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

Tianqi Chen[1], Thierry Moreau[1], Ziheng Jiang[1,2], Lianmin Zheng[3], Eddie Yan[1]

Meghan Cowan[1], Haichen Shen[1], Leyuan Wang[4,2], Yuwei Hu[5], Luis Ceze[1], Carlos Guestrin[1]. Arvind Krishnamurthy[1]
[1]Paul G. Allen School of Computer Science & Engineering, University of Washington

[2] AWS, [3]Shanghai Jiao Tong University, [4]UC Davis, [5]Cornell

Jiahui Cheng

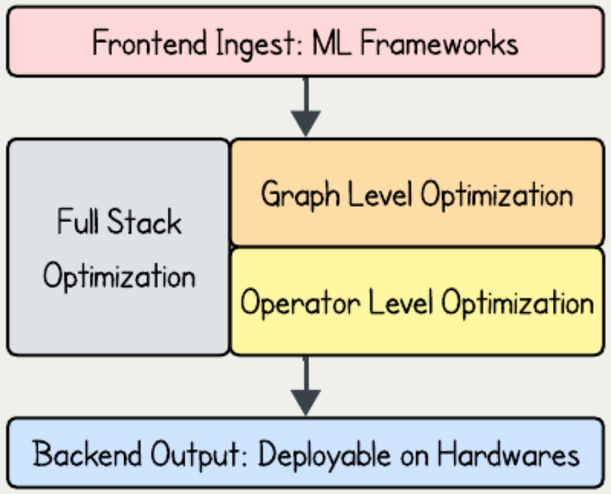
TVM: An automated (3) end-to-end (2) optimizing compiler for deep learning (1)

1. DL Complier

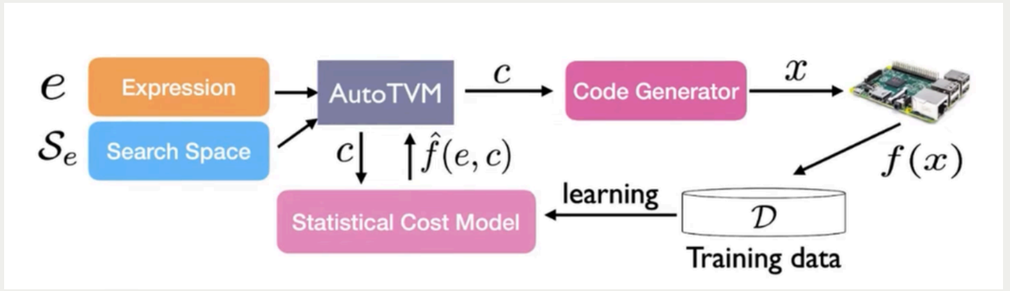
$$T_{\text{total}} = \underbrace{\frac{D_{\text{vol}}}{BW}}_{\text{Data}} + \underbrace{\frac{O}{R_{\text{peak}} \cdot \eta}}_{\text{Compute}} + \underbrace{L_{\text{lat}}}_{\text{Overhead}}$$

> The "source code" is the model architecture (the O term). The framework's job is to take this high-level math and compile it into a series of hardware-specific kernel launches that:
>
> 1. Minimize **Data Movement** (D_{vol}) through techniques like kernel fusion.
> 2. Maximize **Utilization** (η_{hw}) by matching operations to specialized hardware units like Tensor Cores.
> 3. Minimize **Overhead** (L_{lat}) through efficient asynchronous dispatch and graph capture.

2. End To End Transformation



3.1. ML for ML system



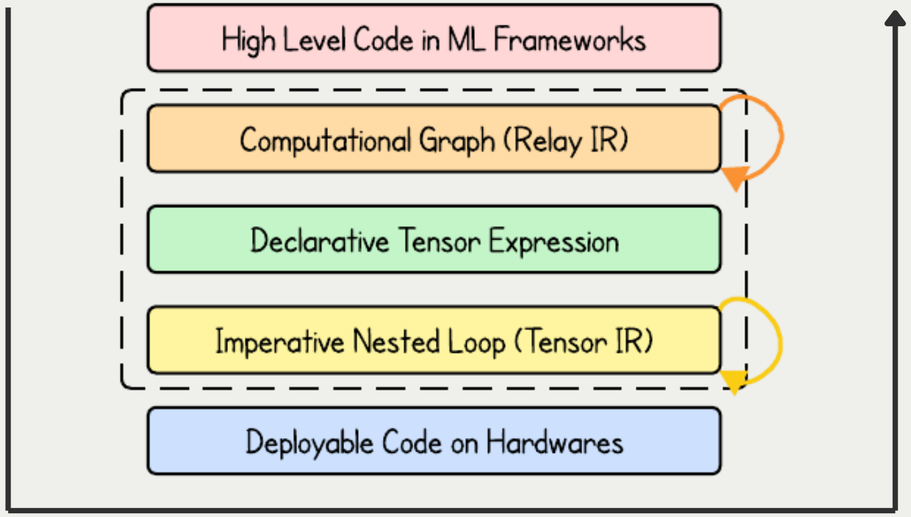
3.2. Solution to M x N x... Problem

"The combination of F frameworks, M microarchitectures (hardware targets), P primitives, S schedules per primitive (a primitive function may have different schedules depending on the size of the operands), and D different numerical formats has an implementation cost in the order of O(FMPSD)"

"Iron Law" Equation Source: "Machine Learning System Vol1 Chap 7"

Complexity Analysis Source: "Deep Learning Systems: Algorithms, Compilers, and Processors for Large-Scale Production Chap 9"

DL Model in layers of Abstraction...



1. ML framework-level code

```python
import torch

def matmul_layer(A, B):
    C = torch.matmul(A, B)
    return C
```

2. Relay IR: graph-level representation

```python
import tvm
from tvm import relay

A = relay.var("A", shape=(128, 64), dtype="float32")
B = relay.var("B", shape=(64, 256), dtype="float32")
B_T = relay.transpose(B, axes=(1, 0))

C = relay.nn.dense(A, B_T, units=256)

func = relay.Function([A, B], C)
mod = tvm.IRModule.from_expr(func)
```

3. Tensor Expression: declarative tensor computation

```python
from tvm import te

A = te.placeholder((128, 64), name="A", dtype="float32")
B = te.placeholder((64, 256), name="B", dtype="float32")

k = te.reduce_axis((0, 64), name="k")

C = te.compute(
    (128, 256),
    lambda i, j: te.sum(A[i, k] * B[k, j], axis=k),
    name="C"
)
```

4.1. TensorIR / TIR: imperative loop-level representation

```python
from tvm.script import tir as T

@T.prim_func
def matmul(
    A: T.Buffer((128, 64), "float32"),
    B: T.Buffer((64, 256), "float32"),
    C: T.Buffer((128, 256), "float32")
):
    for i, j, k in T.grid(128, 256, 64):
        with T.block("C"):
            vi, vj, vk = T.axis.remap("SSR", [i, j, k])

            with T.init():
                C[vi, vj] = T.float32(0)

            C[vi, vj] = C[vi, vj] + A[vi, vk] * B[vk, vj]
```

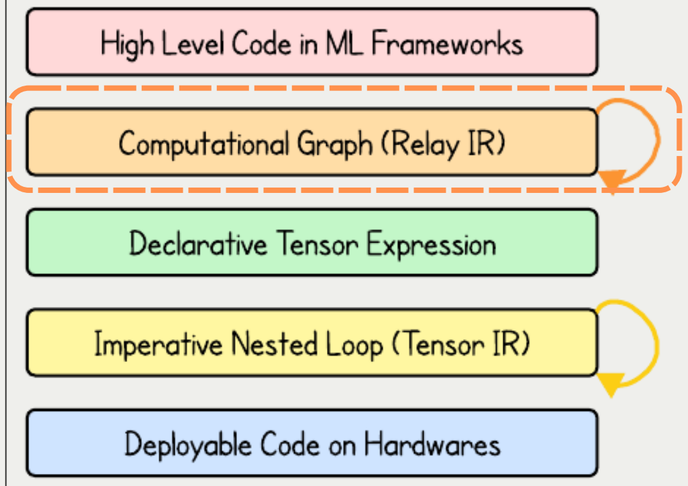
4.2. Scheduled TensorIR: tiled loop version

```python
@T.prim_func
def matmul_tiled(
    A: T.Buffer((128, 64), "float32"),
    B: T.Buffer((64, 256), "float32"),
    C: T.Buffer((128, 256), "float32")
):
    for io in T.parallel(16):           # 128 / 8
        for jo in range(32):            # 256 / 8

            for ii in range(8):
                for ji in T.vectorized(8):
                    i = io * 8 + ii
                    j = jo * 8 + ji

                    C[i, j] = T.float32(0)

                    for k in range(64):
                        C[i, j] = C[i, j] + A[i, k] * B[k, j]
```

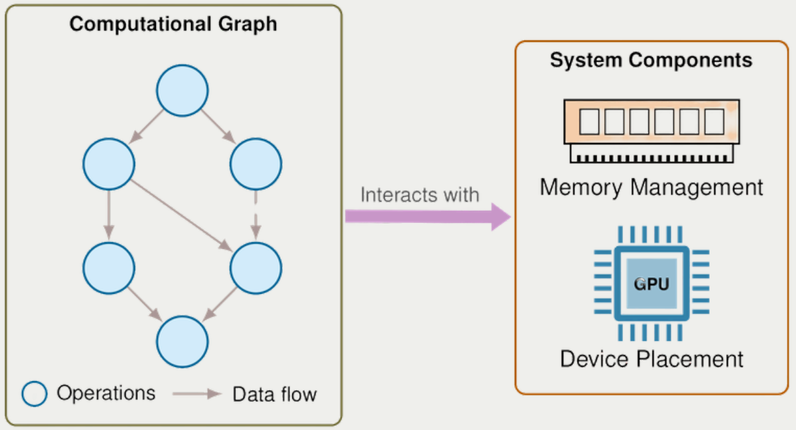
5. Lowered CPU-style loop code

```
void matmul(float* A, float* B, float* C) {
  for (int io = 0; io < 16; io++) {
    for (int jo = 0; jo < 16; jo++) {
      for (int ii = 0; ii < 8; ii++) {
        for (int ji = 0; ji < 8; ji++) {
          int i = io * 8 + ii;
          int j = jo * 8 + ji;

          C[i * 128 + j] = 0.0f;

          for (int k = 0; k < 128; k++) {
            C[i * 128 + j] += A[i * 128 + k] * B[k * 128 + j];
          }
        }
      }
    }
  }
}
```

Road Map



1. Computational Graph and Graph Level Optimization: The Macro-View of Operations



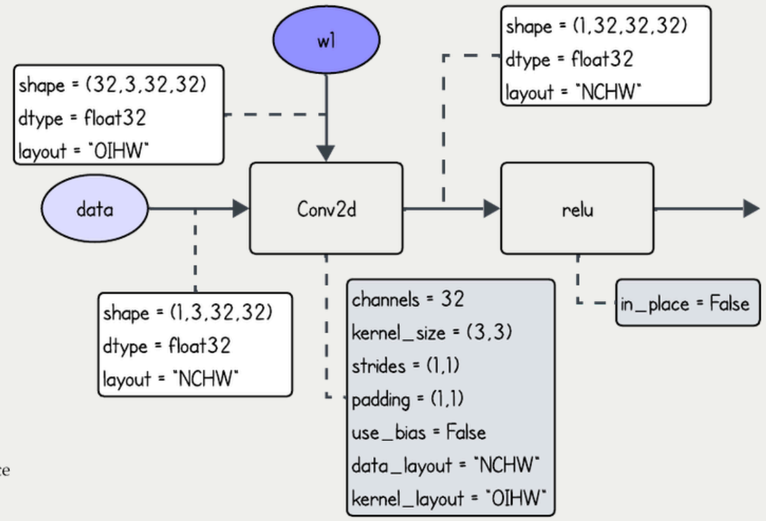


Figure 7.3: Computation Graph with System Interactions: A neural network computation graph (left) alongside system components including memory management and device placement (right) that interact with the graph to optimize resource allocation before execution.

Fun facts about Computational Graph

- Hardware-Agnostic
- DAG Data Structure
- Operate on Tensor
- Pure Dataflow

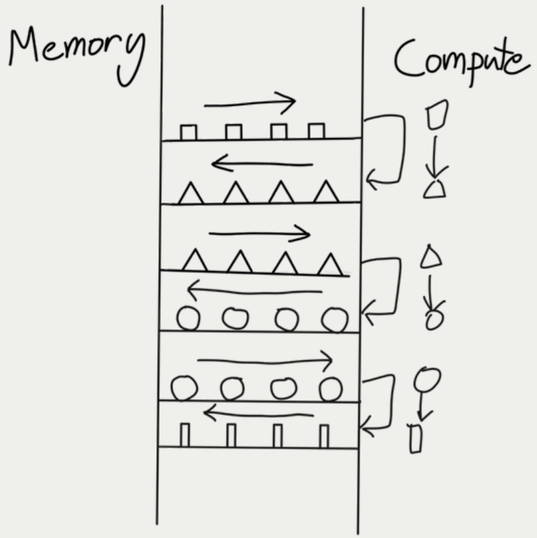
Possible Graph Level Optimizations

- Operator Fusion
- Constant Folding
- Static Memory Planning Pass
- Data Layout Transformations

(+ Dead Code Elimination
+ Common Expression Elimination)

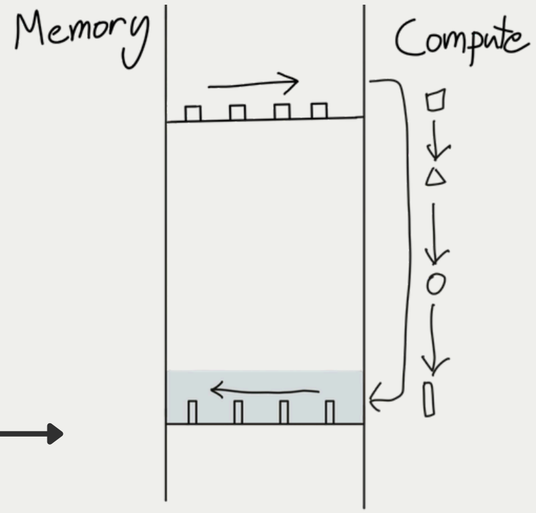
Figure 7.3 source: "Machine Learning System Vol1 Chap 7"

1.1. Operator Fusion: Boost Arithmatic Intensity



Here's how a sequence of pointwise operators might look like.

"Operator fusion combines multiple operators into a single kernel **without saving the intermediate results in memory.**"



Instead of sending our triangle back to global memory just to read it back again, we instead just do all of our operations in one go.

Four categories of operators:

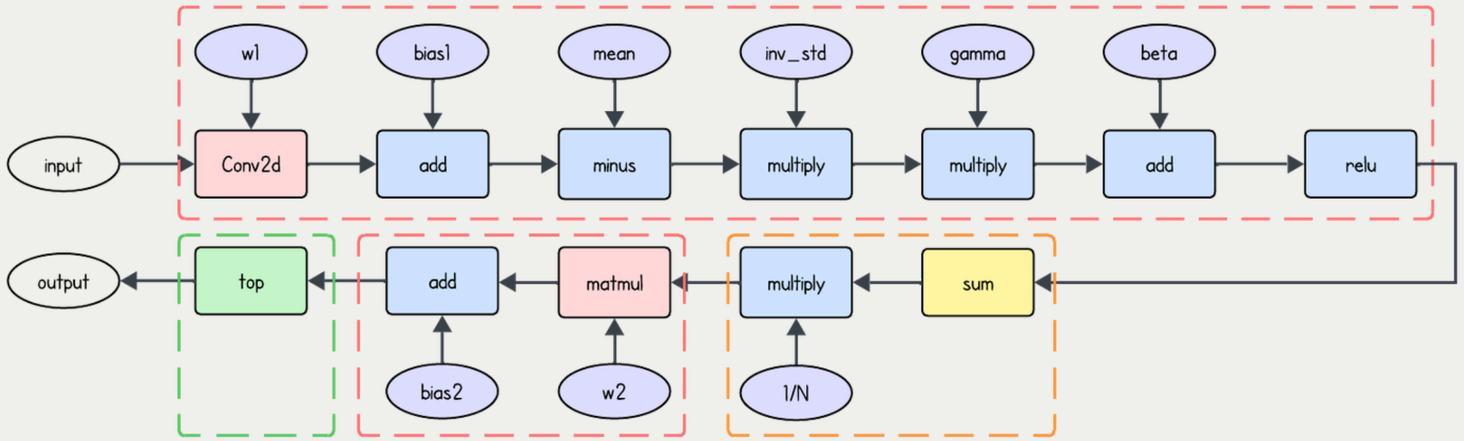
- injective (+element-wise)
- reduction
- complex-out-fusable
- opaque

Three practice of operator fusion:

- Injective → Injective
- Injective → Reduction
- Complex-out-fusable → Injective

Search Method:

- Heuristic / Rule-Based



Hand-Drawn Figure Source: "Making Deep Learning Go Brrrr from First Principles"

1.2. Data Layout Transformation: Align Tensor Storage Format to Physical Access Pattern

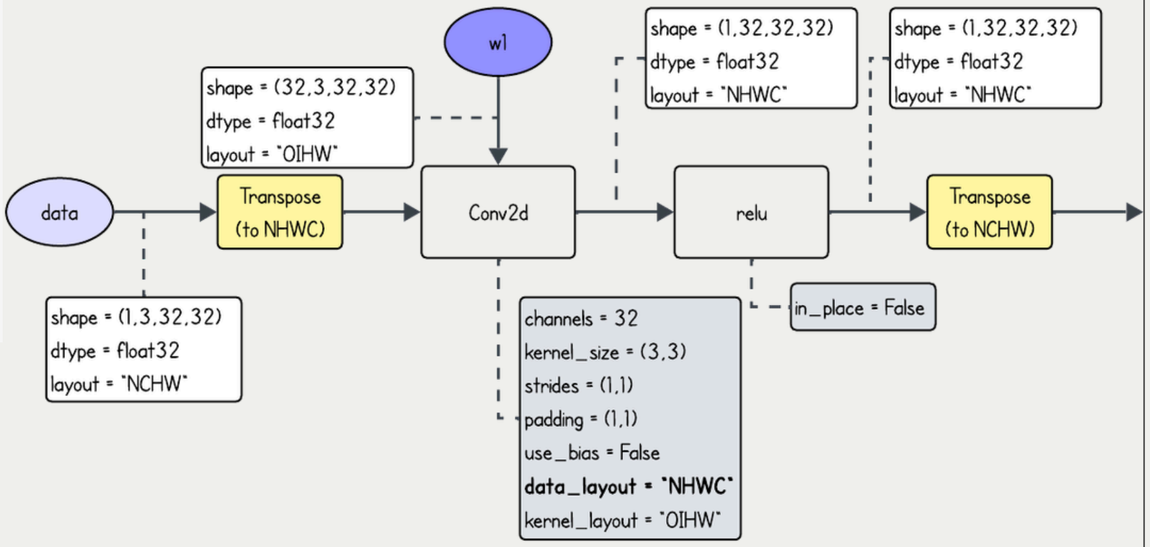
"Convert computational graph into one that can **use better internal data layouts** for execution on the target hardware."

Mapping logical dimensions to linear physical memory address space

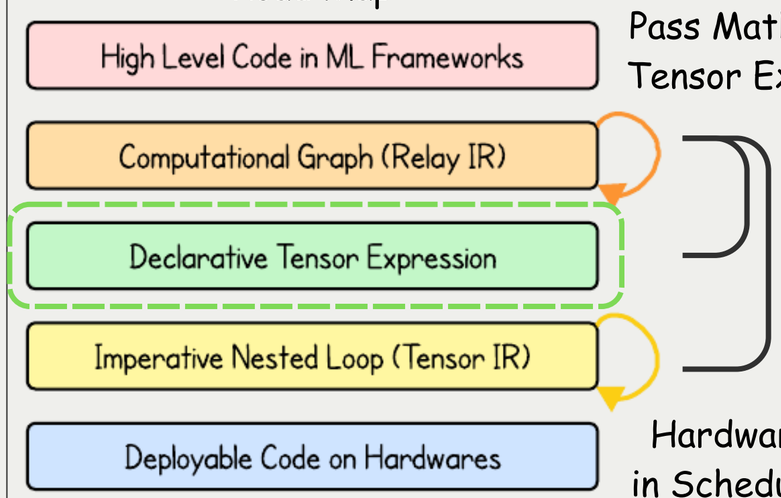
```
NCHW storage:
offset(n,c,h,w) = n x CHW + c x HW + h x W + w

NHWC storage:
offset(n,c,h,w) = n x HWC + h x WC + w x C + c

NCHW16c storage:
offset(n,c,h,w) = n x CHW + floor(c/16) x 16HW + h x 16W + w x 16 + (c mod 16)
```

1. Channel First
2. Channel Last
3. Blocked Layout

Explicit Layout Transformation + Propagation in Graph



Road Map



Pass Math Info to Tensor Expression

Hardware Info lie in Schedule System

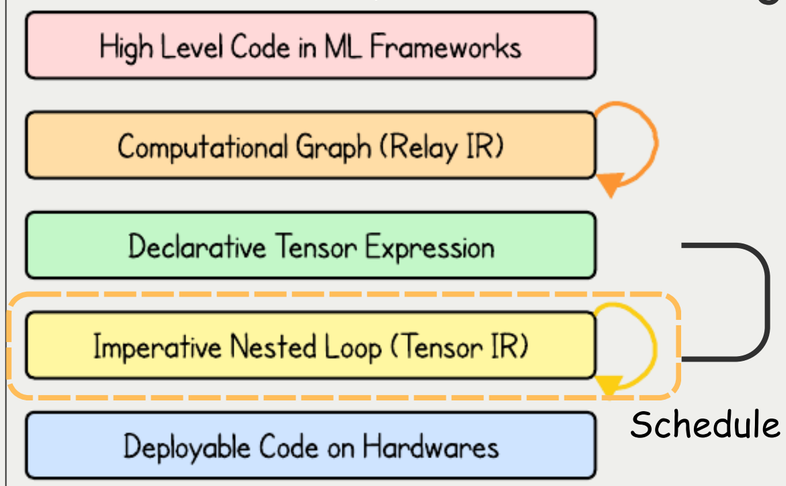
2. Declarative Tensor Expression: Separation of Compute and Schedule

"Tensor expression language **supports common arithmetic and math operations** and covers common DL operator patterns. The language **does not specify the loop structure and many other execution details.**"

```
from tvm import te

A = te.placeholder((128, 64), name="A", dtype="float32")
B = te.placeholder((64, 256), name="B", dtype="float32")

k = te.reduce_axis((0, 64), name="k")

C = te.compute(
    (128, 256),
    lambda i, j: te.sum(A[i, k] * B[k, j], axis=k),
    name="C"
)
```

A mathematical blueprint that defines the iteration spaces and the operations applied to them.

Road Map



Schedule Process:

"Build a schedule by **incrementally applying basic transformations** (schedule primitives) that preserve the program's logical equivalence."

3. Imperative Nested Loop: The Micro-View of Operations

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

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

+ Loop Tiling

```
yo, xo, ko, yi, xi, ki = s[C].tile(y, x, k, 8, 8, 8)
```

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

+ Cache Data on Accelerator Special Buffer

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

+ Map to Accelerator Tensor Instructions

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

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

schedule → schedule transformation ⇢ corresponding low-level code

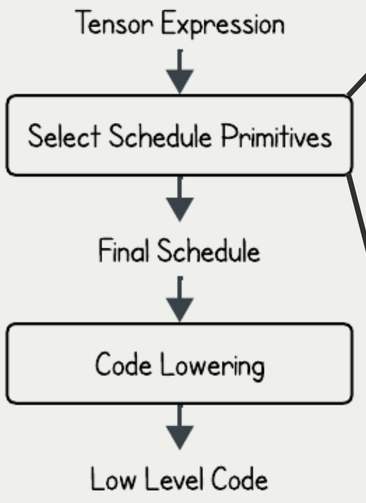
3.1. Nested Loop Transformation: Finding Efficient Hardware Mapping

Factors To Consider...

Mapping encompasses three aspects that form the foundation of effective AI accelerator design.

- **Computation placement**: Systematically assigns operations (e.g., matrix multiplications, convolutions) to processing elements to maximize parallelism and reduce idle time.
- **Memory allocation**: Carefully determines where model parameters, activations, and intermediate results reside within the memory hierarchy to optimize access efficiency.
- **Dataflow and Execution Scheduling**: Structures the movement of data between compute units to reduce bandwidth bottlenecks and ensure smooth, continuous execution.

TVM Schedule Primitives



Shedule Primitives	CPU	GPU	Accelerator
Loop Transformation	Y	Y	Y
Thread Binding	Y	Y	Y
Compute Locality	Y	Y	Y
Special Memory Scope		Y	Y
Tensorization	Y	Y	Y
Latency Hiding			Y

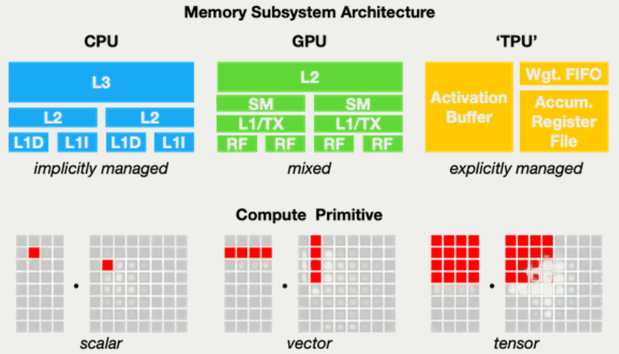


Figure 1: CPU, GPU and TPU-like accelerators require different on-chip memory architectures and compute primitives. This divergence must be addressed when generating optimized code.

"Mapping Principle" Paragraph Source: "Machine Learning System Vol1 Chap 7"

3.2. More about Schedule Primitives...

Special Memory Scope (GPU + DSA):
From "Shared-Nothing" to Cooperatively Fetching

[Def] "Memory fetch sharing, also known as nested parallelism with cooperation, improves fetching data from memory. Threads with local shared memory space cooperatively fetch data from higher levels in the memory hierarchy."

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
1.   __shared__float Mds[TILE_WIDTH][TILE_WIDTH];
2.   __shared__float Nds[TILE_WIDTH][TILE_WIDTH];

3.   int bx = blockIdx.x;  int by = blockIdx.y;
4.   int tx = threadIdx.x; int ty = threadIdx.y;

// Identify the row and column of the Pd element to work on
5.   int Row = by * TILE_WIDTH + ty;
6.   int Col = bx * TILE_WIDTH + tx;

7.   float Pvalue = 0;
// Loop over the Md and Nd tiles required to compute the Pd element
8.   for (int m = 0; m < Width/TILE_WIDTH; ++m) {

// Collaborative loading of Md and Nd tiles into shared memory
9.     Mds[ty][tx] = Md[Row*Width  + (m*TILE_WIDTH + tx)];
10.    Nds[ty][tx] = Nd[(m*TILE_WIDTH  + ty)*Width + Col];
11.    __syncthreads();

12.    for (int k = 0; k < TILE_WIDTH; ++k)
13.      Pvalue += Mds[ty][k] * Nds[k][tx];
14.    __syncthreads();
     }
15.  Pd[Row*Width + Col] = Pvalue;
}
```

FIGURE 5.7

Tiled matrix multiplication kernel using shared memories.

Introduce Corresponding Schedule Primitives in Schedule Space to mimic target CUDA program

```
# MEMORY SCOPES
PL = s.cache_write(Pd, "local")
Mds = s.cache_read(Md, "shared", [PL])
Nds = s.cache_read(Nd, "shared", [PL])

# THREAD MAPPING
row, col = s[Pd].op.axis
by, ty = s[Pd].split(row, factor=TILE_WIDTH)
bx, tx = s[Pd].split(col, factor=TILE_WIDTH)
s[Pd].reorder(by, bx, ty, tx)

# Bind to CUDA hardware execution unit
s[Pd].bind(by, te.thread_axis("blockIdx.y"))
s[Pd].bind(bx, te.thread_axis("blockIdx.x"))
s[Pd].bind(ty, te.thread_axis("threadIdx.y"))
s[Pd].bind(tx, te.thread_axis("threadIdx.x"))

# TILE-LEVEL COMPUTATION
s[PL].compute_at(s[Pd], tx)

# Split the reduction axis 'k' row_inner, col_inner = s[PL].op.axis
row_inner, col_inner = s[PL].op.axis
k_axis = s[PL].op.reduce_axis[0]
m, tk = s[PL].split(k_axis, factor=TILE_WIDTH)

# Reorder local loops so the tile loop 'm' is on the outside
s[PL].reorder(m, tk, row_inner, col_inner)

# COOPERATIVE FETCHING & SYNC
s[Mds].compute_at(s[PL], m)
s[Nds].compute_at(s[PL], m)

# Bind the fetch loops to the thread indices so they load cooperatively.
row_mds, col_mds = s[Mds].op.axis
s[Mds].bind(row_mds, te.thread_axis("threadIdx.y"))
s[Mds].bind(col_mds, te.thread_axis("threadIdx.x"))

row_nds, col_nds = s[Nds].op.axis
s[Nds].bind(row_nds, te.thread_axis("threadIdx.y"))
s[Nds].bind(col_nds, te.thread_axis("threadIdx.x"))
```

Definition Source: "Deep Learning Systems: Algorithms, Compilers, and Processors for Large-Scale Production Chap 8"
Figure 5.7 Source: "Programming Massively Parallel Processors: A Hands-on Approach Chap 5"

3.2. More about Schedule Primitives...

Tensorization (CPU + GPU + DSA):
Extensible Solution to Vectorize with rank > 1

[Def] "**Micro-kernel and intrinsic matching** matches and replaces a block of computations with the corresponding micro-kernel or corresponding hardware intrinsic."

2-Step Support

"We use the same tensor expression language to **declare both the behavior** of each new hardware intrinsic **and the lowering rule** associated with it."

"We introduce a **tensorize schedule primitive** to replace a unit of computation with the corresponding intrinsics. The compiler matches the computation pattern with a hardware declaration and lowers it to the corresponding hardware intrinsic."

```
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

Definition Source: "Deep Learning Systems: Algorithms, Compilers, and Processors for Large-Scale Production Chap 8"

3.2. More about Schedule Primitives...

Latency Hiding/Virtual Threading (DSA only):
For Decoupled Access-Execute Architecture which offload synchronization to software

[Def] "**Memory transfers**, also known as explicit memory latency handling, is used in conjunction with the memory allocation pass. It adds memory access instructions to transfer data to and from memory banks to overlap memory transfers with computations."

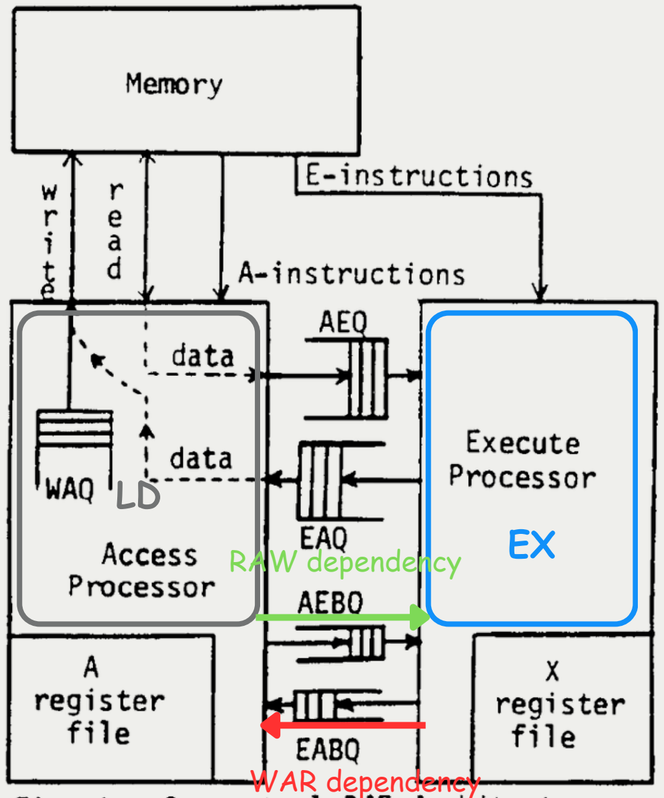
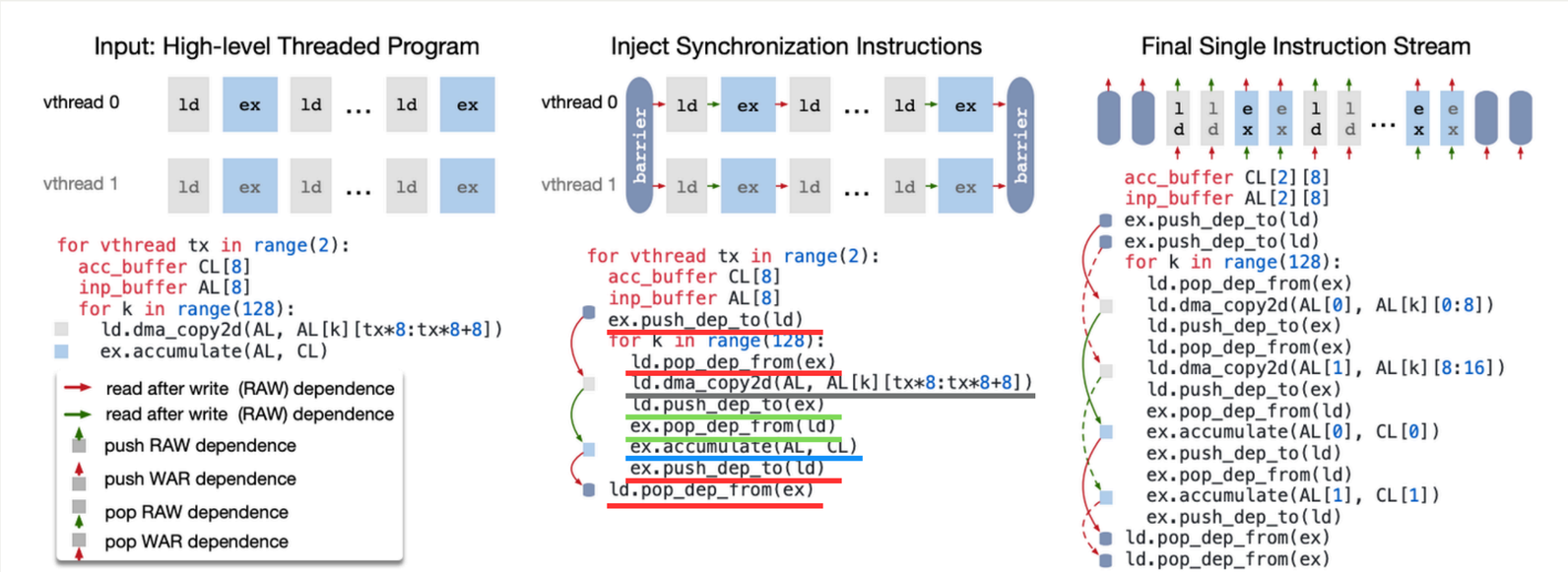


Fig. 1. Conceptual DAE Architecture



1. Write standard multi-threading program

2. Automatically inject low-level enqueue and dequeue tokens

3. Interleaves virtual threads into a single list of instructions

Definition Source: "Deep Learning Systems: Algorithms, Compilers, and Processors for Large-Scale Production Chap 8"
Figure 1 Source: "Decoupled Access/Execute Computer Architectures"

4.1. Explosive Factors to Consider and Possible Search Space

Table 11.13: Computation Placement Challenges: Effective neural network deployment requires strategic allocation of computations to processing elements, balancing workload distribution, data movement costs, and hardware constraints to maximize execution efficiency. These challenges guide the design of mapping strategies that optimize resource utilization and minimize communication overhead.

Challenge	Impact on Execution	Key Considerations for Placement
Workload Imbalance	Some processing elements finish early while others remain overloaded, leading to idle compute resources.	Distribute operations evenly to prevent stalls and ensure full utilization of PEs.
Irregular Computation Patterns	Models like transformers and GNNs introduce nonuniform computation demands, making static placement difficult.	Use adaptive placement strategies that adjust execution based on workload characteristics.
Excessive Data Movement	Frequent memory transfers introduce latency and increase power consumption.	Keep frequently used data close to the compute units and minimize off-chip memory accesses.
Limited Interconnect Bandwidth	Poorly placed operations can create congestion, slowing data movement between PEs.	Optimize spatial and temporal placement to reduce communication overhead.
Model-Specific Execution Needs	CNNs, transformers, and GNNs require different execution patterns, making a single placement strategy ineffective.	Tailor placement strategies to match the computational structure of each model type.

Table 11.14: Memory Allocation Challenges: Efficient memory management in AI accelerators balances data access speed with hardware constraints, mitigating performance bottlenecks caused by latency, bandwidth limitations, and irregular data patterns. Complex models such as transformers and graph networks impose variable and demanding memory requirements that amplify these challenges.

Challenge	Impact on Execution	Key Considerations for Allocation
High Memory Latency	Slow data access delays execution and reduces throughput.	Prioritize placing frequently accessed data in faster memory locations.
Limited On-Chip Storage	Small local memory constrains the amount of data available near compute units.	Allocate storage efficiently to maximize data availability without exceeding hardware limits.
High Off-Chip Bandwidth Demand	Frequent access to external memory increases delays and power consumption.	Reduce unnecessary memory transfers by carefully managing when and how data is moved.
Irregular Memory Access Patterns	Some models require accessing data unpredictably, leading to inefficient memory usage.	Organize memory layout to align with access patterns and minimize unnecessary data movement.
Model-Specific Memory Needs	Different models require different allocation strategies to optimize performance.	Tailor allocation decisions based on the structure and execution characteristics of the workload.

Mapping search space By combining the complexity from computation ordering, parallelization, and memory placement, the total mapping search space can be approximated as:

$$\mathcal{S} = \left(n^d \times d! \times \frac{d!}{(d-k)!} \right)^l$$

where:

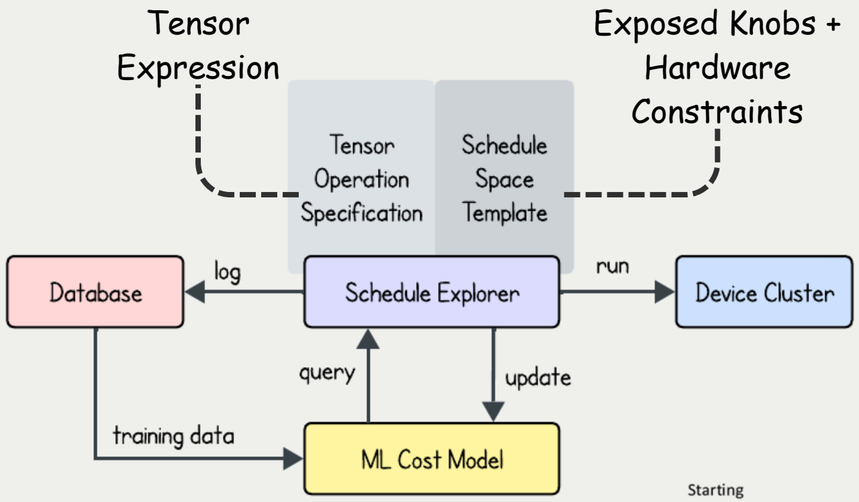
- n^d represents memory placement choices,
- $d!$ accounts for computation ordering choices,
- $\frac{d!}{(d-k)!}$ captures parallelization possibilities,
- l is the number of memory hierarchy levels.

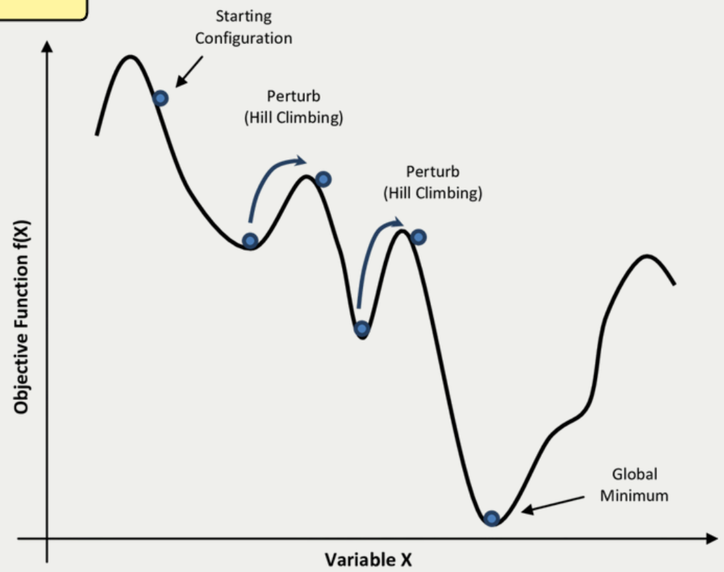
"Finding the optimal schedule is an NP-complete problem with potentially billions of choices for a single expression. RL and ML techniques can improve DL compilers."

Figures Source: "Machine Learning System Vol1 Chap 7"

Comment Source: "Deep Learning Systems: Algorithms, Compilers, and Processors for Large-Scale Production Chap 9"

4.2. Automated Nested Loop Schedule Optimizer





4.2.1 Main Component: Schedule Explorer

Define Search Space

"Search space contains every possible version of a kernel that produces the correct mathematical result but differs in its execution strategy. Constructing this space requires **parametrizing the transformation primitives**.The compiler then treats code generation as a constraint satisfaction and minimization problem where the **objective function is execution time**."

AutoTVM: Template Based

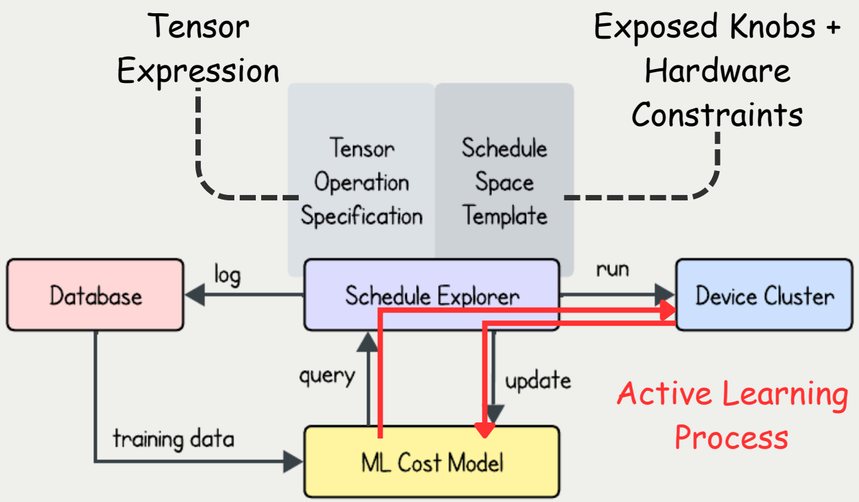
Define Search Algorithm

"Because we cannot compute gradients with respect to schedule parameters, we rely on **black-box optimization algorithms**. These algorithms sample configurations from the search space and **evaluating their quality**."

AutoTVM: Simulated Annealing

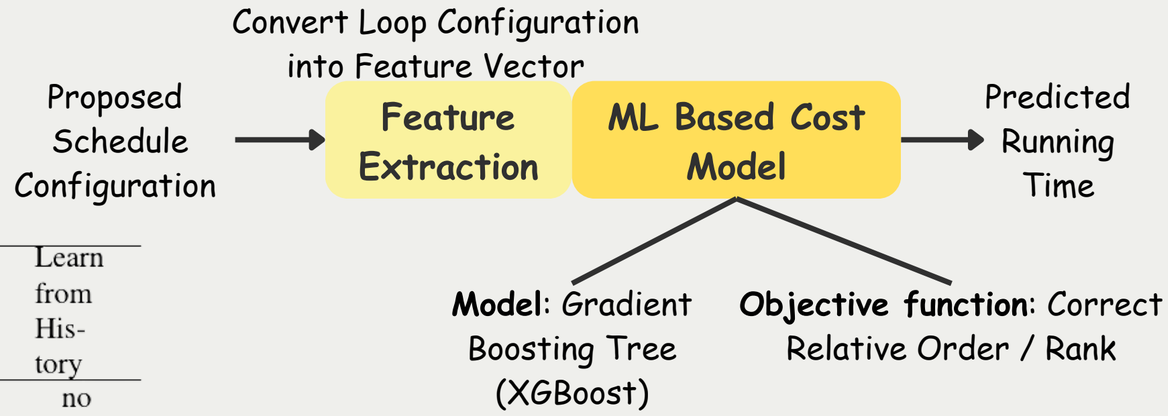
Source: "https://apxml.com/courses/compiler-optimizations-machine-learning"

4.2. Automated Nested Loop Schedule Optimizer



4.2.2 Main Component: ML Cost Model

"A cost model acts as a high-throughput surrogate for the hardware, **predicting the performance of a specific schedule without executing it**. This allows the search algorithm to query the performance of thousands of candidates per second, filtering down to only the most promising configurations for actual hardware benchmarking."



Other alternativs ...

Method Category	Data Cost	Model Bias	Need Hardware Info	Learn from History
Blackbox auto-tuning	high	none	no	no
Predefined cost model	none	high	yes	no
ML based cost model	**low**	**low**	**no**	**yes**

Table 1: Comparison of automation methods. Model bias refers to inaccuracy due to modeling.

Source: "https://apxml.com/courses/intro-ml-compiler-optimization"

5. Many Good Quantative Results...

Server Class GPU

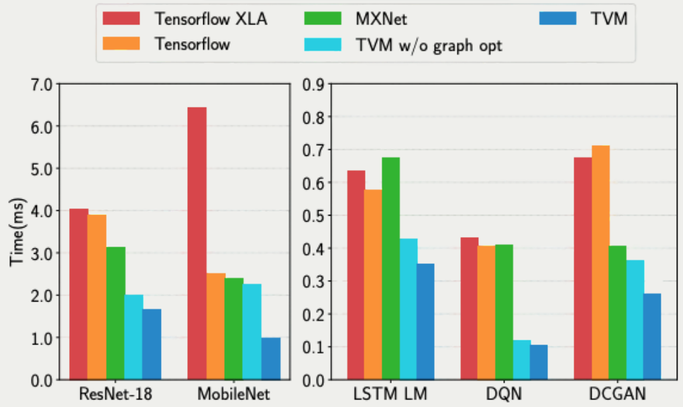


Figure 14: GPU end-to-end evaluation for TVM, MXNet, Tensorflow, and Tensorflow XLA. Tested on the NVIDIA Titan X.

Embedded CPU

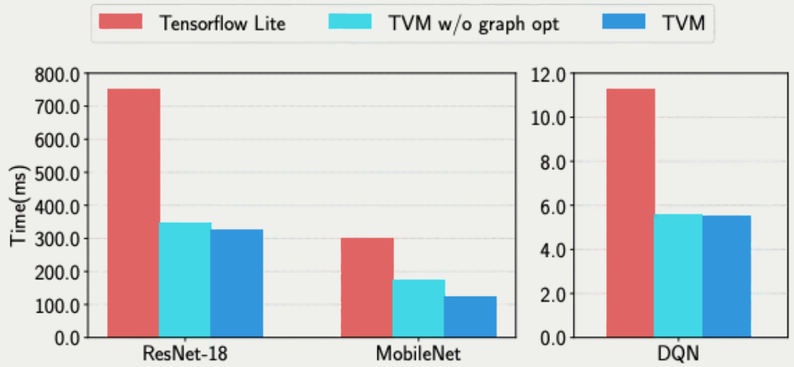


Figure 16: ARM A53 end-to-end evaluation of TVM and TFLite.

Embedded CPU

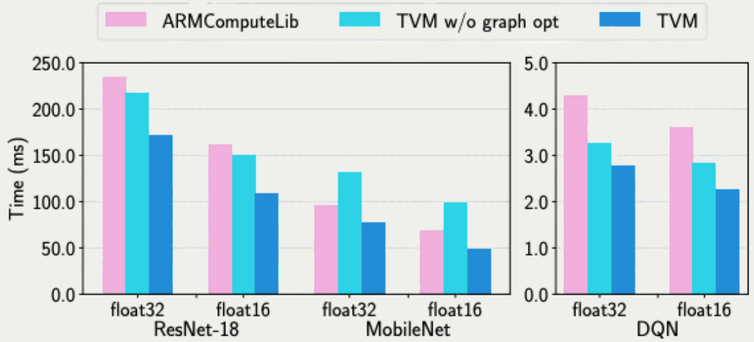


Figure 19: End-to-end experiment results on Mali-T860MP4. Two data types, float32 and float16, were evaluated.

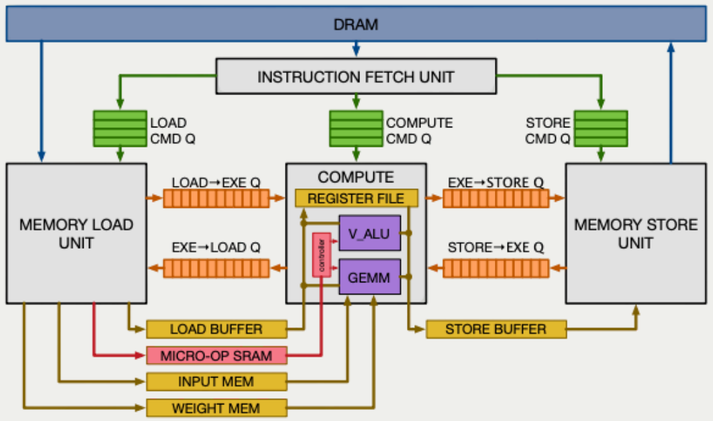


Figure 20: VDLA Hardware design overview.

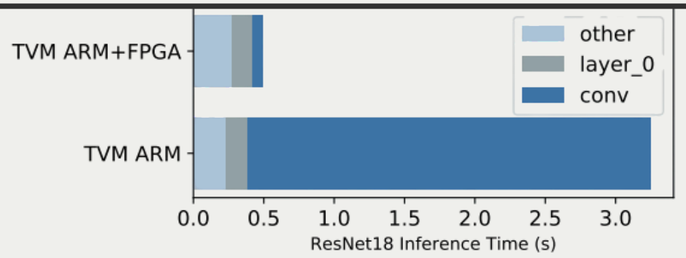


Figure 21: We offloaded convolutions in the ResNet workload to an FPGA-based accelerator. The grayed-out bars correspond to layers that could not be accelerated by the FPGA and therefore had to run on the CPU. The FPGA provided a 40x acceleration on offloaded convolution layers over the Cortex A9.