardware

Example (Conv Op):
volta_scudnn_winograd_128x128_ldg1_ldg4_relu_tile148t_nt_v1

Image from http://torch.ch/blog/2016/02/04/resnets.html
Hardware for Deep Learning

- **Heterogenous hardware**: Need to optimize workload for different hardware.

- **Layered Memory Hierarchy**: Complex scheduling space

- **Parallel Compute Primitives**
  - Threads
  - SIMD/Vector parallelism
  - Specialized primitives (e.g., Tensor Cores)
Challenges of Implementing Ops.

Basic gemm operation from TVM paper

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

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

\[
\begin{align*}
\text{for yo in range(128):} \\
&\hspace{1em} \text{for xo in range(128):} \\
&\hspace{2em} \text{C[y][x] = 0} \\
&\hspace{2em} \text{for k in range(1024):} \\
&\hspace{3em} \text{C[y][x] += A[k][y] * B[k][x]}
\end{align*}
\]

+ Cache Data on Accelerator Special Buffer
\[
\begin{align*}
\text{CL} &= s.cache_write(C, \text{vdla.acc_buffer}) \\
\text{AL} &= s.cache_read(A, \text{vdla.inp_buffer}) \\
\end{align*}
\]

+ Map to Accelerator Tensor Instructions
\[
s[\text{CL}.\text{tensorize(yi, vdla.gemm8x8)}]
\]

\[
\begin{align*}
&\text{inp_buffer AL[8][8], BL[8][8]} \\
&\text{acc_buffer CL[8][8]} \\
&\text{for yo in range(128):} \\
&\hspace{1em} \text{for xo in range(128):} \\
&\hspace{2em} \text{vdla.fill_zero(CL)} \\
&\hspace{2em} \text{for ko in range(128):} \\
&\hspace{3em} \text{vdla.dma_copy2d(AL, A[ko*8:ko*8+8][yo*8:yo*8+8])} \\
&\hspace{3em} \text{vdla.dma_copy2d(BL, B[ko*8:ko*8+8][xo*8:xo*8+8])} \\
&\hspace{3em} \text{vdla.fused_gemm8x8_add(CL, AL, BL)} \\
&\hspace{3em} \text{vdla.dma_copy2d(C[yo*8:yo*8+8][xo*8:xo*8+8], CL)}
\end{align*}
\]

Graphs:
- Need to reason about:
  - Loop order and Tiling
  - Memory layout
  - Relation to other operations

- Relationship to memory hierarchy and specialized hardware
Compiler’s Perspective

Frameworks
- TensorFlow
- PyTorch
- Caffe2
- CNTK

Express computation

Intermediate Representation (s)

Reusable Optimizations

Code generation

Hardware

Slide borrowed from UW-CSE 599W Systems for ML Class
Computation Graph as IR

Represent High level Deep Learning Computations

- **Attributes**
  - channels=32
  - kernel_size=(3,3)
  - paddings=(1,1)
  - use_bias=0

- **Shape**
  - (1,10)

- **Data**
  - data

- **Operations**
  - conv2d
  - relu
  - bn
  - flatten
  - dense
  - softmax

Approach taken by: TensorFlow XLA, Intel NGraph, Nvidia TensorRT

Effective Equivalent Transformations to Optimize the Graph

- **Operation**
  - fused-conv2d-bn-relu

- **Input Dataflow Dependency**
  - relu

Slide borrowed from UW-CSE 599W Systems for ML Class
TensorFlow XLA Compiler

- XLA HLO is an IR composed to tensor operations
- Generates optimized binaries to evaluate models
  - Fuses kernels → eliminating reads and writes to slow memory
  - Optimized data layout
  - Reduced environment size
- User still needs to implement optimized tensor ops for each architecture
  - Smaller set then all of TF
Nvidia TensorRT

- Nvidia’s platform for optimizing deep neural networks
- Quantization of weights
- Data layout and kernel section
- Fuses kernels -- Vertically (conv, relu) and horizontally (reuse inputs)
Intermediate Representation (IR) Approaches

Computation Graph

- DAG Optimization:
  - Operator Fusion
  - No-op Elimination

Typically leverage pre-existing tensor operations

Tensor Loop Algebra

- Halide
  - Optimize loop order, tiling, and memory layout across operators in DAG
  - Support new operator design

MetaFlow

TensorRT

Halide

tvvm

Tensor Comprehensions
Halide: Compiling Image Processing Pipelines

Key Innovation:
- Decouples *algorithm from the compute*
- *So we can express operator in a simple language*
Halide: Compiling Image Processing Pipelines

Key Innovation:
- Decouples **algorithm from the compute**
- User only needs to provide the algorithm, and optionally the schedule.

```
Input: Algorithm
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)
```

```
Input: Schedule
blurx: split x by 4 -> x_i, x_i
  vectorize: x_i
  store at out. y_0
  compute at out. y_0
out: split x by 4 -> x_i, x_i
  split y by 4 -> y_i, y_i
  reorder: y_i, x_i, y_i, x_i
  parallelize: y_i
  vectorize: x_i
```
Halide: Compiling Image Processing Pipelines

Key Innovation:
- Decouples **algorithm from the compute**
- User only needs to provide the algorithm

Auto-tuner can select the optimal “schedule”
- How to split the axis?
- How to vectorize?

---

Input: Algorithm
\[
\text{blur}(x, y) = \text{in}(x-1, y) \\
+ \text{in}(x, y) \\
+ \text{in}(x+1, y)
\]
\[
\text{out}(x, y) = \text{blur}(x, y-1) \\
+ \text{blur}(x, y) \\
+ \text{blur}(x, y+1)
\]

Input: Schedule
- \text{blur}: split x by 4 \rightarrow x_o, x_i
- vectorize: x_i
- store at \text{out}.x_0
- compute at \text{out}.y_i
- \text{out}: split x by 4 \rightarrow x_o, x_i
- split y by 4 \rightarrow y_o, y_i
- reorder: y_o, x_o, y_i, x_i
- parallelize: y_o
- vectorize: x_i
Halide: Compiling Image Processing Pipelines

Key Innovation:
- Decouples **algorithm from the compute**
- User only needs to provide the algorithm

Auto-tuner can select the optimal “schedule”
- How to split the axis?
- How to vectorize?

Input: Algorithm
\[
\text{blur}(x, y) = \text{in}(x-1, y) + \text{in}(x, y) + \text{in}(x+1, y)
\]
\[
\text{out}(x, y) = \text{blur}(x+1, y)+ \text{blur}(x-1, y) + \text{blur}(x, y)
\]

Input: Schedule
- **blur**: split $x$ by 4 $\rightarrow x_0, x_1$
  - vectorize: $x_1$
  - store at out. $x_0$
  - compute at out. $y_1$
- **out**: split $x$ by 4 $\rightarrow x_0, x_1$
  - split by 4 $\rightarrow y_0, y_1$
  - reorder: $y_0, x_0, y_1, x_1$
  - parallelize: $y_0$
  - vectorize: $x_1$
Halide: Compiling Image Processing Pipelines

Key Innovation:
- Decouples **algorithm from the compute**
- User only needs to provide the algorithm

Auto-tuner can select the optimal “schedule”
- How to split the axis?
- How to vectorize?

**Input: Algorithm**
\[
\text{blurx}(x, y) = \text{in}(x-1, y) + \text{in}(x, y) + \text{in}(x+1, y)
\]
\[
\text{out}(x, y) = \text{blurx}(x, y-1) + \text{blurx}(x, y) + \text{blurx}(x, y+1)
\]

**Input: Schedule**
- \text{blurx}: split \(x\) by \(4\rightarrow x_0, x_1\)
- vectorize: \(x_1\)
- store at \(\text{out}.x_0\)
- compute at \(\text{out}.y_1\)

- \text{out}: split \(x\) by \(4\rightarrow x_0, x_1\)
- split \(y\) by \(4\rightarrow y_0, y_1\)
- reorder: \(y_0, x_0, y_1, x_1\)
- parallelize: \(y_0\)
- vectorize: \(x_1\)

**Sec 4.1: Lowering out**
\[
\begin{align*}
\text{par for} & \quad \text{out}.y_0 \text{ in } 0..\text{out}.y.\text{extent}/4 \\
& \text{for} \quad \text{out}.x_0 \text{ in } 0..\text{out}.x.\text{extent}/4 \\
& \quad \text{for} \quad \text{out}.y_1 \text{ in } 0..4 \\
& \quad \quad \text{vec for} \quad \text{out}.x_1 \text{ in } 0..4 \\
& \quad \quad \quad \text{out}(4*x_0+x_1,4*y_0+y_1) = \\
& \quad \quad \quad \quad \text{blurx}(x_0, y_1-1) \\
& \quad \quad \quad \quad \text{blurx}(x_0, y_1) \\
& \quad \quad \quad \quad \text{blurx}(x_0, y_1+1) \\
\end{align*}
\]

**Sec 4.2: Bounds inference**
\[
\text{let} \quad \text{blurx}.y.\text{min} = 4*\text{out}.y.\text{min} + \text{out}.y.\text{min} - 1
\]

**Sec 4.4: Flattening**
\[
\begin{align*}
& \text{out}(\text{out}.y.\text{stride}*(4*(\text{out}.y.\text{-out}.y.\text{min})+\text{out}.y.)) \\
& + 4*(\text{out}.x.\text{-out}.x.\text{min})+\text{out}.x] \\
= \text{blurx}[\text{blurx}.y.\text{stride}*(\text{out}.y.\text{-1-}1\text{blurx}.y.\text{min})+\text{out}.x\text{-blurx}.x.\text{min}] \\
+ \text{blurx}[\text{blurx}.y.\text{stride}*(\text{out}.y.\text{-blurx}.y.\text{min})+\text{out}.x\text{-blurx}.x.\text{min}] \\
+ \text{blurx}[\text{blurx}.y.\text{stride}*(\text{out}.y.\text{1-1-}1\text{blurx}.y.\text{min})+\text{out}.x\text{-blurx}.x.\text{min}]
\end{align*}
\]

**Sec 4.5: Vectorization**
\[
\begin{align*}
& \text{vec for} \quad \text{blurx}.x_1 \text{ in } 0..4 \\
& \quad \text{blurx}[(\text{blurx}.y.\text{stride}*\text{blurx}.y+4\text{blurx}.x_0+x_1)] = \ldots \\
& \quad \text{blurx}[\text{blurx}.y.\text{stride}*[\text{blurx}.y.\text{min}+\text{blurx}.y]4*\text{blurx}.x_0+ramp(4)] \\
& + \ldots
\end{align*}
\]
Halide DSL

```cpp
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;
}
```

- Functional Language
- Embed in C++
- Much Simpler than writing threaded or CUDA program
- Downside:
  - Still requires domain experts to tune it
  - Not built for Deep Learning
  - TC: Assume infinite input range, cannot be optimized for fixed ops.
  - TVM: No special memory scope; no custom hardware intrinsics
Reading This Week
Reading for the Week

- **Optimizing DNN Computation with Relaxed Graph Substitutions** (SysML’19)
  - Improves the graph search but does not modify individual ops.
  - Leverages basic cost model

- **TVM: An Automated End-to-End Optimizing Compiler for Deep Learning** (OSDI’18)
  - Optimizes graph and then individual tensor operations
  - Uses learning based approach

- **Learning to Optimize Halide with Tree Search and Random Programs** (TOG’19)
  - Schedule optimization in Halide using hybrid learning based approach
MetaFlow (SysML’19)

- Optimizes graph only by transforming groups of operators into revised versions of existing operators

Key Insights:
- Use “backtracking” search to allow for less myopic opt.
- Cluster ops. using dep. flow analysis to identify subgraphs
- Static operator impl. have predictable costs
TVM

- Originally derived from Halide
- Leverages similar IR and separation of algorithm from schedule
TVM

- Originally derived from Halide
- Leverages similar IR and separation of algorithm from schedule

```python
import tvm
m, n, h = tvm.var('m'), tvm.var('n'), tvm.var('h')
A = tvm.placeholder((m, h), name='A')
B = tvm.placeholder((n, h), name='B')
k = tvm.reduce_axis((0, h), name='k')
C = tvm.compute((m, n), lambda i, j: tvm.sum(A[i, k] * B[j, k], axis=k))
```

**Shape of C**

```python
out = tvm.compute((c, h, w),
    lambda i, x, y: tvm.sum(data[kc,x+kx,y+ky] * w[i,kx,ky], [kx,ky,kc]))
```

**Computation Rule**

**Guess what this describes?**


**TVM**

- Enables declaring new hardware intrinsics
- Simplifies adding support for new hardware

\[
w, x = t.\text{placeholder}((8, 8)), t.\text{placeholder}((8, 8)) \\
k = t.\text{reduce_axis}((0, 8)) \\
y = t.\text{compute}((8, 8), \text{lambda } i, j: t.\text{sum}(w[i, k] \times x[j, k], \text{axis}=k))
\]

```python
def gemm_intrinsic_lower(inputs, outputs):
    ww_ptr = inputs[0].\text{access_ptr}("r")
    xx_ptr = inputs[1].\text{access_ptr}("r")
    zz_ptr = outputs[0].\text{access_ptr}("w")
    compute = t.\text{hardware_intrinsic}("gemm8x8", ww_ptr, xx_ptr, zz_ptr)
    reset = t.\text{hardware_intrinsic}("\text{fill_zero}", zz_ptr)
    update = t.\text{hardware_intrinsic}("fuse_gemmm8x8_add", ww_ptr, xx_ptr, zz_ptr)
    return compute, reset, update

gemm8x8 = t.\text{decl_tensor_intrinsic}(y.\text{op}, \text{gemm_intrinsic_lower})
```

---

**Section 3**

High Level Graph Rewriting

Optimized Computational Graph

---

**Section 4**

Declarative Tensor Expressions

Hardware-Aware Optimization Primitives

---

**Section 5**

Machine Learning Based Automated Optimizer

Optimized Low Level Loop Program

---

Deployable Module
Learning based auto-tuner

- Parametrized the AST
- Use Gradient Boosted Trees (GBT) to optimize a “rank loss” to predict the relative order of program runtime
Learning to Optimize Halide with Tree Search and Random Programs

- Published in ACM Transactions of Graphics (2019)
- Halide grew out of graphics community
- Addresses missing scheduler optimizer + auto-tuner
- Adopts learning based approach

---

**ACM Trans. Graph., Vol. 38, No. 4, Article 121.**

**ACM Reference Format:**

```
Andrew Adams, Tzu-Mao Li, with or without autotuning. It produces schedules which are on average
additional subset of schedules. We devise a parameterization of possible schedules. They generally work by
consider a small subset of all possible schedules. They generally work by
cost model by generating and featurizing hundreds of thousands of random
performance is hard: it requires expertise in hardware architecture
cost model + autotuning. The cost model is trained by benchmarking thousands of
result automatic scheduling
•
image processing and deep learning. We signi
cost model
•
•
•
•
```

---

**Additional Key Words and Phrases:**
- optimizing compilers
- Halide
- Tree Search and Random Programs
- optimizing compilers, Halide
- CCS Concepts:
- algorithm to signi
- Domain speci
- ne a parameterization of possible schedules
- are di
- consider a small subset of all possible schedules. They generally work by
cost model by generating and featurizing hundreds of thousands of random
performance is hard: it requires expertise in hardware architecture
cost model + autotuning. The cost model is trained by benchmarking thousands of
result automatic scheduling
•
image processing and deep learning. We signi
Learning to Optimize Halide with Tree Search and Random Programs

- **Beam search** of rich schedule space
  - Beam search ~ breadth first search with pruning
  - Search is constructed inductively from final stage in pipeline

h(x, y) = ...;
g(x, y) = pow(h(x, y), 1.8);
f(x, y) = g(x, y-1) + g(x, y+1);
Done!