Machine Learning Compiler: An Overview

Bojian Zheng

2022/4/1 @Amazon Reading Group
Agenda for Today

• Why Machine Learning Compilers?
  • State-of-the-Art Machine Learning Frameworks Design, and Flaws:
    1. Vendor libraries not delivering the optimal performance.

• 2 Classes of Machine Learning Compilers:
  • Halide\footnote{1}/TVM\footnote{2}: Easier to write high-performance programs.

---

\footnote{1} J. Ragan-Kelley et al. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. PLDI 2013

\footnote{2} T. Chen et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. OSDI 2018
Why Machine Learning Compilers?

- State-of-the-Art Machine Learning Frameworks Design:
  - Python Programming Front-end: define neural networks
  - C++ Framework Core: optimize graphs; schedule operators; allocate hardware resources ...
  - Vendor APIs & Libraries
  - Hardware Architectures

What can go wrong?
Machine Learning Frameworks

• State-of-the-Art Machine Learning Frameworks Design:

  - Python Programming Front-end: define neural networks
  - C++ Framework Core: optimize graphs; schedule operators; allocate hardware resources ...
  - Vendor APIs & Libraries
  - Hardware Architectures

Guaranteed optimal performance?
Vendor Libraries

• Example Workload:
  \[ Y = XW^\top \quad X : [M, K], W : [N, K] \]

• Q: How to efficiently handle for all cases of \((M, K, N)\)?

• Solutions (e.g., cuBLAS):
  • Provide efficient kernels that cover to all the use cases. E.g.,
    \[ y = xw^\top \]
    \[
    \begin{cases}
    x : [32, *], w : [32, *] \\
    x : [64, *], w : [64, *] \\
    x : [128, *], w : [128, *]
    \end{cases}
    \]
  • Dispatch at runtime to the most suitable kernel.
Vendor Libraries

Even as the shapes vary, cuBLAS only invokes a handful of kernels.

What if the workloads DO NOT fit into those kernels?

1 Color = 1 Unique CUDA Kernel

$Y = XW^T$, $X: [B \times T, 768]$, $W: [768, 768]$
Vendor Libraries

• Example Workload:
  \[ Y = XW^\top \quad X : [M, K], W : [N, K] \]

• Q: How to efficiently handle for all cases of \((M, K, N)\)?

• Solutions (e.g., cuBLAS):
  • Provide efficient kernels that cover to all the use cases. E.g.,
    \[
    \begin{aligned}
    y &= xw^\top \\
    \begin{align*}
    &x : [32, *], w : [32, *] \\
    &x : [64, *], w : [64, *] \\
    &x : [128, *], w : [128, *]
    \end{align*}
    \end{aligned}
    \]
  • Dispatch at runtime to the most suitable kernel.

\[ X : [129, K], W : [129, K] \]
Vendor Libraries

• Leads to **sub-optimal** performance (up to $13\times$) due to
  • Low Hardware Utilization\(^1\) and/or
  • Redundant Computations (by padding)\(^2\)

• Develop high-performance customized kernels?
  • Requires huge expertise + engineering efforts: thousands of lines per operator per architecture\(^3\).

---

\(^1\) F. Yu et al. Towards Latency-aware DNN Optimization with GPU Runtime Analysis and Tail Effect Elimination. arXiv 2020
\(^2\) Alibaba. Bringing TVM into TensorFlow for Optimizing Neural Machine Translation on GPU. 2018
\(^3\) NVIDIA. CUTLASS: CUDA Templates for Linear Algebra Subroutines. [https://github.com/NVIDIA/cutlass](https://github.com/NVIDIA/cutlass)
Tensor Program Compilers

• State-of-the-Arts: Halide\textsuperscript{[1]} & TVM\textsuperscript{[2]}
  • Halide: image processing
  • TVM: machine learning & better GPU support

• Objective: Easier to write high-performance programs.

• Key Idea: Abstracts low-level implementations using \textbf{schedules}.

\textbf{45 lines of Python Scheduling Code}

\begin{verbatim}
s = create_schedule(Y.op)
Y_local = s.cache_write([Y], "local")
i, j, k = tuple(Y_local.op.axis)
i_o_i, i_i = s[Y_local].split(i, factor=8)
i_o_o_i, i_o_i = s[Y_local].split(i_o_i, factor=2)
i_o_o_o, i_o_o_i = s[Y_local].split(i_o_o_i, factor=1)
...
\end{verbatim}
Tensor Program Compilers

  • Halide: image processing
  • TVM: machine learning & better GPU support
• Objective: Easier to write high-performance programs.
• Key Idea: Abstracts low-level implementations using schedules.

45 lines of Python Scheduling Code

```python
s = create_schedule(Y.op)
Y_local = s.cache_write([Y], "local")
i, j, k = tuple(Y_local.op.axis)
i_o_i, i_i = s[Y_local].split(i, factor=8)
i_o_o_i, i_o_i = s[Y_local].split(i_o_i, factor=2)
i_o_o_o, i_o_o_i = s[Y_local].split(i_o_o_i, factor=1)
...```

Tensor Program Compilers

- State-of-the-Arts: Halide\(^1\) & TVM\(^2\)
  - Halide: image processing
  - TVM: machine learning & better GPU support
- Objective: Easier to write high-performance programs.
- Key Idea: Abstracts low-level implementations using schedules.

45 lines of Python Scheduling Code

\[\texttt{s = create\_schedule}(Y.\texttt{op})\]
\[\texttt{Y\_local = s.cache\_write}([Y], \texttt{"local"})\]
\[\texttt{i, j, k = tuple(Y\_local.\texttt{op.axis})}\]
\[\texttt{i\_o\_i, i\_i = s[Y\_local].split(i, factor=8)}\]
\[\texttt{i\_o\_o\_i, i\_o\_i = s[Y\_local].split(i\_o\_i, factor=2)}\]
\[\texttt{i\_o\_o\_o, i\_o\_o\_i = s[Y\_local].split(i\_o\_o\_i, factor=1)}\]

946 lines of CUDA Code

\[\texttt{\_global\_ default\_function(...)}\ {\texttt{\{}}\]
\[\texttt{...}\]
\[\texttt{float Y\_local[128];}\]
\[\texttt{\_shared\_ float X\_shared[768];}\]
\[\texttt{\_shared\_ float W\_shared[384];}\]
\[\texttt{...}\]
\[\texttt{\}}\]

---

\(^1\) J. Ragan-Kelley et al. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. PLDI 2013

\(^2\) T. Chen et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. OSDI 2018
Tensor Program Compilers

  • Halide: image processing
  • TVM: machine learning & better GPU support
• Objective: Easier to write high-performance programs.
• Key Idea: Abstracts low-level implementations using schedules.

45 lines of Python Scheduling Code

```python
s = create_schedule(Y.op)
Y_local = s.cache_write([Y], "local")
i, j, k = tuple(Y_local.op.axis)
i_o_i, i_i = s[Y_local].split(i, factor=8)
i_o_o_i, i_o_i = s[Y_local].split(i_o_i, factor=2)
i_o_o_o, i_o_o_i = s[Y_local].split(i_o_o_i, factor=2)
```

946 lines of CUDA Code

```c
__global__ default_function(...) {
  ...
  float Y_local[128];
  __shared__ float X_shared[768];
  __shared__ float W_shared[384];
  ...
}
```

automatic inference

Tensor Program Compilers

Pros

• Easily programmed, modified and parameterized.

Cons

• Less flexible
  • What if my transformations are NOT supported by primitives?
  • Not quite “easily” programmed.

45 lines of Python Scheduling Code

```python
s = create_schedule(Y.op)
Y_local = s.cache_write([Y], "local")
i, j, k = tuple(Y_local.op.axis)
i_o_i, i_i = s[Y_local].split(i, factor=8)
i_o_o_i, i_o_i = s[Y_local].split(i_o_i, factor=2)
i_o_o_o, i_o_o_i = s[Y_local].split(i_o_o_i, factor=1)
...```

13
Tensor Program Compilers

**Pros**

• Easily programed, modified and parameterized.

**Cons**

• Less flexible
  • What if my transformations are NOT supported by primitives?
  • Not quite “easily” programmed.
  • Optimal schedules are hardware-specific.

45 lines of Python Scheduling Code

```python
s = create_schedule(Y.op)
Y_local = s.cache_write([Y], "local")
i, j, k = tuple(Y_local.op.axis)
i_o_i, i_i = s[Y_local].split(i, factor=8)
i_o_o_i, i_o_i = s[Y_local].split(i_o_i, factor=2)
i_o_o_o, i_o_o_i = s[Y_local].split(i_o_o_i, factor=1)
...```

Why (8, 2, 1)?

Want to generate high-performance schedules automatically.
Auto-Scheduler

- State-of-the-Arts: Halide\textsuperscript{[1]} and TVM\textsuperscript{[2]} Auto-Schedulers
- Objective: Given a compute definition, \textit{automatically} find a high-performance schedule.

\textsuperscript{[1]} A. Adams et al. Learning to Optimize Halide with Tree Search and Random Programs. ACM Transactions on Graphics (TOG) 2019
\textsuperscript{[2]} L. Zheng et al. Ansor: Generating High-Performance Tensor Programs for Deep Learning. OSDI 2020
Auto-Scheduler System Overview

\[ Y = XW^\top \quad X : [M, K], W : [N, K] \]

Sketch Generation Rules

```python
for i in range(2048):
    for j in range(2304):
        for k in range(768):
            Y[i][j] += X[i][k] * W[j][k]
```

\((M, K, N) = (2048, 768, 2304)\)
Auto-Scheduler System Overview

Operator Specification

Shape Description

Search Space

∞ plausible schedules

High-Performance Schedule

for i.o.o.o in range(∞):
    for i.o.o.i in range(∞):
        for i.o.i in range(∞):
            for i.i in range(∞):
                ...

Brute-force permutation on real hardware?

Time-consuming!
Auto-Scheduler System Overview

Auto-Scheduler

Operator Specification

Shape Description

Search Space

Cost Model

for i.o.o.o in range(?)
    for i.o.o.i in range(?)
    for i.o.i in range(?)
    for i.i in range(?):
        ...

Predict performance of many schedules simultaneously
Auto-Scheduler System Overview

Random Sample → Cost Model → Search Space → Auto-Scheduler

Operator Specification → Shape Description

```
for i.o.o.o in range(?)
    for i.o.o.i in range(?)
        for i.o.i in range(?)
            for i.i in range(?)
                ...
```
Auto-Scheduler System Overview

Auto-Scheduler

Random Sample

Operator Specification

Shape Description

Search Space

Cost Model

Train

Measure

for i.o.o.o in range(128):
    for i.o.o.i in range(8):
        for i.o.i in range(2):
            for i.i in range(1):
                ...

for i.o.o.o in range(128):
    for i.o.o.i in range(8):
        for i.o.i in range(2):
            for i.i in range(1):
                ...
Auto-Scheduler System Overview

Auto-Scheduler

Operator Specification

Shape Description

Search Space

Random Sample

Learned Cost Model

for i.o.o.o.o in range(128):
    for i.o.o.i in range(8):
        for i.o.i in range(2):
            for i.i in range(1):
                ...

Measured Throughputs

Measured and Predicted are strongly correlated.


Auto-Scheduler System Overview

Auto-Scheduler

Operator Specification
Shape Description
Search Space

Learned Cost Model
Guide

High-Performance Schedule
Auto-Scheduler Evaluation

Up to 1.7× better than the 2nd best alternative.
DietCode: Automatic Code Generation for Dynamic Tensor Programs

Bojian Zheng*, 1, 2, 3, Ziheng Jiang**, 4, 5, Cody Yu2, Haichen Shen2, Josh Fromm4, Yizhi Liu2, Yida Wang2, Luis Ceze4, 5, Tianqi Chen4, 6, Gennady Pekhimenko1, 2, 3

* Equal Contribution
Auto-Scheduler System Overview

- Operator Specification
- Shape Description
- Search Space
- Learned Cost Model
- Guide
- High-Performance Schedule
Auto-Scheduler System Overview

An operator has $\infty$ possible schedules

Compute:

```java
for (int i = 0; i < 50; ++i) {
    A[i] = ...
}
```
Auto-Scheduler System Overview

An operator has $\infty$ possible schedules

Example

Schedule:

```java
for (int io = 0; io < [50/t]; ++io) {
    for (int ii = 0; ii < t; ++ii) {
        if (io*t + ii < 50) A[io*t + ii] = ... 
    }
}
```

$t \in [2, \infty)$!
Auto-Scheduler System Overview

Limit the candidates to **perfect factors**

Example Schedule:

```java
for (int io = 0; io < [50/t]; ++io) {
    for (int ii = 0; ii < t; ++ii) {
        if (io*t + ii < 50) A[i] = ...
    }
}
```

`t ∈ {2, 5, 10, 25}`
Auto-Scheduler System Overview

Auto-Scheduler

Operator Specification

Static Shape Description

Shape-Dependent Search Space

Learned Cost Model

Guide

High-Performance Schedule
Challenges Faced by the Current System

• Challenge #1: Hard to share schedules across different shapes of the same operator.

Example

Schedule:
for (int io = 0; io < [50/t]; ++io) {
    for (int ii = 0; ii < t; ++ii) {
        A[io×t + ii] = ...  
    }
}

\( t \in \{2, 5, 10, 25\} \)
Challenges Faced by the Current System

• Challenge #1:
  • Hard to share schedules across different shapes of the same operator.

Example

Schedule:

```java
for (int io = 0; io < [49/t]; ++io) {
    for (int ii = 0; ii < t; ++ii) {
        A[io×t + ii] = ...
    }
}
```

$t ∈ \{7\}$ \bigcap\{2, 5, 10, 25\} = \ø
Challenges Faced by the Current System

• **Challenge #1:**
  - Hard to share schedules across different shapes of the same operator.

  **Example**

```c
Schedule:
for (int io = 0; io < \lfloor 49/t \rfloor; ++io) {
    for (int ii = 0; ii < t; ++ii) {
        A[io \times t + ii] = ...
    }
}
```

\[
t \in \{7\} \quad \cap \{2, 5, 10, 25\} = \emptyset
\]

Prohibitively expensive auto-scheduling time for dynamic-shape workloads.

---

**Operator Specification**

**Static Shape Description**

**Shape-Dependent Search Space**

**Learned Cost Model**

**Guide**

**High-Performance Schedule**

**Shape S_1** → auto-schedule → **Program P_1**

**Shape S_2** → auto-schedule → **Program P_2**

... → ... → ...
Challenges Faced by the Current System

• Challenge #2:
  • Can deliver sub-optimal performance for not considering non-perfect candidates.

Example

Schedule:
```c
for (int io = 0; io < 49/t; ++io) {
    for (int ii = 0; ii < t; ++ii) {
        if (io*t + ii < 49) A[io*t + ii] = ...
    }
}
```

Example

Schedule:
```c
for (int io = 0; io < 49/t; ++io) {
    for (int ii = 0; ii < t; ++ii) {
        if (io*t + ii < 49) A[io*t + ii] = ...
    }
}
```

Example

Schedule:
```c
for (int io = 0; io < 49/t; ++io) {
    for (int ii = 0; ii < t; ++ii) {
        if (io*t + ii < 49) A[io*t + ii] = ...
    }
}
```

Observation: Performance overhead of if-checks is negligible with local padding (i.e., pad tensors locally by the size of local and/or shared memory variables).
DietCode: A New Auto-Scheduler Framework
DietCode: Key Ideas

• Key Idea #1: **Shape-Generic Search Space**
  • Composed of **micro-kernels**. Each does a tile of the entire compute.
  • A micro-kernel can be ported to **all** shapes of the same operator.
  • Sampled from **hardware constraints** instead of shape factors (i.e., shape-generic).

Example:
\[
Y = XW^T \quad X: [1024, 768], W: [2304, 768]
\]
with micro-kernel dense_128x128, which evaluates
\[
Y = XW^T \quad X: [128, 768], W: [128, 768]
\]
DietCode: Key Ideas

• Key Idea #2: **Micro-Kernel-based Cost Model**
  • Observation: A cost model trained on one shape can be **inaccurate** on other shapes.
  • Compute throughputs exhibit **predictable linear** trend w.r.t. shape dimensions.
  • Decompose the cost model into:

\[ f_{\text{MK}} \cdot f_{\text{spatial}} \]
DietCode: Key Ideas

• Key Idea #2: **Micro-Kernel-based Cost Model**
  • Observation: A cost model trained on one shape can be **inaccurate** on other shapes.
  • Compute throughputs exhibit **predictable linear** trend w.r.t. shape dimensions.
  • Decompose the cost model into:
    \[ f_{MK} \cdot f_{\text{spatial}} \]
    • Trainable Micro-Kernel Cost
DietCode: Key Ideas

- Key Idea #2: **Micro-Kernel-based Cost Model**
  - Observation: A cost model trained on one shape can be **inaccurate** on other shapes.
  - Compute throughputs exhibit **predictable linear** trend w.r.t. shape dimensions.
  - Decompose the cost model into:
    \[ f_{\text{MK}} \cdot f_{\text{spatial}} \]
    - Trainable Micro-Kernel Cost
    - Analytical Spatial Generalization Cost (linear function)
Evaluation

Hardware: NVIDIA Tesla T4 GPU

Software: TVM + CUDA + cuDNN

v11.3
v0.8.dev0

v8.3
Evaluation

Performance: 30.5% better than Ansor; 5.3% better than Vendor
Auto-Scheduling Time: 5.6× less than Ansor
Evaluation

What about multiple dynamic axes?

Performance: 24.2% better than Ansor; 15.4% better than Vendor
Summary

• DietCode: An auto-scheduler for dynamic-shape workloads.
• Based on 2 key ideas:
  (1) Shape-Generic Search Space and
  (2) Micro-Kernel-based Cost Model
• Key Features:
  • **Auto-Schedule Once and For All Shapes**: Large reduction in the auto-scheduling time $5.6 \times$ on dynamic-shape workloads.
  • **Better Performance**: Up to 30.5% speedup than Ansor, 15.4% than Vendor.
Last Time

• Why Machine Learning Compilers?
  • State-of-the-Art Machine Learning Frameworks Design, and Flaws:
    1. Vendor libraries not delivering the optimal performance.

• 2 Classes of Machine Learning Compilers:
  • Halide\(^1\)/TVM\(^2\): Easier to write high-performance programs.

\(^1\) J. Ragan-Kelley et al. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. PLDI 2013
\(^2\) T. Chen et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. OSDI 2018
Agenda for Today

• Why Machine Learning Compilers?
  • State-of-the-Art Machine Learning Frameworks Design, and Flaws:
    1. Vendor libraries not delivering the optimal performance.
    2. Distinct representations at different system levels.

• 2 Classes of Machine Learning Compilers:
  • Halide\textsuperscript{[1]}/TVM\textsuperscript{[2]}: Easier to write high-performance programs.
  • MLIR\textsuperscript{[3]}/TensorIR\textsuperscript{[4]}: Unified compiler infrastructure.

\textsuperscript{[1]} J. Ragan-Kelley et al. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. PLDI 2013
\textsuperscript{[2]} T. Chen et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. OSDI 2018
\textsuperscript{[3]} C. Lattner et al. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. CGO 2021
\textsuperscript{[4]} S. Feng et al. Understanding TensorIR: An Abstraction for Tensorized Program Optimization. TVMCon 2021
Why Unity?

• State-of-the-Art Machine Learning Frameworks Design:

<table>
<thead>
<tr>
<th>Python Programming Front-end</th>
<th>define neural networks</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++ Framework Core</td>
<td>optimize graphs; schedule operators; allocate hardware resources ...</td>
</tr>
<tr>
<td>Vendor APIs &amp; Libraries</td>
<td></td>
</tr>
<tr>
<td>Hardware Architectures</td>
<td></td>
</tr>
</tbody>
</table>
TensorFlow System Overview

Distinct representations at different system levels

- TF Graph
- XLA HLO
- TPU IR
- LLVM IR
- Tensor RT
- Several Others
- nGraph
- Many Others
- TF Lite
- NNAPI
- Core ML

input data
TVM System Overview

**Distinct** representations at different system levels

output = relay.dense(data, weight)

```python
for i in range(...):
    for j in range(...):
        for k in range(...):
            Y[i][j] += X[i][k] * W[j][k]
```

```python
primfn(X_1: handle, W_1: handle, Y_1: handle) -> ()
attr [IterVar(blockIdx.x: int32)] "thread_extent" = 144;
allocate(Y.local: Pointer(local float32), [64]);
allocate(X.shared: Pointer(shared float32), [512]);
allocate(W.shared: Pointer(shared float32), [512]);
attr [IterVar(threadIdx.x: int32)] "thread_extent" = 256 {
...
}
```

`__global__ void __launch_bounds__(256)`

`default_function_kernel0(
float* __restrict__ X,
float* __restrict__ W,
float* __restrict__ Y) {
...
}`

Schedule Tree

Hard boundary between representations
• Duplicated Infrastructure
  • Similar utility classes and functions for each IR.

• Premature Lowering
  • Need to lower ALL regions at once.
  • Unidirectional lowering makes it hard to recover high-level IRs.
Multi-Level Intermediate Representation[1]

• *abbrev.* MLIR

• Objective: Unified compiler infrastructure.

• Key Idea: Same infrastructure, distinct **dialects** at different levels.

• Each dialect is a set of operators.

<table>
<thead>
<tr>
<th>IRs</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>TensorFlow Graphs</td>
<td></td>
</tr>
<tr>
<td>Affine For</td>
<td></td>
</tr>
<tr>
<td>LLVM Instructions</td>
<td></td>
</tr>
</tbody>
</table>

TensorIR \cite{1}

- Objective: Unified compiler infrastructure.
- Key Idea: Remove schedule tree representation.

TVM System Overview

\[\text{output} = \text{relay.dense}(\text{data}, \text{weight})\]

```
Y_local = s.cache_write(Y, "local")
I, j, k = tuple(Y_local.op.axis)
...
```

```
primRX_i: handle, W_j: handle, Y_i: handle) -> ()
attr [IterVar[BlockSize.x: int32]] "thread_extent" = 144;
allocate(Y, local: Pointer(local float32), [64]);
allocate(W, shared: Pointer(shared float32), [512]);
allocate(W, shared: Pointer(shared float32), [512]);
attr [IterVar[threadIdx.x: int32]] "thread_extent" = 256 { 
} ...
```

```
__global__ void __launch_bounds__(256)
default_function_kernel0(
    float* __restrict__ X,
    float* __restrict__ W,
    float* __restrict__ Y)
{ 
} ...
```

TensorIR

Schedule Tree

```python
s = te.create_schedule(Y)
Y_local = s.cache_write([Y], "local")
i, j, k = tuple(Y_local.op.axis)
i_o_i, i_i = s[Y_local].split(i, factor=8)
i_o_o_i, i_o_i = s[Y_local].split(i_o_i, factor=2)
i_o_o_o, i_o_o_i = s[Y_local].split(i_o_o_i, factor=1)
...```

TensorIR

```python
def MatMul(x, w, y):
    X = tir.match_buffer(x, (1024, 1024), "float32")
    W = tir.match_buffer(w, (1024, 1024), "float32")
    Y = tir.match_buffer(y, (1024, 1024), "float32")
    reducer = tir.comm_reducer(lambda x, y: x + y, tir.float32(0))

    with tir.block([1024, 1024, tir.reduce_axis(0, 1024)], "Y") as [vi, vj, vk]:
        reducer.step(Y[vi, vj], X[vi, vk] * W[vk, vj])
```

(-) Not Intuitive
(-) Hard to support even existing hardware primitives.
TensorIR[1]

Schedule Tree

s = te.create_schedule(Y)
Y_local = s.cache_write([Y], "local")
i, j, k = tuple(Y_local.op.axis)
i_o_i, i_i = s[Y_local].split(i, factor=8)
i_o_o_i, i_o_i = s[Y_local].split(i_o_i, factor=2)
i_o_o_o, i_o_o_i = s[Y_local].split(i_o_o_i, factor=1)
...

TensorIR

def MatMul(x, w, y):
    X = tir.match_buffer(x, [1024, 1024])
    W = tir.match_buffer(w, [1024, 1024])
    Y = tir.match_buffer(y, [1024, 1024])
    reducer = tir.comm_reducer(lambda a, b: a + b, tir.float32(0))
    for i0_outer, i1_outer, i2_outer, i2_inner,
        i0_inner, i1_inner in tir.grid(32, 32, 256, 4, 32, 32):
        with tir.block([1024, 1024, tir.reduce_axis(0, 1024)], "C") as [vi, vj, vk]:
            tir.bind(vi, ((i0_outer*32) + i0_inner))
            tir.bind(vj, ((i1_outer*32) + i1_inner))
            tir.bind(vk, ((i2_outer*4) + i2_inner))
            reducer.step(Y[vi, vj], (X[vi, vk]*W[vk, vj]))

(-) Not Intuitive
(-) Hard to support even existing hardware primitives.
(+) Expressive
(+) Hierarchical Block structure makes it easier to map to hardware primitives.
Concluding Remarks

• Why Machine Learning Compilers?
  • State-of-the-Art Machine Learning Frameworks Design, and Flaws:
    1. Vendor libraries not delivering the optimal performance.
    2. Distinct representations at different system levels.

• 2 Classes of Machine Learning Compilers:
  • Halide\textsuperscript{[1]}/TVM\textsuperscript{[2]}: Easier to write high-performance programs.
  • MLIR\textsuperscript{[3]}/TensorIR\textsuperscript{[4]}: Unified compiler infrastructure.

\textsuperscript{[1]} J. Ragan-Kelley et al. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. PLDI 2013
\textsuperscript{[2]} T. Chen et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. OSDI 2018

\textsuperscript{[3]} C. Lattner et al. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. CGO 2021
\textsuperscript{[4]} S. Feng et al. Understanding TensorIR: An Abstraction for Tensorized Program Optimization. TVMCon 2021
Python Programming Front-end

C++ Framework Core

Vendor APIs & Libraries

Hardware Architectures

define neural networks

optimize graphs; schedule operators; allocate hardware resources ...
$X: [{\color{red} \mathbf{129}}, K], \quad W: [{\color{red} \mathbf{129}}, K]$

\begin{align*}
y &= xw^\top \quad \text{quad} \\
\begin{cases}
  x: [32, *], w: [32, *] \\
  x: [64, *], w: [64, *] \\
  x: [128, *], w: [128, *]
\end{cases}
\end{align*}
Current TensorFlow System

- Task #1: Eliminate dead graph nodes.
- Task #2: Eliminate dead C++ code.

Really different?
TVM System Overview

```
s = create_schedule(Y.op)
Y_local = s.cache_write([Y], "local")
i, j, k = tuple(Y_local.op.axis)
...
output = relay.dense(data, weight)
for i in range(...):
    for j in range(...):
        for k in range(...):
            Y[i][j] += X[i][k] * W[j][k]
primfn(X_1: handle, W_1: handle, Y_1: handle) -> ()
    attr [IterVar(blockIdx.x: int32)] "thread_extent" = 144;
    allocate(Y.local: Pointer(local float32), [64]);
    allocate(X.shared: Pointer(shared float32), [512]);
    allocate(W.shared: Pointer(shared float32), [512]);
    attr [IterVar(threadIdx.x: int32)] "thread_extent" = 256 {
        ...
    }
    __global__ void __launch_bounds__(256)
    default_function_kernel0{
        float* __restrict__ X,
        float* __restrict__ W,
        float* __restrict__ Y) {
            ...
        }
```