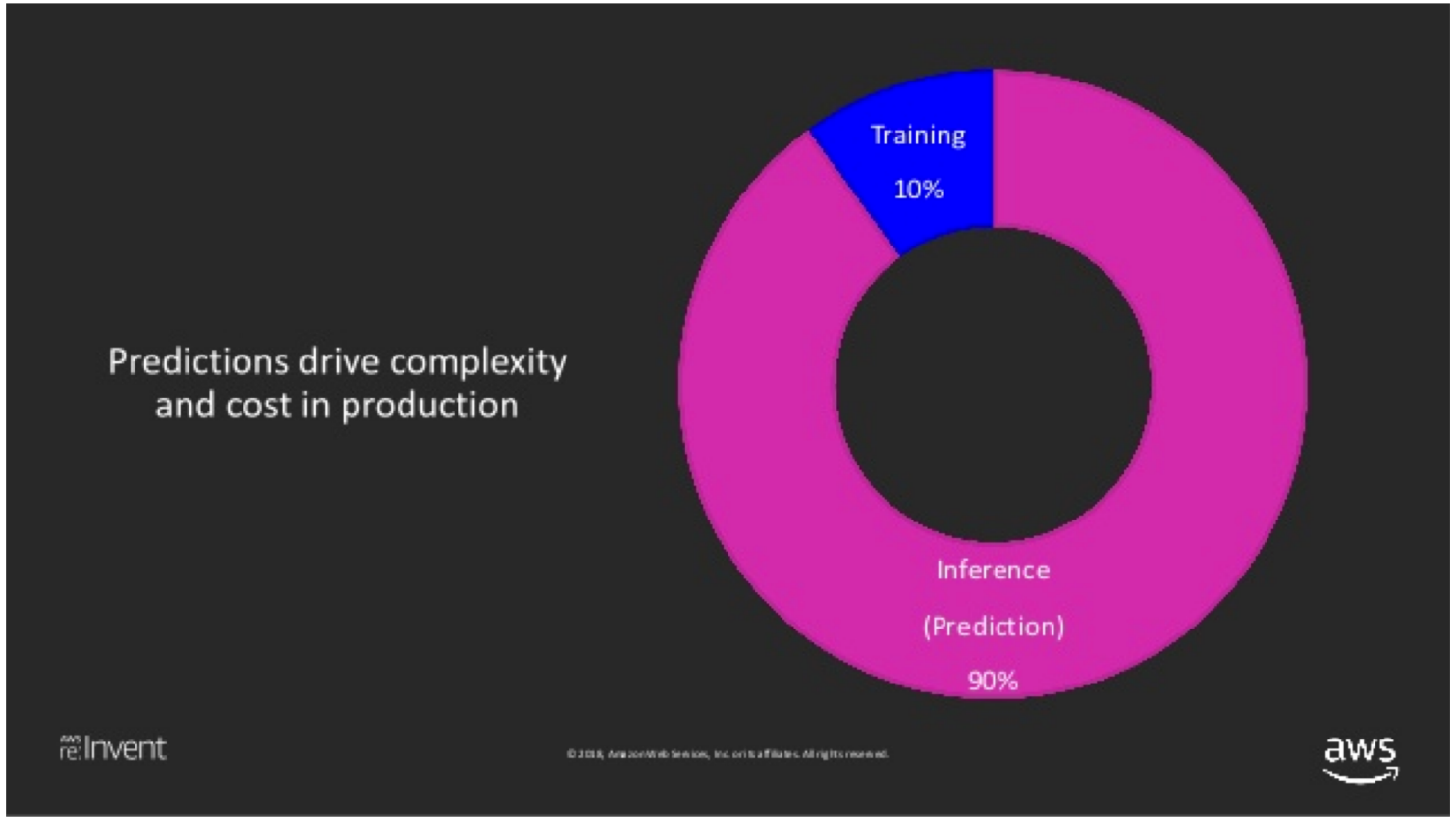
Deep Learning Compilers

- Hardware and Software Challenges for DL inference
- Halide the precursor of DL compilers
- Deep Learning Compilers
  - TVM
  - Tensor Comprehensions

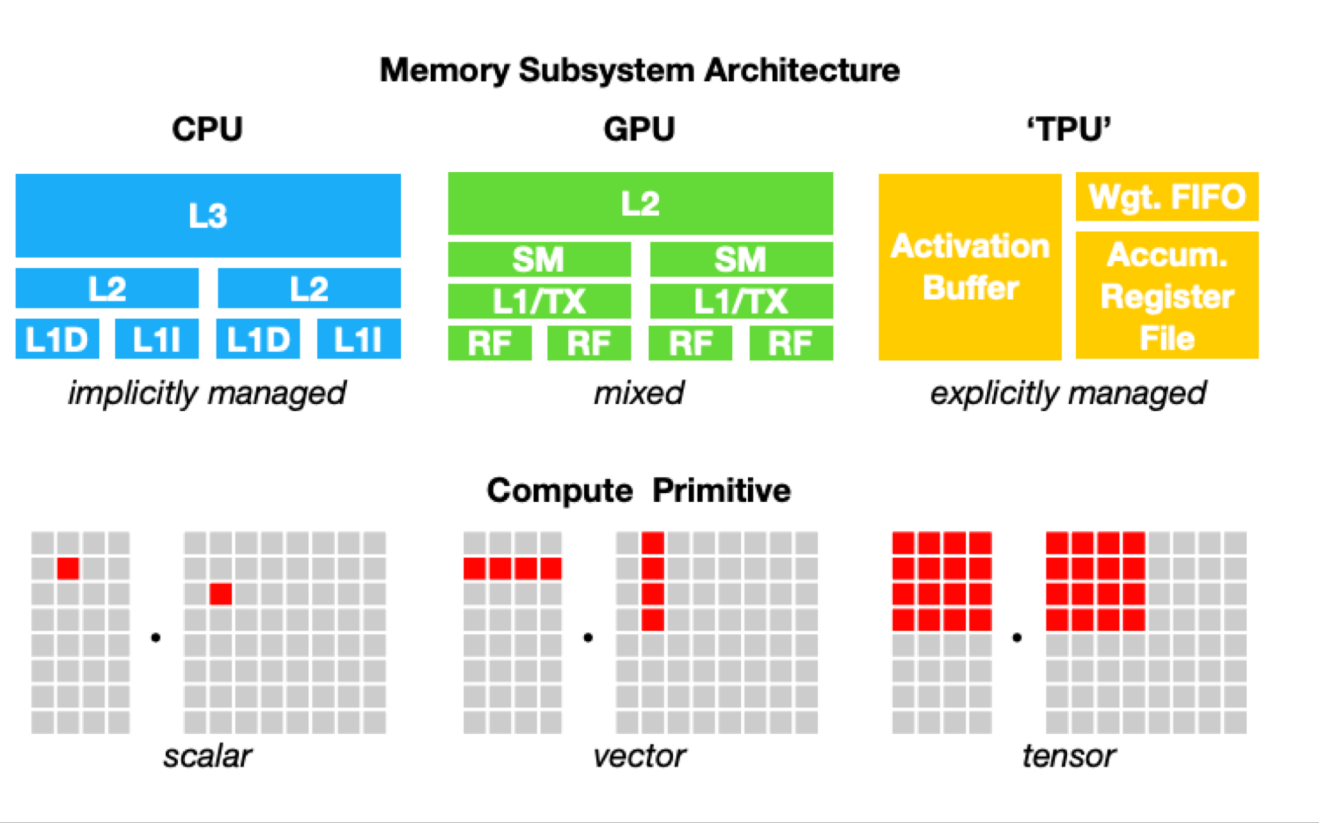
Simon Mo
AI-Sys Seminar

Note

- This talk is focus on *inference* stage of deep learning workload
- But* these DL compilers should also applied to *training*



Hardware for Deep Learning



- **Heterogenous hardware**:
  - Need to optimize workload for different hardware.
- **Layered Memory Hierarchy**:
  - Complex scheduling space
- **Parallel Compute Primitives**
  - SIMD
  - SIMT
  - Intrinsics

Software for Deep Learning

Declarative DAG of
High-Level Operators

Early-Binding to Compute

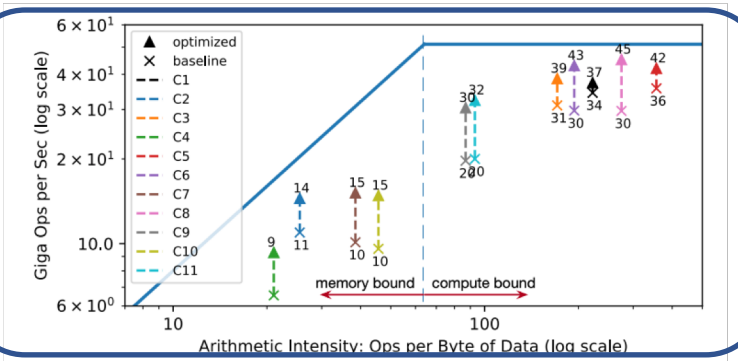
Mixed Memory
& Compute Requirement

Conv2d

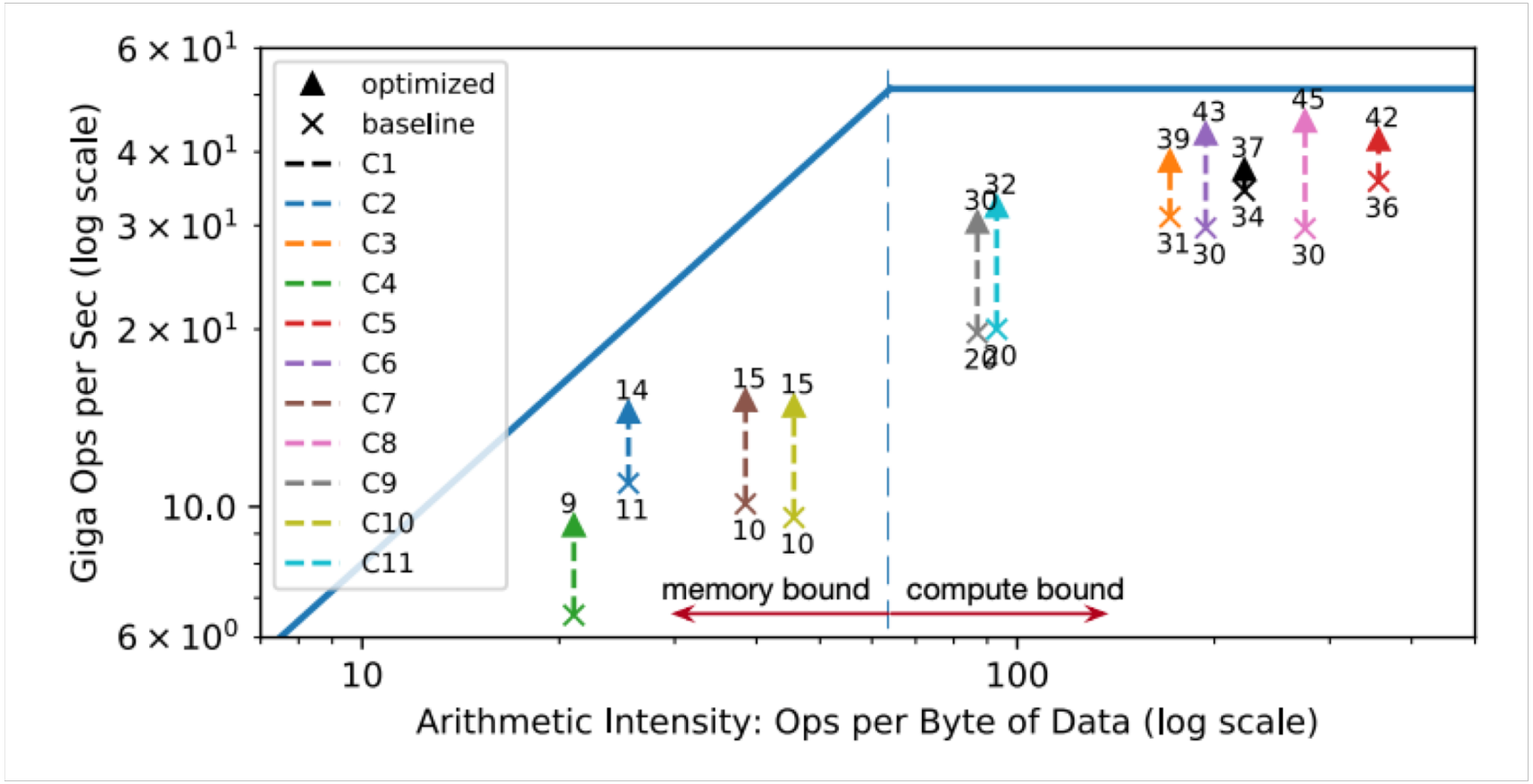
CLASS `torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True)` [SOURCE]

Applies a 2D convolution over an input signal composed of several input planes.

volta_scudnn_winograd_128x128_ldg1_ldg4_relu_tile148t_nt_v1



Software for Deep Learning



Mixed Memory
& Compute Requirement

Reality -> Problem Statement

Reality

(1) Everyone knows how Convolution works. Few can implement fast convolution in CUDA.

(2) DL Framework *depends* on hand-tuned kernel implementation for specific hardware, by experts.

(3) Researcher can't create efficient new operators. Whiteboard -\-> Physical Operator.

Problem Statement

(1) For a given operator, express it in a simple language that **abstract away the complexity of hardware**.

(2) For a given operator, we want to automatically optimize it for **different hardware**.

(3) For any new operator, we want to easily find an efficient implementation **without thinking about hardware** at all.

Problem Statement

(1) Abstract away the complexity of hardware.

(2) Automatically optimize for different hardware.

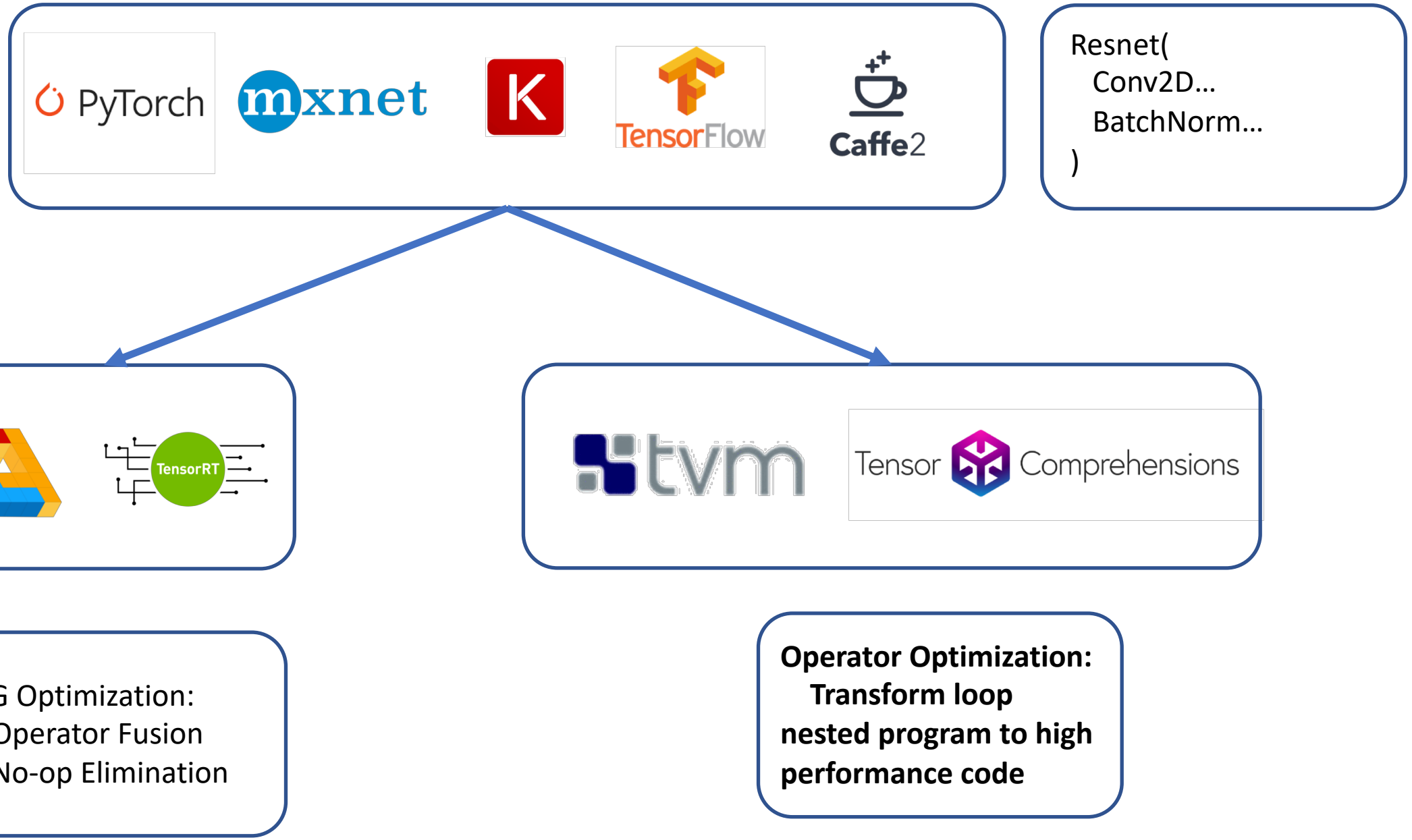
(3) New operator without thinking about hardware at all.

System Proposed

Halide

TVM

Tensor Comprehension (TC)

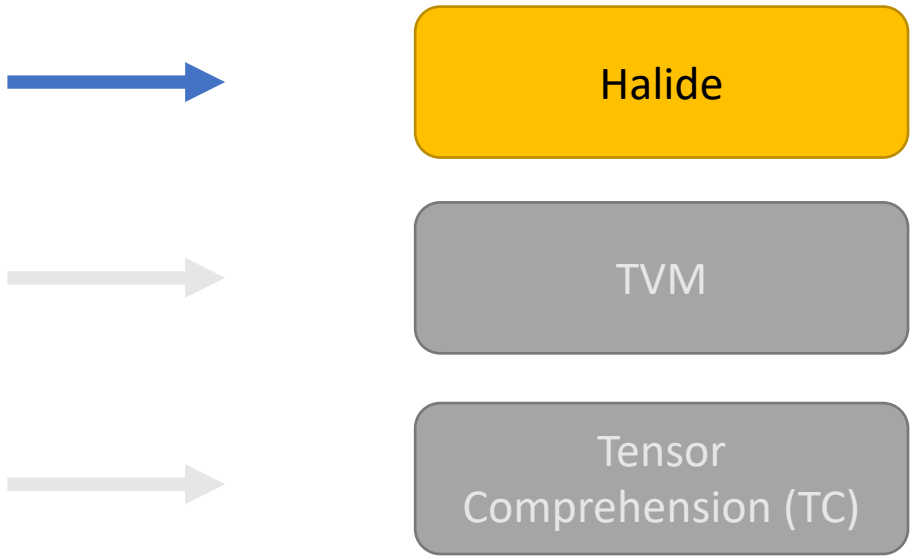

PyTorch
mxnet
TensorFlow
Caffe2
Resnet(
Conv2D…
BatchNorm…
)
TensorRT
tvm
Tensor Comprehensions
DAG Optimization:
- Operator Fusion
- No-op Elimination
Operator Optimization:
Transform loop nested program to high performance code

Problem Statement

(1) Abstract away the complexity of hardware.

(2) Automatically optimize for different hardware.

(3) New operator without thinking about hardware at all.



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

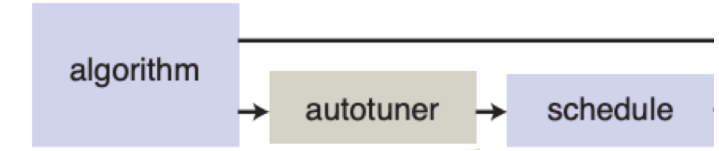
```
blurx: split x by 4 → x_o, x_i
       vectorize: x_i
       store at out.x_o
       compute at out.y_i

out:   split x by 4 → x_o, x_i
       split y by 4 → 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

```
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

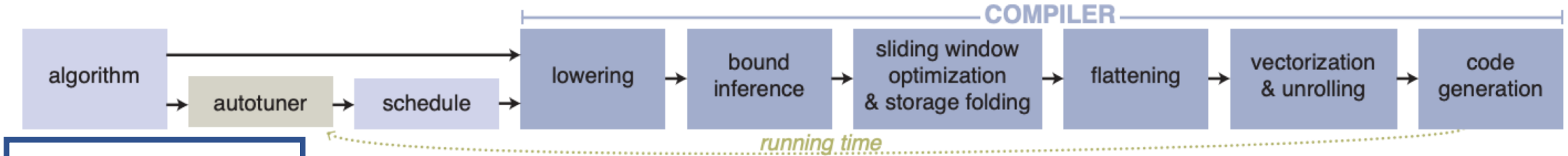
```
blurx: split x by 4 → x_o, x_i
       vectorize: x_i
       store at out.x_0
       compute at out.y_i

out:   split x by 4 → x_o, x_i
       split y by 4 → 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

```
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

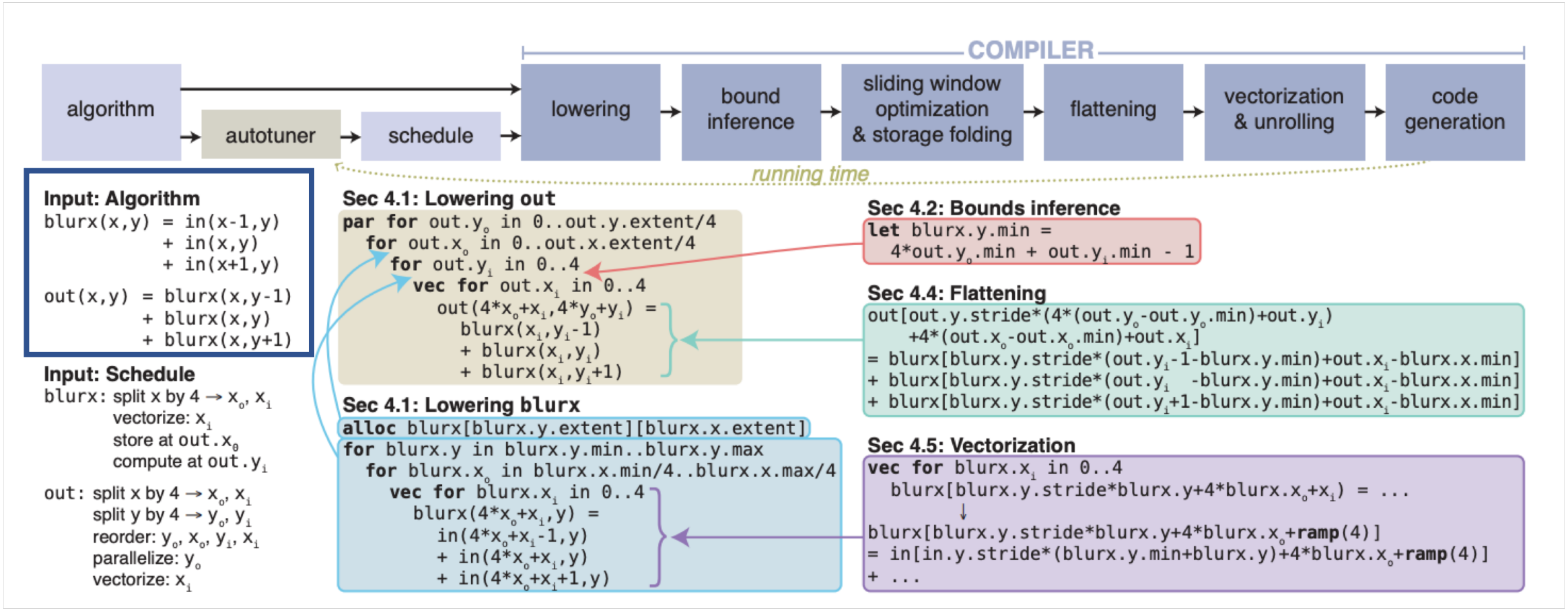
```
blurx: split x by 4 → x_o, x_i
       vectorize: x_i
       store at out.x_0
       compute at out.y_i

out:   split x by 4 → x_o, x_i
       split y by 4 → 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?



Halide DSL

```
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

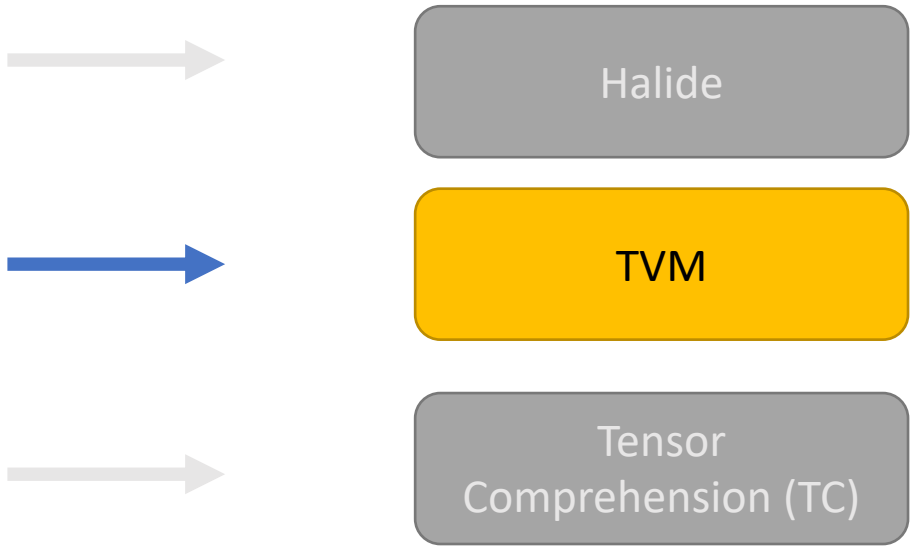
Problem Statement

(1) Abstract away the complexity of hardware.

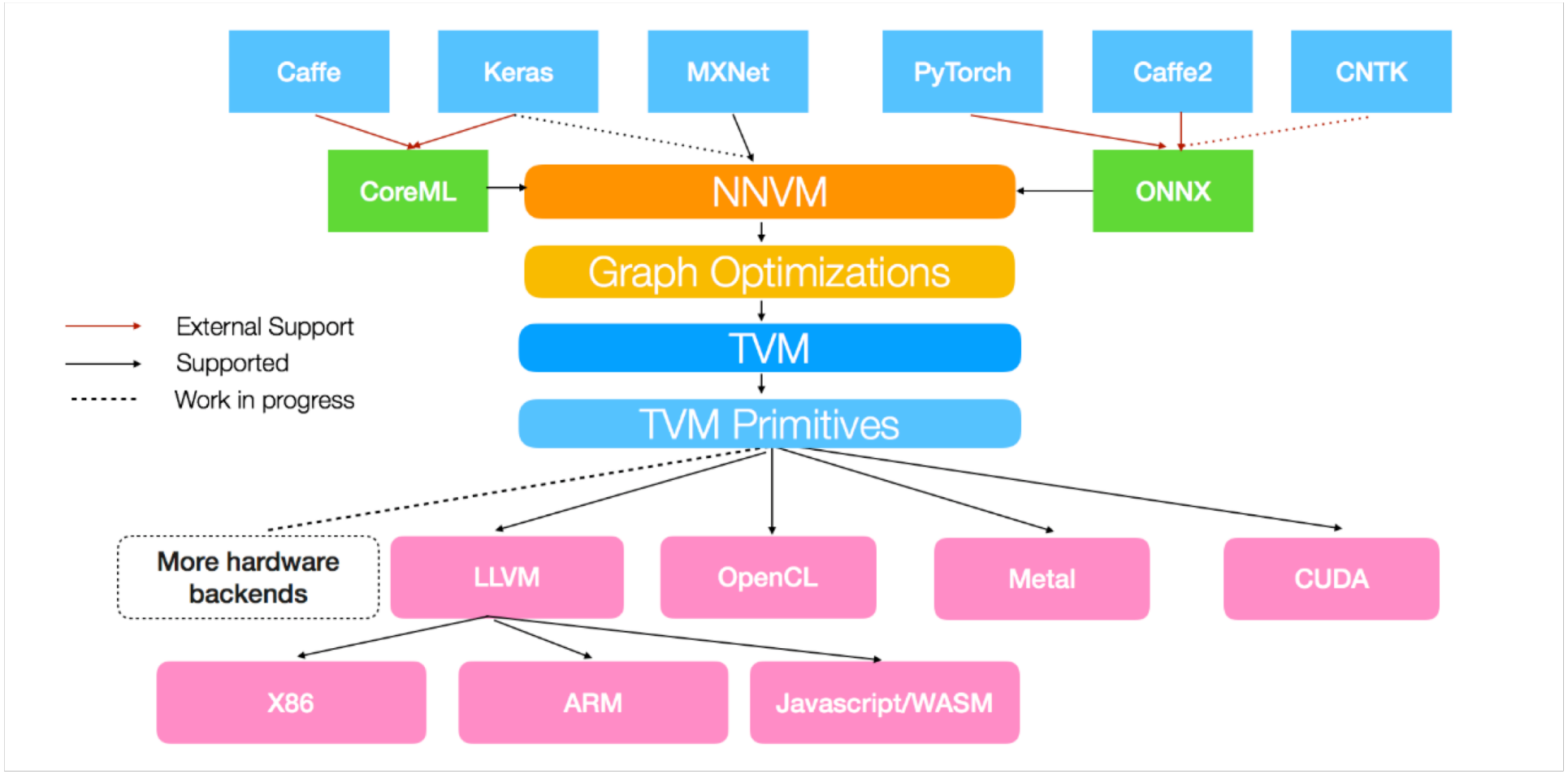
(2) Automatically optimize for different hardware.

(3) New operator without thinking about hardware at all.

System Proposed



TVM: An automated End-to-End Optimizing Compiler for Deep Learning



TVM DSL

```
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]
```

Very similar to Halide

- Specify the algorithm
- Specify the schdule

TVM DSL

+ 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]
```

Use *tile, split, reduce, etc* to transform the loop nested program into a complex schedules. To

- Optimize data locality
- Minimize memory conflict
- Optimize for device cache
- Optimize for latency hiding

TVM DSL

+ 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)
```

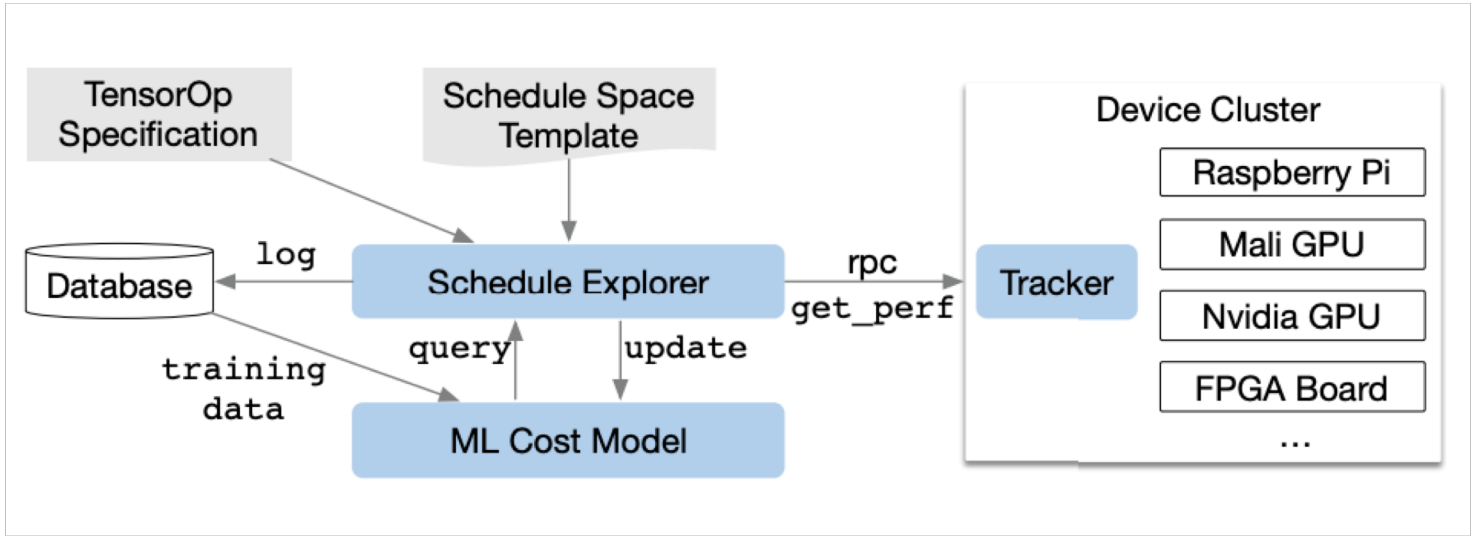
- Note that Halide doesn't have these

(1) TVM allows read and write to special memory scope

(2) TVM can hook into hardware instructions

(3) TVM can optimize for pipeline parallelism via reordering

TVM's **DSL + Autotuner** enables it to target many devices



In TVM, you can templatize your schedule and let autotuner find the optimal configuration for a group of devices

TVM produces high performance operators for different hardware

GPU

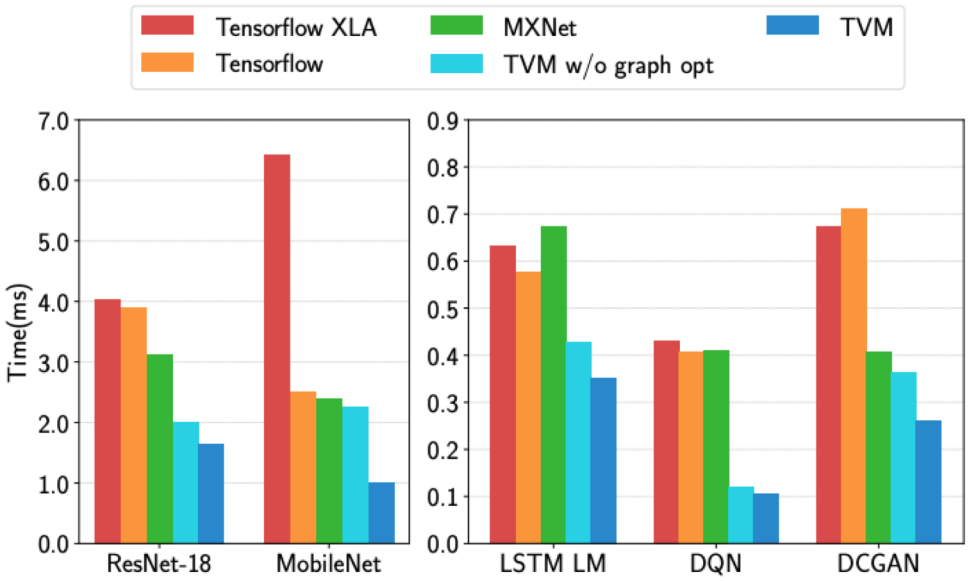


Figure 14: GPU end-to-end evaluation for TVM, MXNet, Tensorflow, and Tensorflow XLA. Tested on the NVIDIA Titan X.

TVM produces high performance operators for different hardware

Embedded CPU

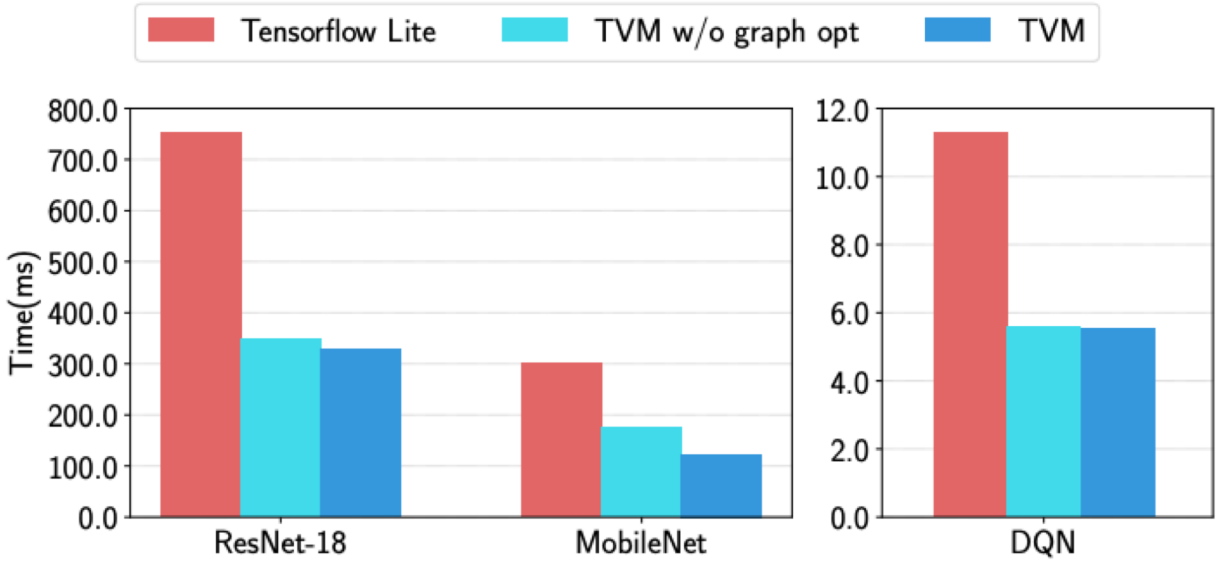


Figure 16: ARM A53 end-to-end evaluation of TVM and TFLite.

TVM produces high performance operators for different hardware

FPGA

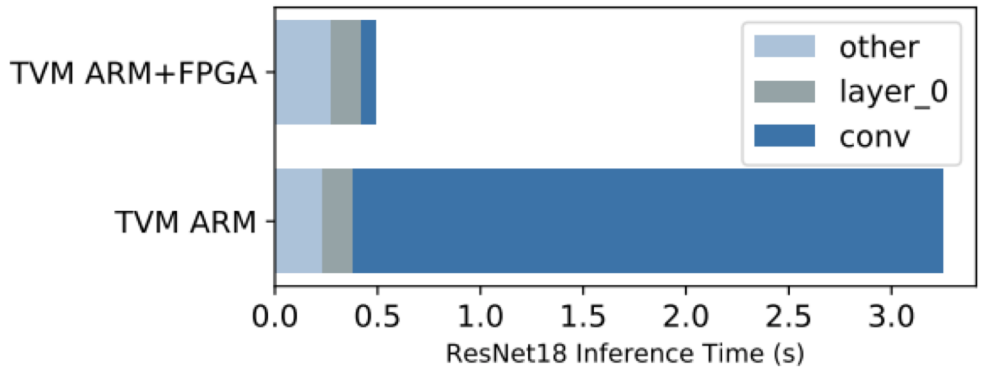


Figure 21: We offloaded convolutions in the ResNet workload to an FPGA-based accelerator. The grayed-out bars correspond to layers that could not be accelerated by the FPGA and therefore had to run on the CPU. The FPGA provided a 40x acceleration on offloaded convolution layers over the Cortex A9.

TVM's Auto-tuner uses ML techniques

```
for y in range(8):
  for x in range(8):
    C[y][x]=0
    for k in range(8):
      C[y][x]+=A[k][y]*B[k][x]
```

(a) Low level AST

touched memory

	C	A	B
y	64	64	64
x	8	8	64
k	1	8	8

outer loop length

y	1
x	8
k	64

(b) Loop context vectors

- Parametrized the AST
- Use Gradient Boost Tree (GBT) to optimize a "rank loss" to predict the relative order of program runtime

encourages the model to predict cost accurately. On the other hand, as we care only about the relative order of program run times rather than their absolute values in the selection process, we can instead use the following rank loss function [6]:

$$\sum_{i,j} \log(1 + e^{-\operatorname{sign}(c_i - c_j)(\hat{f}(x_i) - \hat{f}(x_j))}). \quad (2)$$

TVM's Auto-tuner uses ML techniques

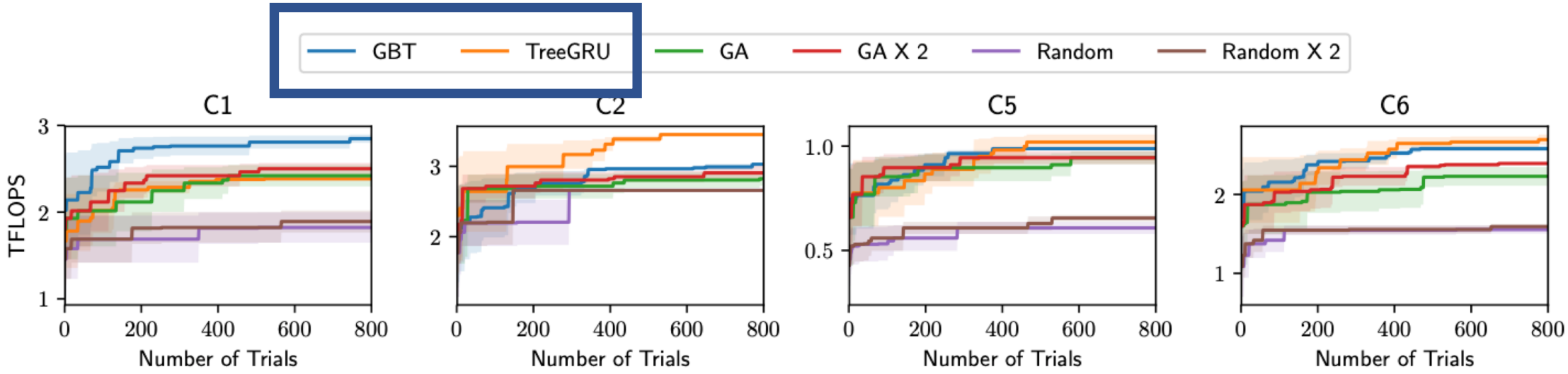


Figure 4: Statistical cost model vs. genetic algorithm (GA) and random search (Random) evaluated on NVIDIA TITAN X. 'Number of trials' corresponds to number of evaluations on the real hardware. We also conducted two hardware evaluations per trial in Random $\times 2$ and GA $\times 2$. Both the GBT- and TreeGRU-based models converged faster and achieved better results than the black-box baselines.

TVM: Summary

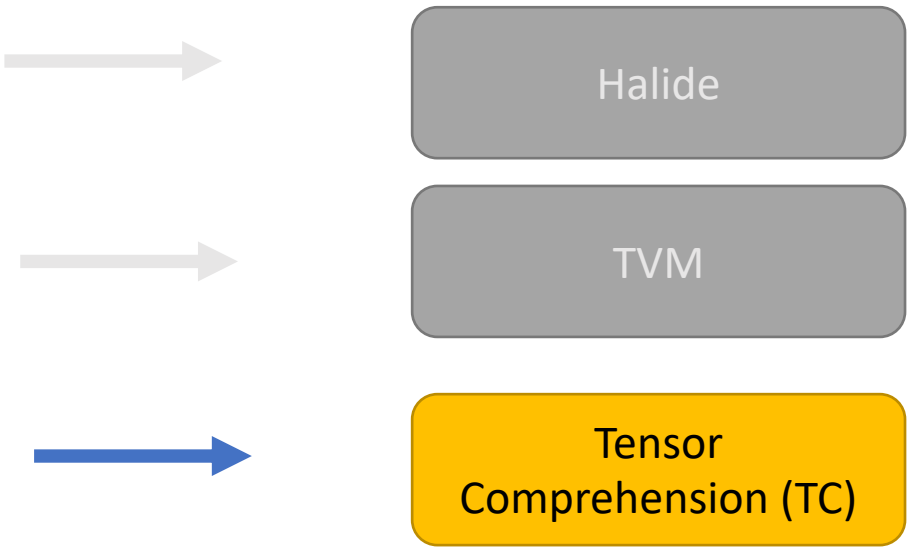
- What is the problem that is being solved?
  - Optimize operator for many different devices
- What are the metrics of success?
  - Performance improvement
- What are the key innovations over prior work?
  - Versatile DSL
  - Powerful Auto-tuner
- What are the key results?
  - Significant speedup across different devices
- What are some of the limitations and how might this work be improved?
  - Auto-tuning take forever, cannot be JIT compiled
  - Extremely large scheduling space, maybe RL based, learned cost model?
- How might this work have long term impact?
  - In production use.

Problem Statement

(1) Abstract away the complexity of hardware.

(2) Automatically optimize for different hardware.

(3) New operator without thinking about hardware at all.



TC: From whiteboard to machine code

TC's DSL is extremely simple. Algorithm only.

```
def sgemm(float a, float b,
          float(N,M) A, float(M,K) B) → (C) {
  C(i,j)  = b * C(i,j)             # initialization
  C(i,j) += a * A(i,k) * B(k,j)    # accumulation
}
```

Figure 1: Tensor Comprehension for the sgemm BLAS

TC resembles the **whiteboard mathematical model** of a deep neural network and makes it easy to reason about, communicate, and to manually alter the computations and storage/computation tradeoffs.

TC: Targeted Audience

- Rapid prototyping new operators for researchers
- Provide comparable performance than manual tuning
- It's embedded inside PyTorch, Caffe2

```
import tc
ee = tc.ExecutionEngine()
ee.define("""
  def mm(float(M,K) A,
         float(K,N) B) -> (C) {
    C(m,n) +=! A(m,kk) * B(kk,n)
  }
""")
```

Figure 11: Build execution engine

```
import torch
A = torch.randn(3,4)
B = torch.randn(4,5)
C = ee.mm(A, B)
```

Figure 12: JIT compile, tune, or hit the compilation cache, then run

TC: Polyhedral Optimization Replaces User Defined Schedule

TC: Polyhedral Optimization Replaces User Defined Schedule

Polyhedral scheduling, optimizes for (outer) loop parallelism
and locality + hand tuned affine scheduling heuristic

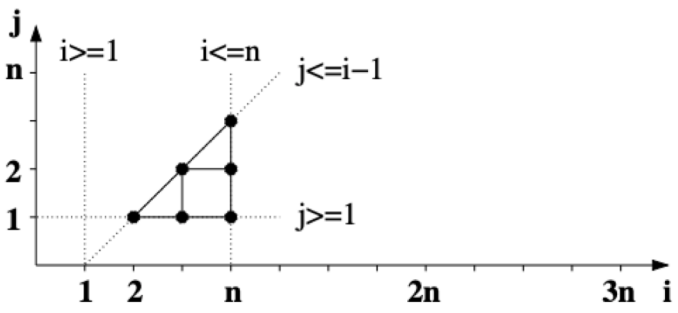

```
def sgemm(float a, float b,
          float(N,M) A, float(M,K) B) → (C) {
  C(i,j)  = b * C(i,j)              # initialization
  C(i,j) += a * A(i,k) * B(k,j)     # accumulation
}
```

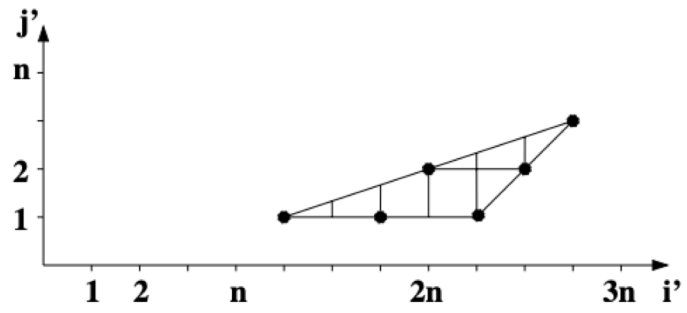
$$
\begin{aligned}
&\text{Domain} \left[\begin{array}{l} \{\mathtt{S}(i,j) \quad | 0 \le i < N \wedge 0 \le j < K\} \\ \{\mathtt{T}(i,j,k) \,|\, 0 \le i < N \wedge 0 \le j < K \wedge 0 \le k < M\} \end{array} \right. \\
&\text{Band} \left[\begin{array}{l} \{\mathtt{S}(i,j) \quad \to (32\lfloor i/32 \rfloor, 32\lfloor j/32 \rfloor)\} \\ \{\mathtt{T}(i,j,k) \quad \to (32\lfloor i/32 \rfloor, 32\lfloor j/32 \rfloor)\} \end{array} \right. \\
&\quad \text{Sequence} \\
&\quad\quad \text{Filter}\{\mathtt{S}(i,j)\} \\
&\quad\quad\quad \text{Band}\{\mathtt{S}(i,j) \to (i \bmod 32, j \bmod 32)\} \\
&\quad\quad \text{Filter}\{\mathtt{T}(i,j,k)\} \\
&\quad\quad\quad \text{Band}\{\mathtt{T}(i,j,k) \to (32\lfloor k/32 \rfloor)\} \\
&\quad\quad\quad\quad \text{Band}\{\mathtt{T}(i,j,k) \to (k \bmod 32)\} \\
&\quad\quad\quad\quad\quad \text{Band}\{\mathtt{T}(i,j,k0 \to (i \bmod 32, j \bmod 32)\}
\end{aligned}
$$

(d) fused, tiled and sunk

```
int b0 = blockIdx.x; int b1 = blockIdx.y; int b2 = blockIdx.z;
int t0 = threadIdx.x; int t1 = threadIdx.y; int t2 = threadIdx.z;
float (*O)[512][(((7 - 3) + 1))][(((7 - 3) + 1))] = reinterpret_cast<float (*)[512]
const float (*I)[512][7][7] = reinterpret_cast<const float (*)[512][7][7]>(pI);
const float (*W1)[512][3][3] = reinterpret_cast<const float (*)[512][3][3]>(pW1
__shared__ float _O_0[1][64][5][5];
__shared__ float _I_0[1][16][7][7];
__shared__ float _W1_0[64][16][3][3];
for (int c3 = 0; c3 <= 511; c3 += 16) {
  __syncthreads();
  _O_0[0][t2][t1][t0] = O[0][t2 + 64 * b0][t1][t0];
  if (t0 <= 1) {
    _O_0[0][t2][t1][t0 + 3] = O[0][t2 + 64 * b0][t1][t0 + 3];
  }
  if (t1 <= 1) {
    _O_0[0][t2][t1 + 3][t0] = O[0][t2 + 64 * b0][t1 + 3][t0];
    if (t0 <= 1) {
      _O_0[0][t2][t1 + 3][t0 + 3] = O[0][t2 + 64 * b0][t1 + 3][t0 + 3];
    }
  }
  _O_0[0][t2 + 32][t1][t0] = O[0][t2 + 64 * b0 + 32][t1][t0];
  if (t0 <= 1) {
    _O_0[0][t2 + 32][t1][t0 + 3] = O[0][t2 + 64 * b0 + 32][t1][t0 + 3];
  }
  if (t1 <= 1) {
```

TC: Polyhedral Transformation + Mapping





$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 1 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \geq \begin{pmatrix} 1 \\ -n \\ 1 \\ 1 \end{pmatrix}$$

(a) original polyhedron $A\vec{x} \geq \vec{c}$

$$\begin{bmatrix} 1/2 & -1/2 \\ -1/2 & 1/2 \\ 0 & 1 \\ 1/2 & -3/2 \end{bmatrix} \begin{pmatrix} i' \\ j' \end{pmatrix} \geq \begin{pmatrix} 1 \\ -n \\ 1 \\ 1 \end{pmatrix}$$

(b) usual transformation $(AT^{-1})\vec{y} \geq \vec{c}$

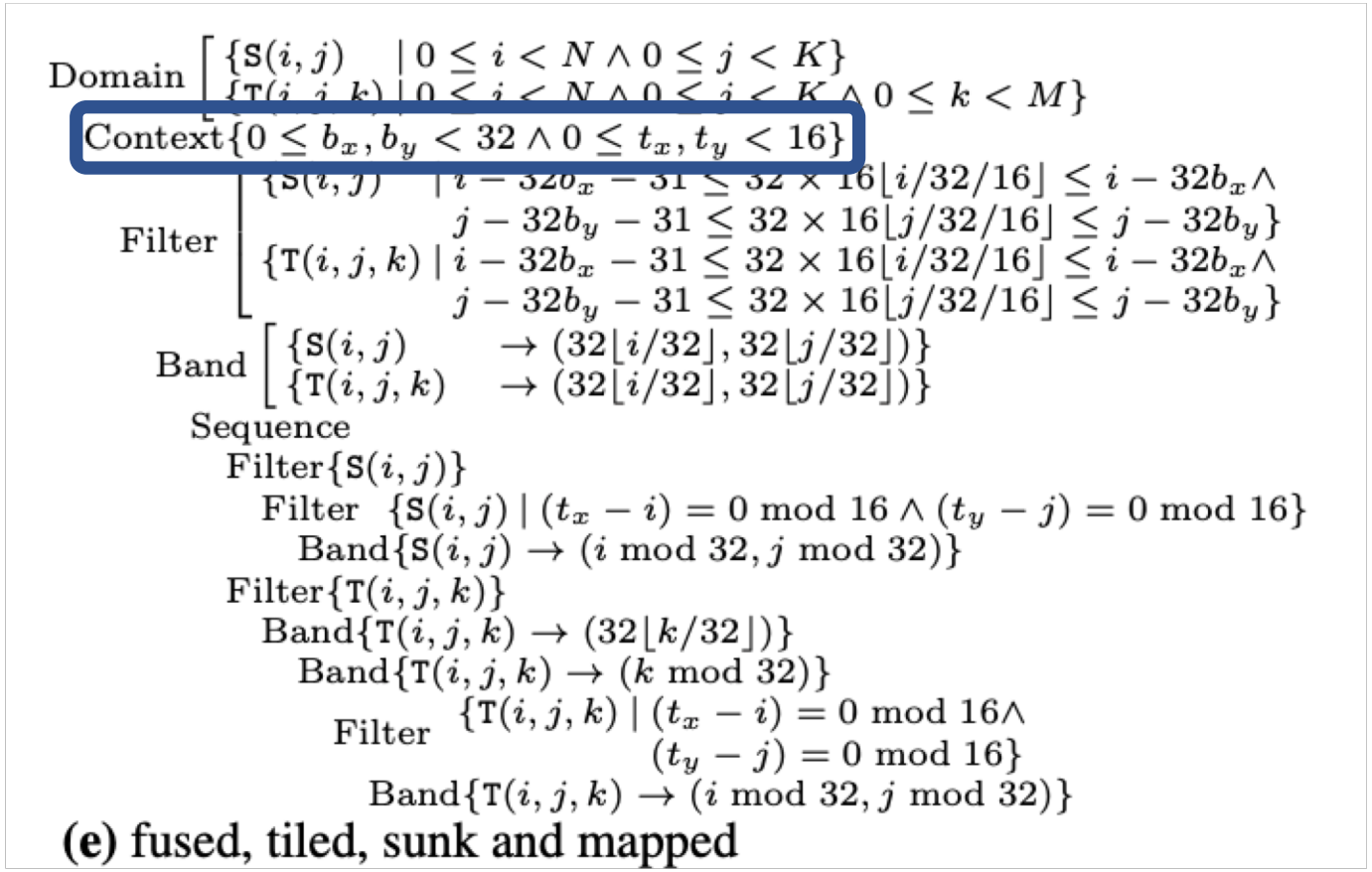
- Given a program in loop nested form, automatically, by formulating the problem as *integer linear program*, optimize for outer loop parallelism and data locality.

Original Program

Transformed Program

TC: Polyhedral Transformation + **Mapping**

Map GPU compute and memory resources to the newly transformed program

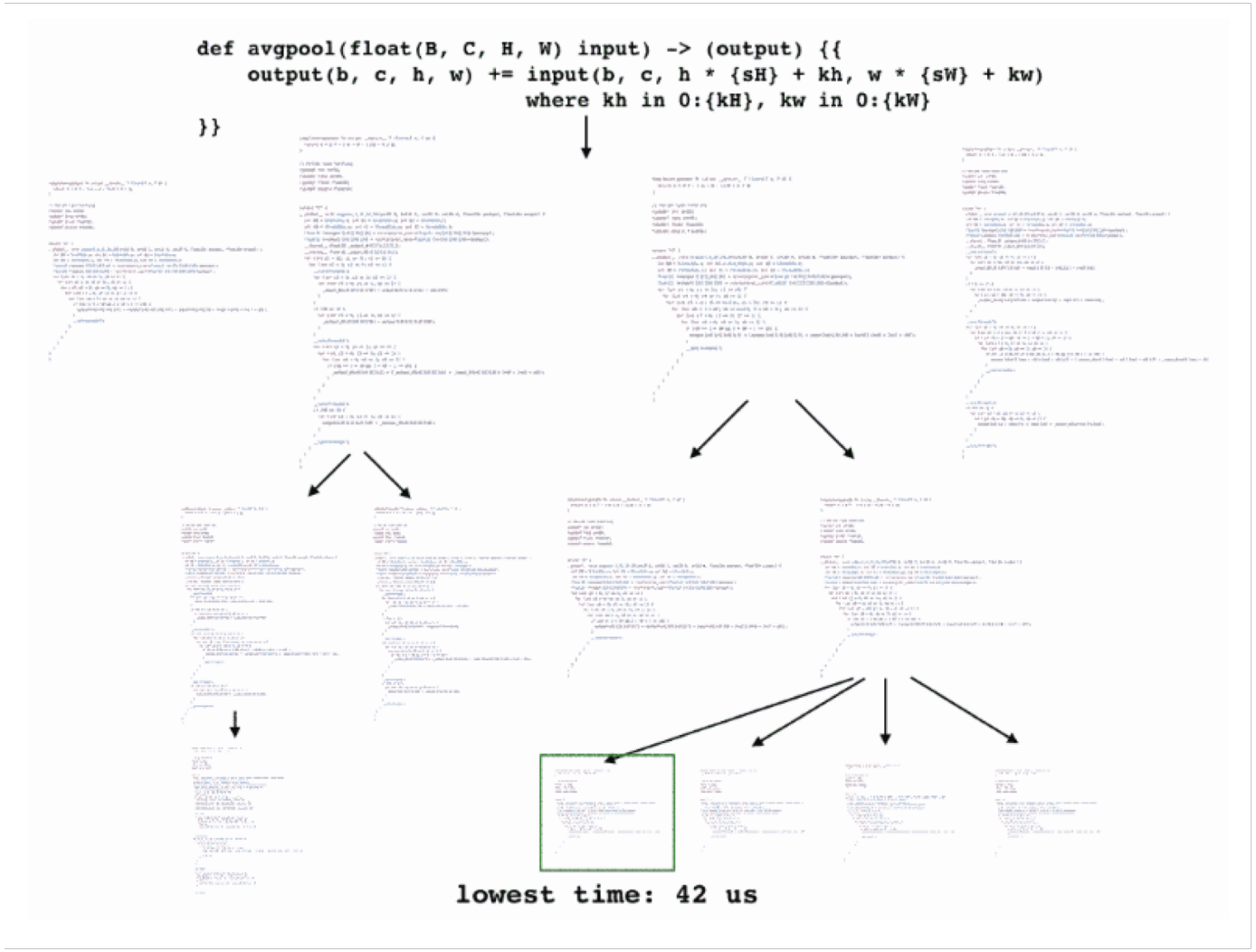


(e) fused, tiled, sunk and mapped

TC: Mapping requires hyperparameters

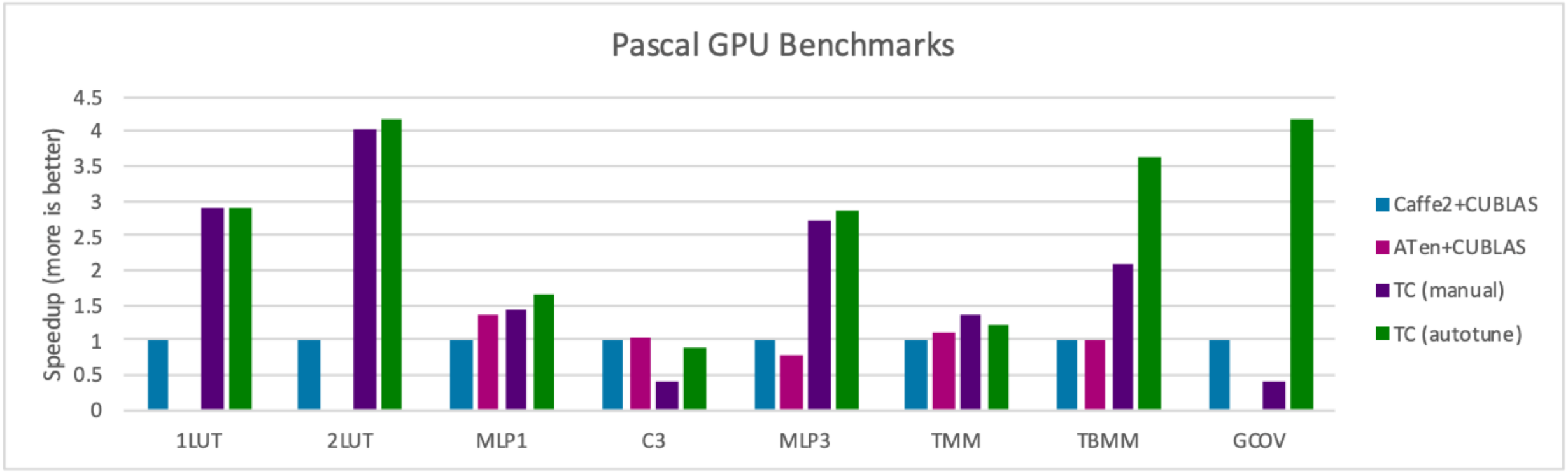
```
tc::IslKernelOptions::makeDefaultMappingOptions()
   .tile({4, 32})
   .mapToThreads({1, 32})
   .mapToBlocks({100, 100})
   .useSharedMemory(true)
   .usePrivateMemory(true)
   .unrollCopyShared(true)
   .unrollGpuTile(true)
   .unroll(1024)
```

TC: Use Genetic Algorithm to Find Best Config



1. three parents are selected probabilistically based on their fitness, the higher the fitness the higher the selection chance;
2. each “gene”, which corresponds to one tuning parameter, of the new candidate is randomly selected from the parents.

TC: Performance



TC: Summary

1. What is the problem that is being solved?
    1. Create optimized operator from simple tensor operation
2. What are the metrics of success?
    1. Speedup and ease of use
3. What are the key innovations over prior work?
    1. Use Polyhedral optimization techniques to automatically come up with the schedule
4. What are the key results?
    1. Up to 4x speedup in certain kernel
5. What are some of the limitations and how might this work be improved?
    1. Only for tensor operation, one tensor operation per kernel.
    2. E.g. We can’t express Winograd convolution
6. How might this work have long term impact?
    1. Shown the potential of polyhedral optimization

Discussion

Architecture:

- “GPU is too slow for deep neural nets, we should build FPGA”
- Is TPU a step in the right direction? Considering low utilization, low memory throughput, etc.

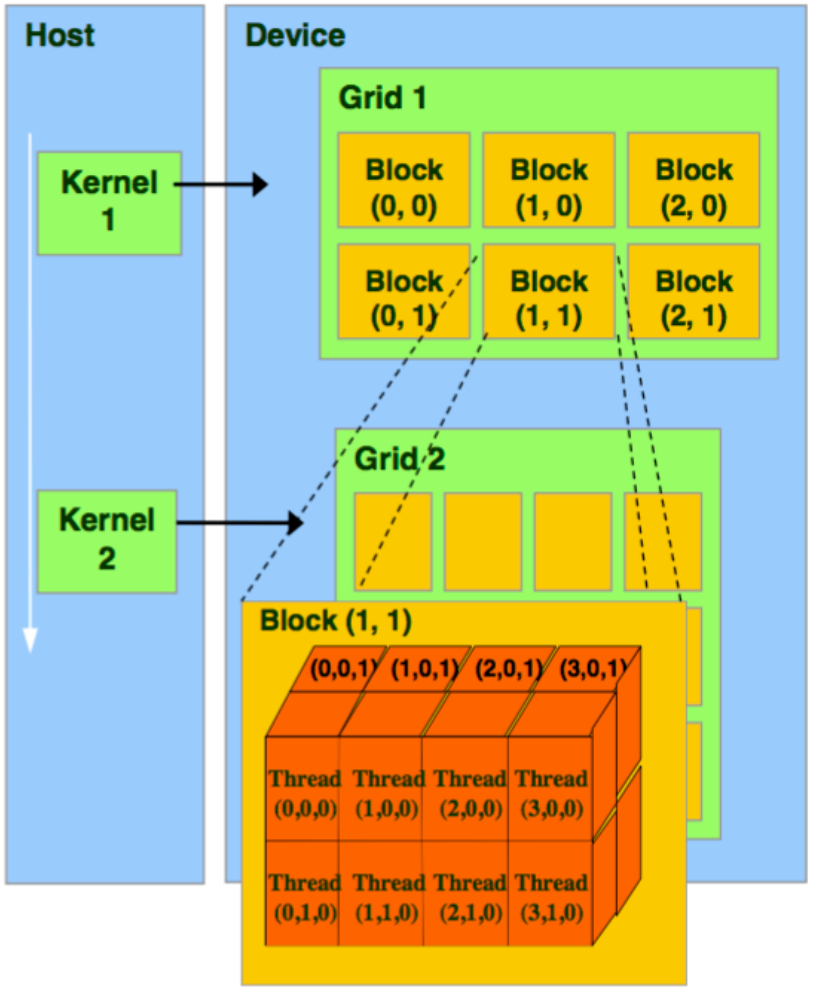
Machine Learning:

- TVM only considered very basic ML techniques, any chance for RL?
- Do you think these kind of problem (intractable scheduling space) is well suited for machine learning?

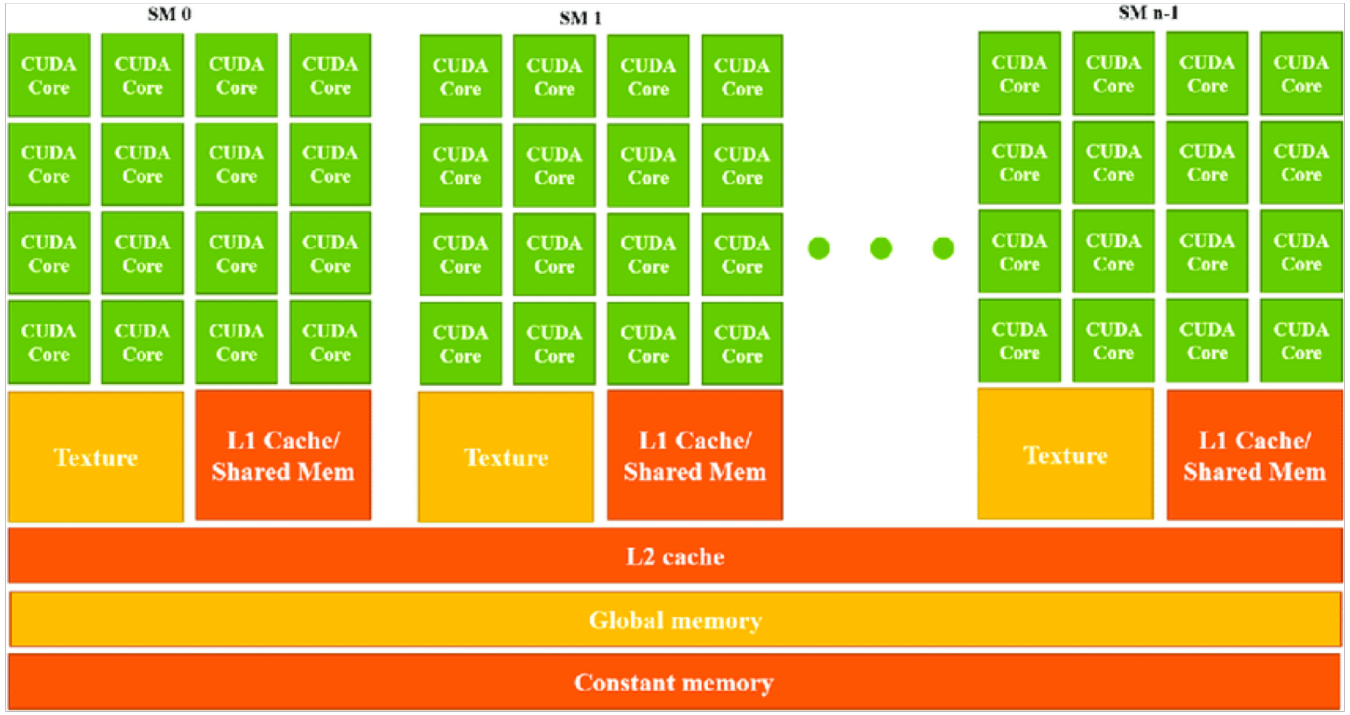
Your Questions…

Backup Slides

Quick Background on GPU

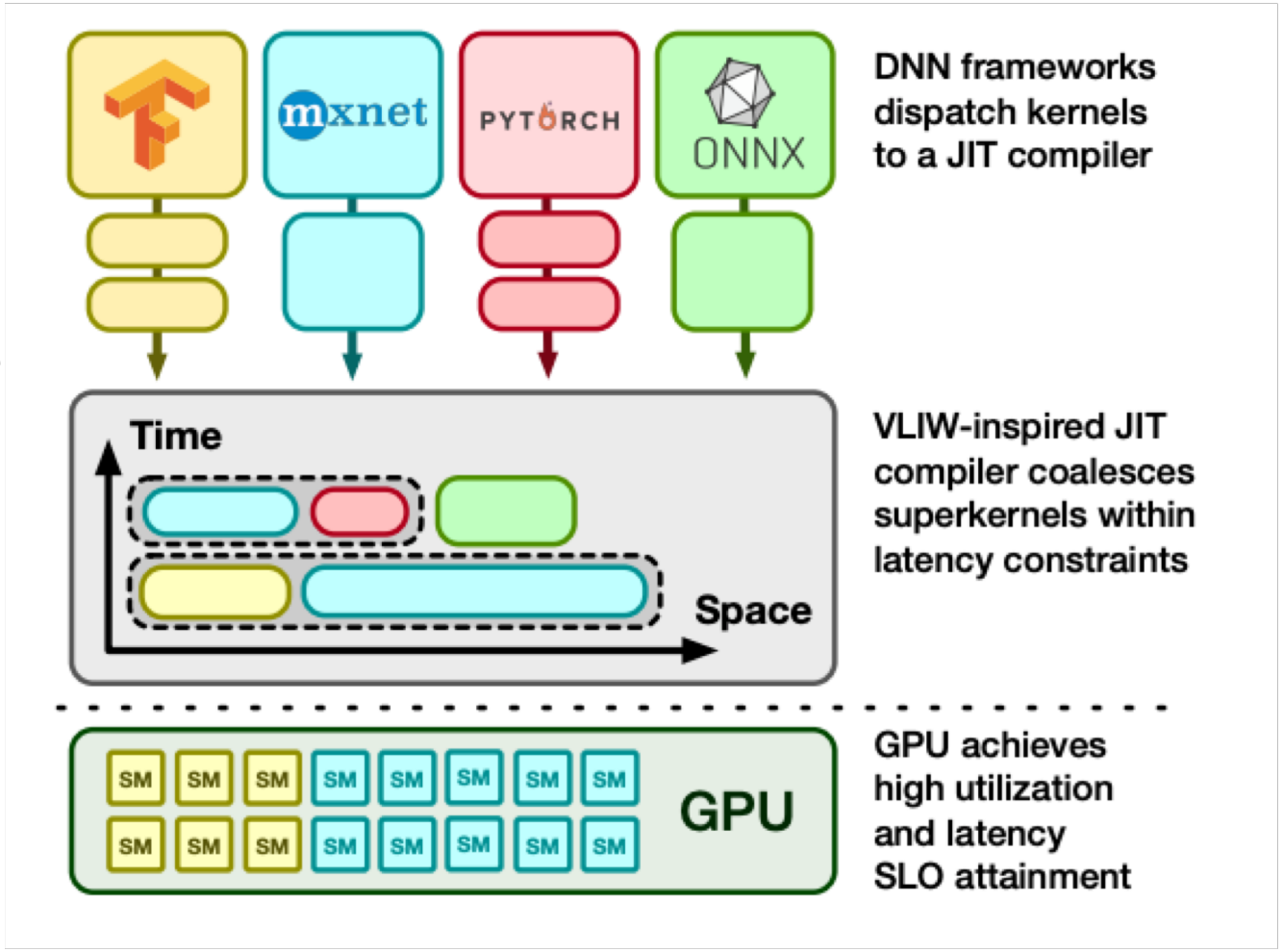


- SM: Streaming Multi-processors
- Each Block is assigned to a SM
- Each SM executes a group of 32 threads (warp) in lock steps
- Free context switch within SM among warps -> latency hiding



Future Research Direction

- Compilers are great at Ahead of Time scheduling, what about Just-In-Time scheduling?
- Any way we can share GPU in predictable way and maximize utilization for DNN inference?
- Can we optimize for “fitness” of the kernel when it’s executed along with other kernels instead of its latency?



Performance Comparison: TVM vs TC