TVM: End-to-End Optimization Stack for Deep Learning

Tianqi Chen[1], Thierry Moreau[1], Ziheng Jiang[2,3], Haichen Shen[1]
Eddie Yan[1], Leyuan Wang[2,4], 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]Amazon Web Service, [3]Fudan University, [4]UC Davis, [5]TuSimple

Abstract

Scalable frameworks, such as TensorFlow, MXNet, Caffe, and PyTorch drive the current popularity and utility of deep learning. However, these frameworks are optimized for a narrow range of server-class GPUs and deploying workloads to other platforms such as mobile phones, embedded devices, and specialized accelerators (e.g., FPGAs, ASICs) requires laborious manual effort. We propose TVM, an end-to-end optimization stack that exposes graph-level and operator-level optimizations to provide performance portability to deep learning workloads across diverse hardware back-ends. We discuss the optimization challenges specific to deep learning that TVM solves: high-level operator fusion, low-level memory reuse across threads, mapping to arbitrary hardware primitives, and memory latency hiding. Experimental results demonstrate that TVM delivers performance across hardware back-ends that is competitive with state-of-the-art libraries for low-power CPU and server-class GPUs. We also demonstrate TVM's ability to target new hardware accelerator back-ends by targeting an FPGA-based generic deep learning accelerator. The compiler infrastructure and FPGA accelerator design will be open-sourced.

1 Introduction

Deep learning models can now recognize images, process natural language, and defeat humans in challenging strategy games. The steadily advancing compute capabilities of modern hardware has played a prominent role in deep learning's present ubiquity and relevance in many problem domains. Many of the most popular deep learning frameworks, such as TensorFlow, MXNet, Caffe, and PyTorch, harness the power of modern hardware by focusing support on a narrow class of server-class GPU devices—with this support depending on the use of highly engineered and vendor-specific GPU libraries. However, the number and diversity of specialized deep learning accelerators is increasing rapidly in the wild. These accelerators pose an adoption challenge as they introduce new hardware abstractions that modern compilers and frameworks are ill-equipped to deal with.

Providing support in various deep learning frameworks for diverse hardware back-ends in the present ad-hoc fashion is unsustainable. Ultimately, the goal is to easily deploy deep learning workloads to all kinds of hardware targets, including embedded devices, GPUs, FPGAs, and ASICs (e.g, the TPU), which significantly diverge in terms of memory organization, compute etc. as shown in Figure 1. Given these requirements, the development of an optimization frameworks that can lower a high-level specification of a deep learning program down to low-level optimized code for any hardware back-end is critical.

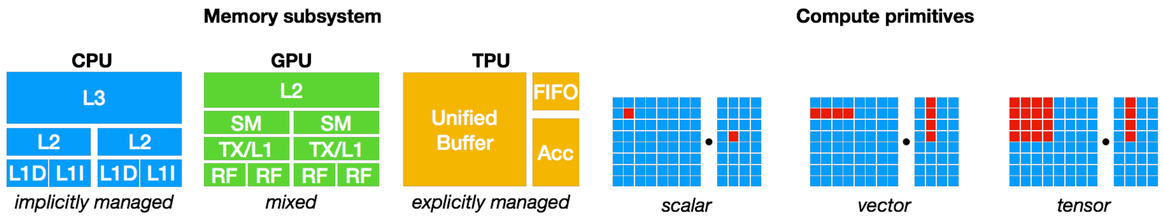
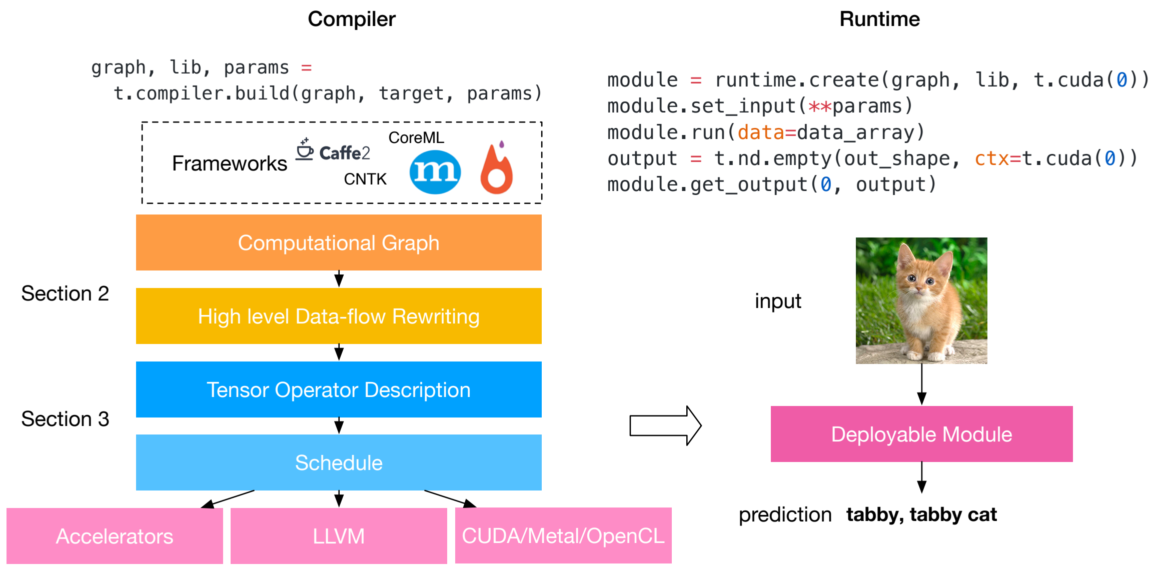


Figure 1: CPU, GPU and TPU-like accelerators require different on-chip memory architecture and compute primitives. This divergence must be addressed when generating optimized code.



```
graph, lib, params =
  t.compiler.build(graph, target, params)
```

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

Figure 2: The TVM Stack Diagram. Current stack support: descriptions from many deep learning frameworks and targeting major CPU, GPU and specialized accelerators.

Current deep learning frameworks rely on a computational graph intermediate representation to implement optimizations such as auto differentiation and dynamic memory management [3, 7, 4]. Graph-level optimizations, however, are often too high-level to handle hardware back-end-specific operator-level transformations. On the other hand, current operator-level libraries that deep learning frameworks rely on are too rigid and specialized to be easily ported across hardware devices. To address these weaknesses, we need a compiler framework that can expose optimization opportunities across both the graph and operator levels to deliver competitive performance across hardware back-ends.

Fundamental Optimization Challenges An optimizing compiler for deep learning needs to expose both high-level and low-level optimizations. We summarize four fundamental challenges at the computation graph level and tensor operator level:

1. *High-level dataflow rewriting*: Different hardware devices may have vastly different memory hierarchies, so enabling strategies to fuse operators and optimize data layouts are crucial for optimizing memory access.

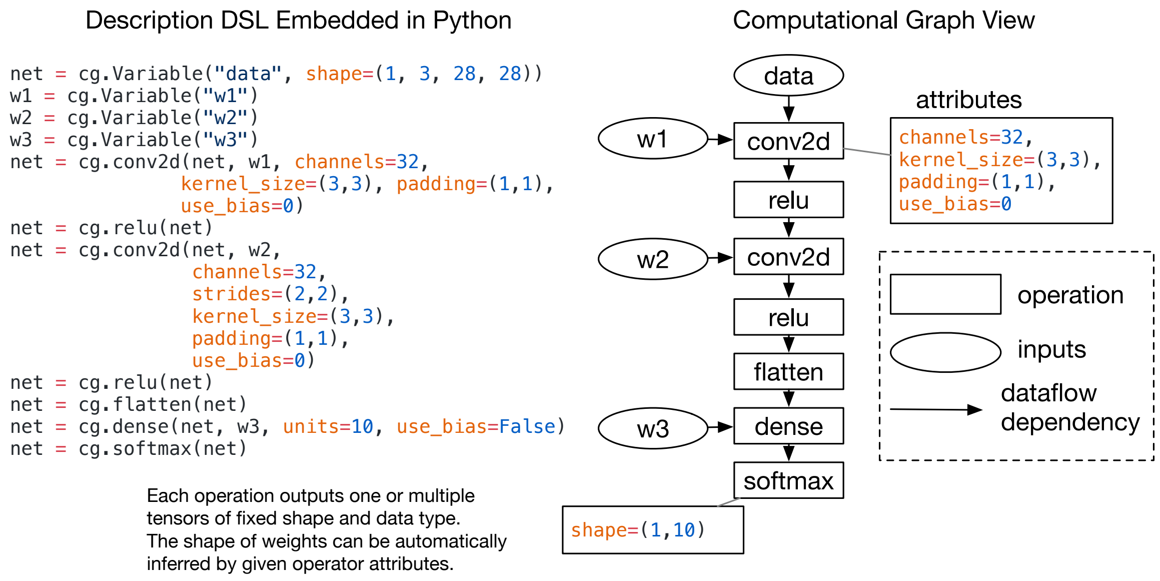
2. *Memory reuse across threads*: Modern GPUs and specialized accelerators have memory that can be shared across compute cores. The traditional shared-nothing nested parallel model is no longer optimal. Cooperation among threads on shared memory loaded is required for optimized kernels.

3. *Tensorized compute intrinsics*: The latest hardware provides new instructions that go beyond vector operations, like the GEMM operator in the TPU or the tensor core in NVIDIA's Volta. Consequently, the scheduling procedure must break computation into tensor arithmetic intrinsics instead of scalar or vector code.

4. *Latency Hiding*: While traditional architectures with simultaneous multithreading and automatically managed caches implicitly hide latency in modern CPUs/GPUs, specialized accelerator designs usually favor leaner control and offload most of the scheduling complexity to the compiler stack. Still, scheduling must be performed carefully to hide memory access latency.

TVM : An End-to-End Optimization Stack We present TVM (shown in Figure 2), an end-to-end optimizing compiler stack to lower and fine-tune deep learning workloads to diverse hardware back-ends. TVM is designed to separate the algorithm description, schedule, and hardware interface. This principle is inspired by Halide [21]'s compute/schedule separation, but extends the concept by separating the schedule from the target hardware intrinsics. This additional separation enables support for novel specialized accelerators and their corresponding new intrinsics. TVM presents two optimization layers: a computation graph optimization layer to address the first scheduling challenge, and a tensor optimization layer with new schedule primitives to address the remaining three challenges.

Operation	Description
conv2d	$B_{i,y,x} = \sum_{ry,rx,k} A_{k,y+rx,x+rx} W_{i,k,ry,rx}$
relu	$B_{i,y,x} = \max(A_{i,y,x}, 0)$
dense	$B_{i,j} = \sum_k A_{i,k} W_{j,k} + b_i$
softmax	$B_{i,j} = e^{A_{i,j}} / \sum_k e^{A_{i,k}}$


Description DSL Embedded in Python
Computational Graph View
data
w1
conv2d
relu
w2
conv2d
relu
flatten
w3
dense
softmax
attributes
channels=32,
kernel_size=(3,3),
padding=(1,1),
use_bias=0
shape=(1,10)
operation
inputs
dataflow
dependency
Each operation outputs one or multiple
tensors of fixed shape and data type.
The shape of weights can be automatically
inferred by given operator attributes.

```
net = cg.Variable("data", shape=(1, 3, 28, 28))
w1 = cg.Variable("w1")
w2 = cg.Variable("w2")
w3 = cg.Variable("w3")
net = cg.conv2d(net, w1, channels=32,
                kernel_size=(3,3), padding=(1,1),
                use_bias=0)
net = cg.relu(net)
net = cg.conv2d(net, w2,
                 channels=32,
                 strides=(2,2),
                 kernel_size=(3,3),
                 padding=(1,1),
                 use_bias=0)
net = cg.relu(net)
net = cg.flatten(net)
net = cg.dense(net, w3, units=10, use_bias=False)
net = cg.softmax(net)
```

Figure 3: Example computational graph of a two layer convolutional neural network. Each node in the graph represents an operation that consumes one or more tensors and produces one or multiple tensors. The tensor operation can be parameterized by attributes to configure its behavior (e.g., padding or strides).

By combining these optimization layers, TVM can take model descriptions from most deep learning frameworks, perform joint high-level and low-level optimizations, and generate hardware-specific optimized code for back-ends such as the Raspberry Pi, GPUs, and FPGA-based specialized accelerators. Our paper makes the following contributions:

- We build an end-to-end compilation optimization stack allowing the deployment of deep learning workloads specified in a high-level framework (including Caffe, MXNet, PyTorch, Caffe2, CNTK) to diverse hardware back-ends (including CPUs, GPUs, and FPGA-based accelerators).
- We identify the major optimization challenges in providing performance portability to deep learning workloads across diverse hardware back-ends. We introduce novel schedule primitives to take advantage of cross-thread memory reuse, novel hardware intrinsics, and latency hiding.
- We evaluate TVM on a generic FPGA-based accelerator to provide a concrete case study on how to optimally target specialized accelerators.

Our compiler generates deployable code that is performance-competitive with state-of-the-art vendor-specific libraries, and can target new specialized accelerator back-ends. The remainder of the paper is organized as follows. We describe graph optimizations in section 2 as well as tensor operator scheduling in section 3. We include experimental results in those two sections to quantitatively assess the optimizations that we propose. We discuss runtime support on section 4. Our end-to-end evaluation is in section 5. Related work is discussed in section 6.

2 Optimizing Computational Graphs

Computational Graph Computational graphs are a common way to represent programs in deep learning frameworks [3, 6, 7, 4]. Figure 3 shows an example computational graph representation of a two layer convolutional neural network. The main difference

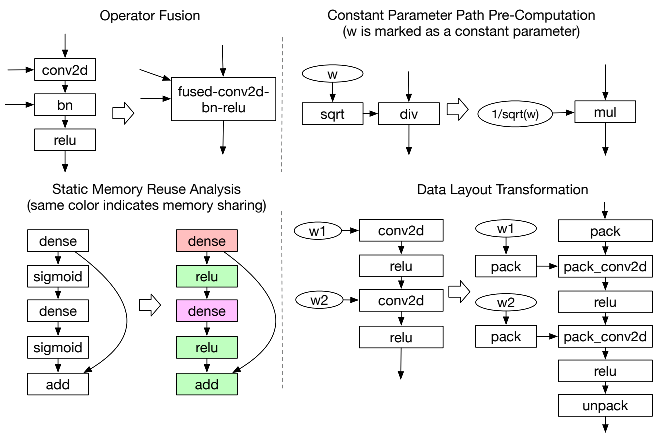


Figure 4: High-level dataflow optimizations in TVM.

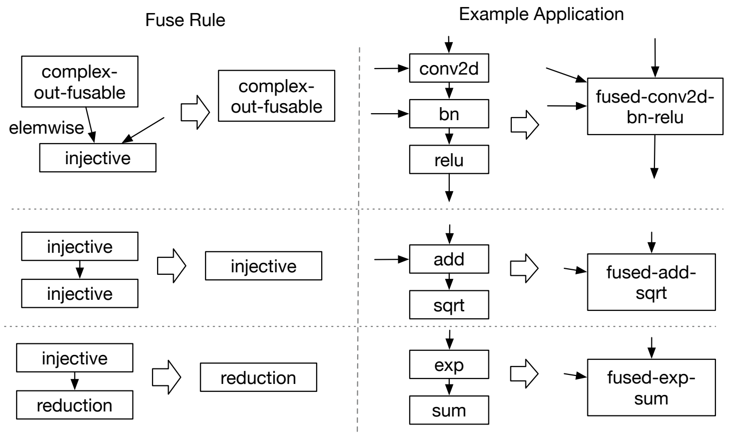


Figure 5: Rules for operation fusion pass.

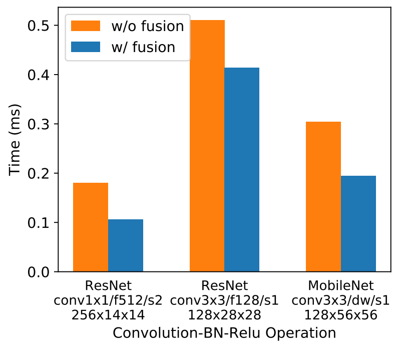


Figure 6: Comparison between fused operations of convolution, batch normalization, and relu, and non-fused operations on NVIDIA Tesla K80. Both are generated by TVM.

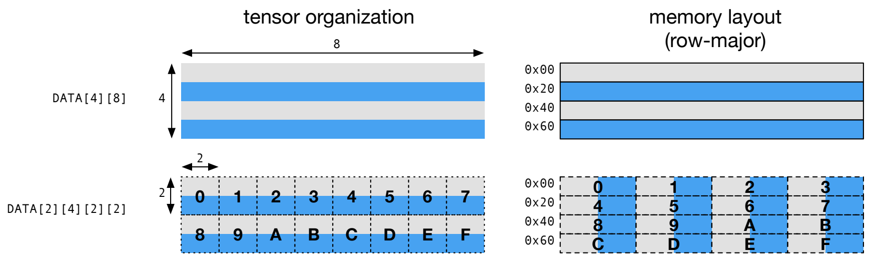


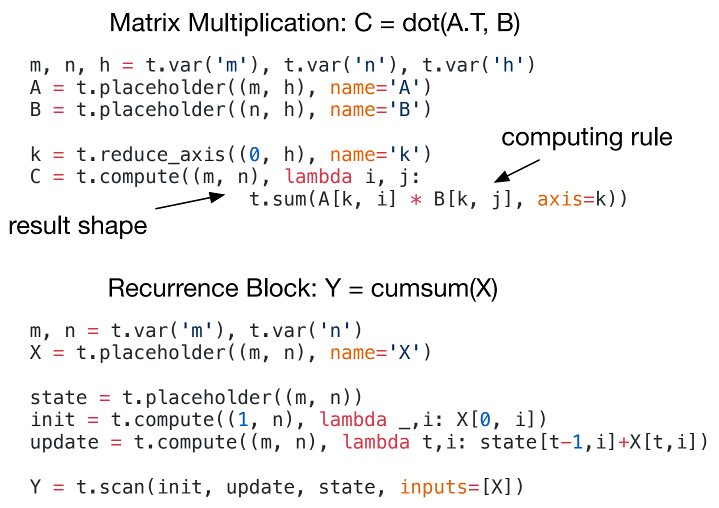
Figure 7: Data layout requirements can be affected by tensorization: here a 2×2 tensorized operation dictates a data layout transformation.

between this high-level representation and a low-level compiler IR, such as LLVM, is that the intermediate data items are large multi-dimensional tensors. TVM exploits a computational graph representation to apply high-level optimizations: nodes represent operations on tensors of statically known dimensions, and edges represent dataflow data dependences between tensor operations.

Computational graphs provide a global view on computation tasks, yet avoid specifying how each computation task needs to be implemented. A static memory planning pass can be performed on the graph to pre-allocate memory to hold each intermediate tensor result. This assignment phase is similar to register allocation passes in traditional compilers. Similar to LLVM IR, a computational graph can be transformed into functionally equivalent graphs to apply optimizations. For example, a constant-folding pass can be applied to pre-compute the parts of the graph that can be determined statically, saving execution cost. Figure 4 gives an overview of novel graph-level optimizations implemented in TVM: operator fusion and data layout transformation.

Operator Fusion Fusing multiple operators together is an optimization that can greatly reduce execution time, particularly in GPUs and specialized accelerators. The idea is to combine multiple operators together into a single kernel without saving the intermediate results back into global memory. Specifically, we recognize four categories of graph operators: injective (one-to-one map), reduction, complex-out-fusable (can fuse element-wise map to output), and opaque (cannot be fused). We apply the rules listed in Figure 5 to transform the computation graph into a fused version. Figure 6 demonstrates the impact of this optimization by comparing the performance of fused and non-fused version in three workloads.

Data Layout Transformation Tensor operations are the basic operators of computational graphs. The tensors involved in computation can have divergent layout requirements across different operations. For instance, a deep learning accelerator might exploit 4×4


Matrix Multiplication: C = dot(A.T, B)
computing rule
result shape
Recurrence Block: Y = cumsum(X)

```
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 i, j:
                   t.sum(A[k, i] * B[k, j], axis=k))
```

```
m, n = t.var('m'), t.var('n')
X = t.placeholder((m, n), name='X')

state = t.placeholder((m, n))
init = t.compute((1, n), lambda _,i: X[0, i])
update = t.compute((m, n), lambda t,i: state[t-1,i]+X[t,i])

Y = t.scan(init, update, state, inputs=[X])
```

Figure 8: Tensor expression language used in TVM for algorithm description. Each compute operation contains the shape of the output tensor, and an expression describing how to compute each element of the output tensor.

tensorized operations, requiring data to be tiled into 4×4 chunks to optimize for access locality. Figure 7 shows how a matrix's data layout can be transformed to accommodate for 2×2 tensorized operations on its data. Optimizing data layout starts with specifying the preferred data layout of each operator given the constraints dictated their implementation in hardware. We then perform the proper layout transformation between a producer and a consumer if their data layout does not match.

Limitations of Graph-Level Optimizations While high-level dataflow graph optimizations can greatly improve the efficiency of deep learning workloads, they are only as effective as what the operator library provides. Currently, the few deep learning frameworks that support operator fusion require the operator library to provide an implementation of the fused patterns. With more network operators introduced on a regular basis, the number of possible fused kernels can grow dramatically. This approach is no longer sustainable when targeting an increasing number of hardware back-ends, as the required number of fused patterns implementations grows combinatorially with the number of data layouts, data types, and hardware accelerator intrinsics that need to be supported. It is not feasible to handcraft operator kernels for this massive space of back-end specific operators. To this end, we propose a code-generation approach that can generate tensor operators in the next section.

3 Optimizing Tensor Operations

This section addresses how TVM can generate fine-tuned versions of the same operator for a wide array of hardware back-ends.

3.1 Tensor Expression Language

We introduce a dataflow tensor expression language to support automatic code generation. Unlike high-level computation graph languages, where the implementation of tensor operations is opaque, each operation is described in an index formula expression language as displayed in Figure 8.

Our tensor expression language takes cues from languages like Halide [21], Darkroom [13], and TACO [18]. Our tensor expression language supports common arithmetic and math operations found in common languages like C. We explicitly introduce a commutative reduction operator to easily schedule commutative reductions across multiple threads. We further introduce a high-order scan operator that can combine basic compute operators to form recurrent computations over time. TVM compute operations also support reduction among tuples of tensors, making it easy to support functions like argmax. This representation can describe all tensor operations used in the high-level dataflow graph and covers the common patterns exhibited in deep learning.

CPU:
Tiled Computation
Persistent Parallel Group

```
launch parallel threads(pg):
  for each yo in 0..128:
    for each xo in 0..128 parallel by(pg):
      alloc CL[8][8] = 0
      for each k in 0..1024:
        alloc AL[8] = A[xo*8:xo*8+8][k]
        alloc BL[8] = B[yo*8:yo*8+8][k]
        for each yi in 0..8:
          for each xi in 0..8:
            CL[yi][xi] += AL[yi] * BL[xi]
      C[xo*8:xo*8+8][yo*8:yo*8+8] = CL
```

GPU: Thread Cooperation

```
for thread_group (by, bx) in cross(0..64, 0..64):
  for thread_item (ty, tx) in cross(0..2, 0..2):
    local CL[8][8] = 0
    shared AS[2][8], BS[2][8]
    for each k in 0..1024:
      for each i in 0..4:
        AS[ty][i*4+tx] = A[by*16+ty*8+i*2+tx][k]
      for each i in 0..4:
        BS[ty][i*4+tx] = B[bx*16+ty*8+i*2+tx][k]
      memory_barrier_among_threads()
      for each yy in 0..8:
        for each xx in 0..8:
          CL[yy][xx] += AS[ty][yy] * BS[tx][xx]
      for each yi in 0..8:
        for each xi in 0..8:
          C[yo*8+yi][xo*8+xi] = CL
```

Accelerator:
Tensor Compute Intrinsics
Explicit Instruction Synchronization

```
out_buffer CL[2][8][8]
in_buffer AL[2][8][8], BL[2][8][8]

for each yo in 0..64:
  q0.push_dep_to(q1)
  q0.push_dep_to(q1)
  for each xo in 0..128:
    q1.pop_dep_from(q0)
    q1.fill_zero(CL[0])
    q1.push_dep_to(q0)
    q1.pop_dep_from(q0)
    q1.fill_zero(CL[1])
    q1.push_dep_to(q0)
    for each k in 0..128:
      q0.pop_dep_from(q1)
      q0.dma_copy2d(AL[0], A[yo*8:yo*8+8][k])
      q0.dma_copy2d(BL[0], B[xo*8:xo*8+8][k])
      q0.push_dep_to(q1)
      q0.pop_dep_from(q1)
      q0.dma_copy2d(AL[1], A[512+yo*8:512+yo*8+8][k])
      q0.dma_copy2d(BL[1], B[xo*8:xo*8+8][k])
      q0.push_dep_to(q1)
      q1.pop_dep_from(q0)
      q1.fuse_gemm8x8_add(AL[0], BL[0], CL[0])
      q1.push_dep_to(q0)
      q1.pop_dep_from(q0)
      q1.fuse_gemm8x8_add(AL[1], BL[1], CL[1])
      q1.push_dep_to(q0)
    q0.pop_dep_from(q1)
    q0.dma_copy2d(C[yo*8:yo*8+8][xo*8:xo*8+8],CL[0])
    q0.push_dep_to(q1)
    q0.pop_dep_from(q1)
    q0.dma_copy2d(C[512+yo*8:512+yo*8+8][xo*8:xo*8+8],CL[1])
    q0.push_dep_to(q1)
```

Figure 9: Matrix multiplication code example to the highlight optimization challenges that each hardware back-end exposes. CPUs require tiling and work distribution across parallel groups to optimize cache locality. GPUs require thread cooperation to allow shared memory reuse among thread groups. ASIC and FPGA accelerators exploit tensorized computation and rely on explicit synchronization to hide memory access latency.

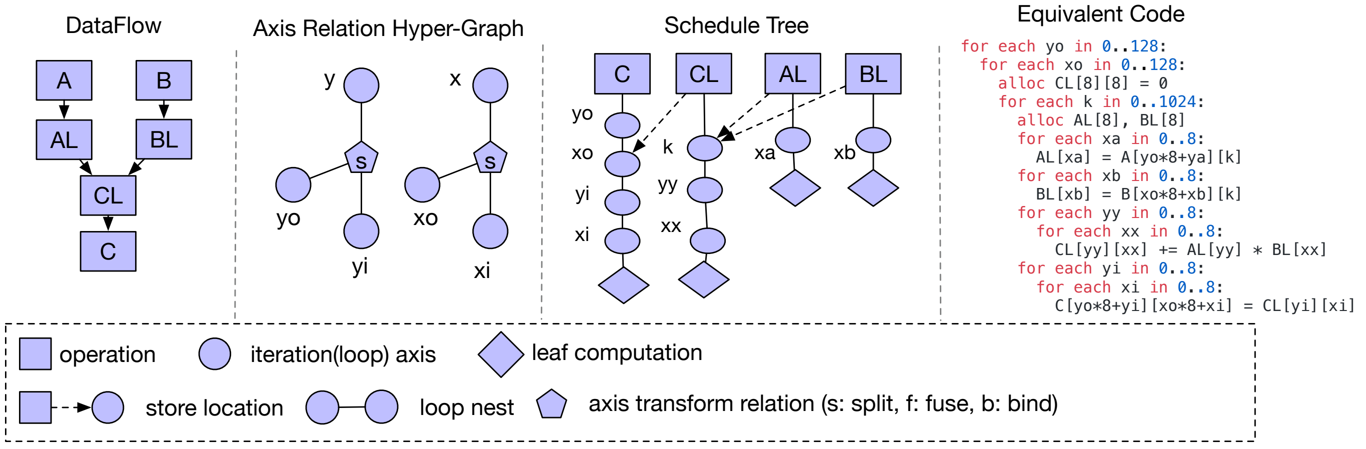
3.2 Schedule Space

Given a tensor expression, it is still challenging to create high-performance implementations for each hardware back-end. As an example, Figure 9 highlights typical optimizations applied to CPU, GPU, and deep learning accelerator designs. Each optimized low-level program is the result of different combinations of scheduling strategies, imposing a large burden on the kernel writer. We adopt the principle of decoupling compute descriptions from schedule optimizations from Halide [21]. Schedules are the specific rules that lower compute descriptions down to back-end-optimized implementations. The idea is to formally model the schedule space and the transformations used to traverse this space, thus providing different ways to generate low-level code. The schedule space of TVM is shown in Figure 10.

In order to quickly explore the schedule space, we need to provide effective schedule primitives that transform schedules. Figure 11 shows a collection of common schedule primitives used in TVM . Many of these primitives echo practices in high-performance computing. We adopt useful schedule primitives from Halide and introduce new ones to tackle the challenges introduced by GPU and specialized hardware accelerators. We describe these primitives in detail in the next three subsections.

3.3 Nested Parallelism with Cooperation

Parallel programming is key to improving the efficiency of compute intensive kernels in deep learning workloads. Modern GPUs offer massive parallelism, requiring us to bake parallel programming models into schedule transformations. Most existing solutions adopt a parallel programming model referred to as *nested parallel programs*, which is a form of fork–join parallelism. Specifically, we can use a parallel schedule primitive to parallelize a data parallel task. Each parallel task can be further recursively subdivided into subtasks to exploit the multi-level thread hierarchy on the target architecture (e.g., thread groups in GPU).



```
for each yo in 0..128:
  for each xo in 0..128:
    alloc CL[8][8] = 0
    for each k in 0..1024:
      alloc AL[8], BL[8]
      for each xa in 0..8:
        AL[xa] = A[yo*8+ya][k]
      for each xb in 0..8:
        BL[xb] = B[xo*8+xb][k]
      for each yy in 0..8:
        for each xx in 0..8:
          CL[yy][xx] += AL[yy] * BL[xx]
      for each yi in 0..8:
        for each xi in 0..8:
          C[yo*8+yi][xo*8+xi] = CL[yi][xi]
```

Figure 10: Schedule space representation in TVM. This example shows an explicitly tiled matrix multiplication. The dataflow representation provides an expression specifying the computation of each tensor element. The axis relation hyper-graph defines a bijective relation between the original loop axis and the transformed loop axis. The schedule tree represents the loop nest structure and producer store location of the computing structure. The schedule tree representation is derived from Halide [21], with a special emphasis on store location relation. Together, the dataflow representation, axis relation graph, and schedule tree describe a schedule that can be lowered to the code shown in the rightmost panel.

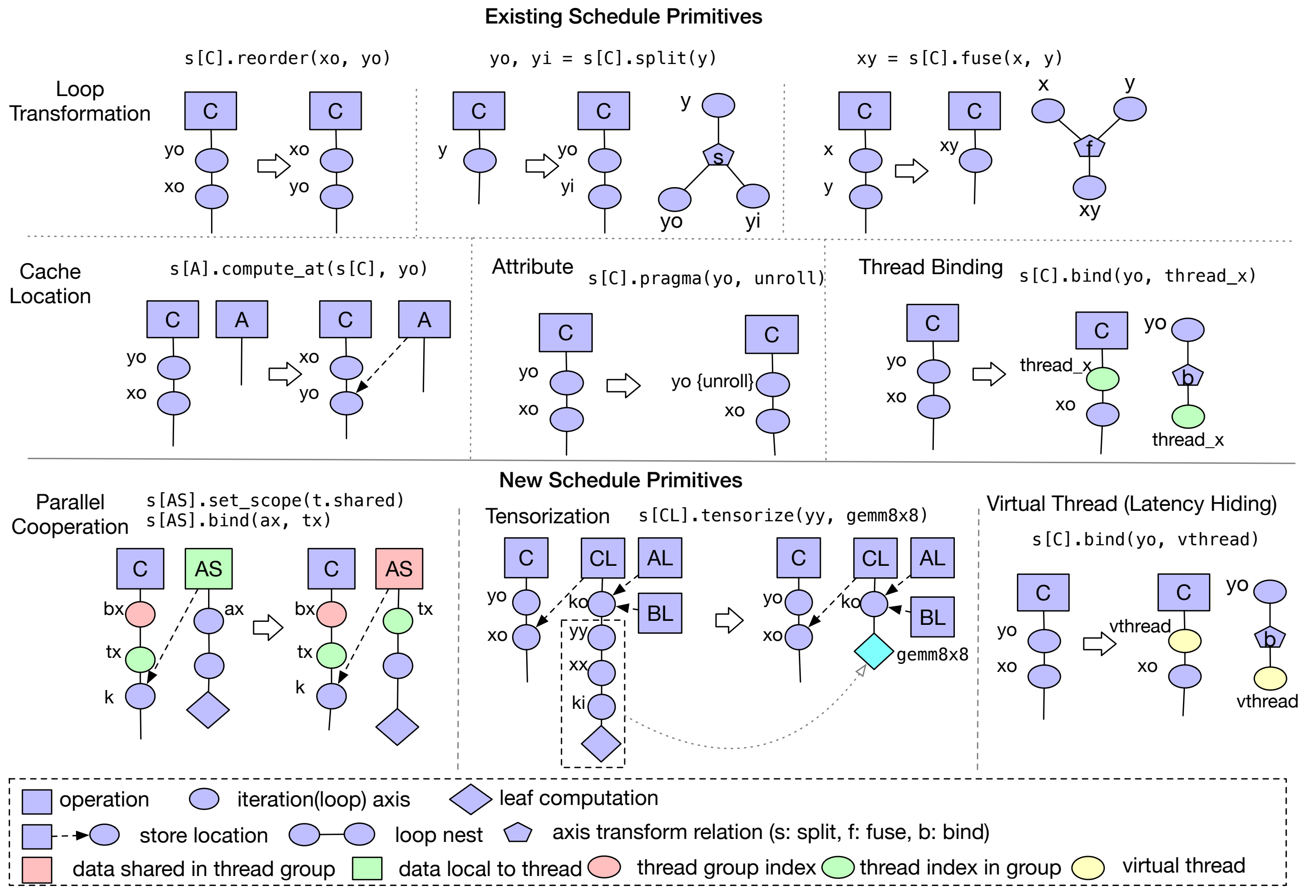
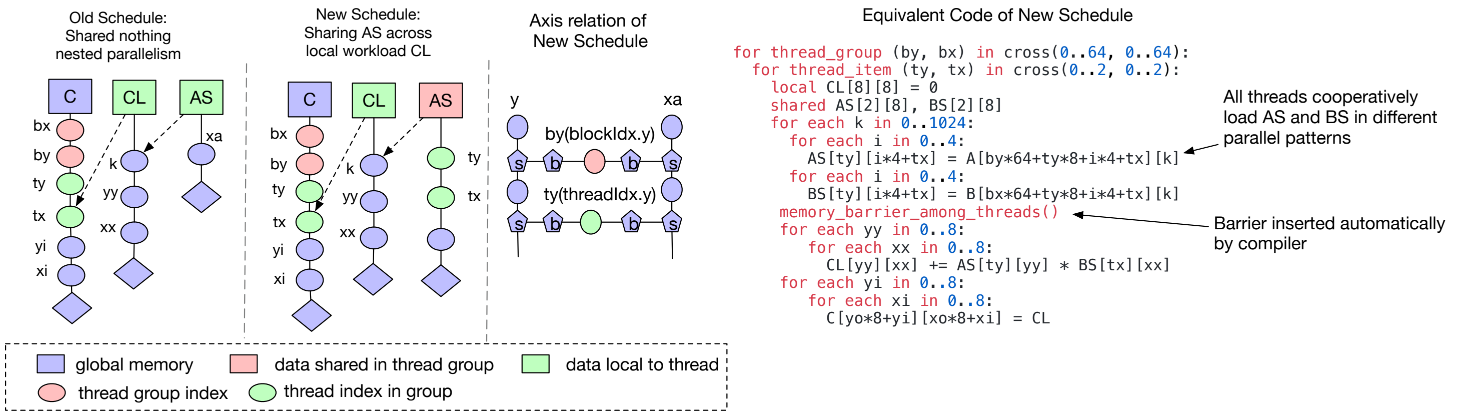


Figure 11: Schedule primitives (i.e. transformations) used in TVM .



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

Figure 12: Schedule for cooperative nested parallelism. Memory scoping is introduced to enable task sharing among parallel jobs. The resulting new schedule can take better advantage of the shared memory hierarchy in modern GPUs.

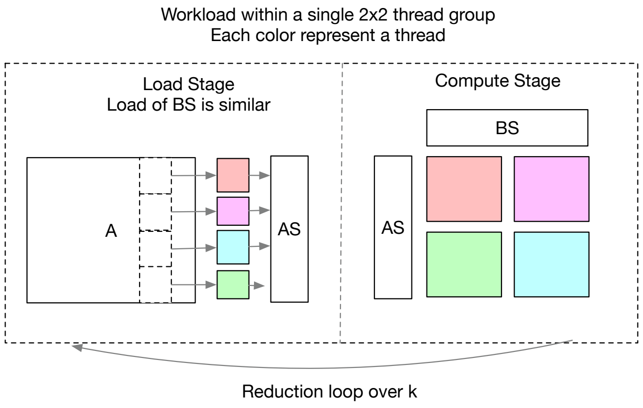


Figure 13: Memory Loading Pattern of Cooperative Parallel Program

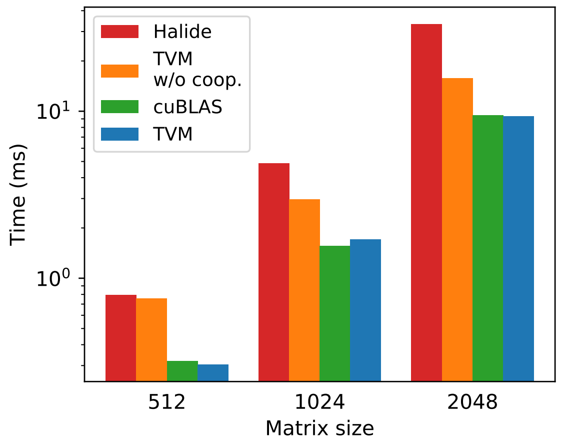


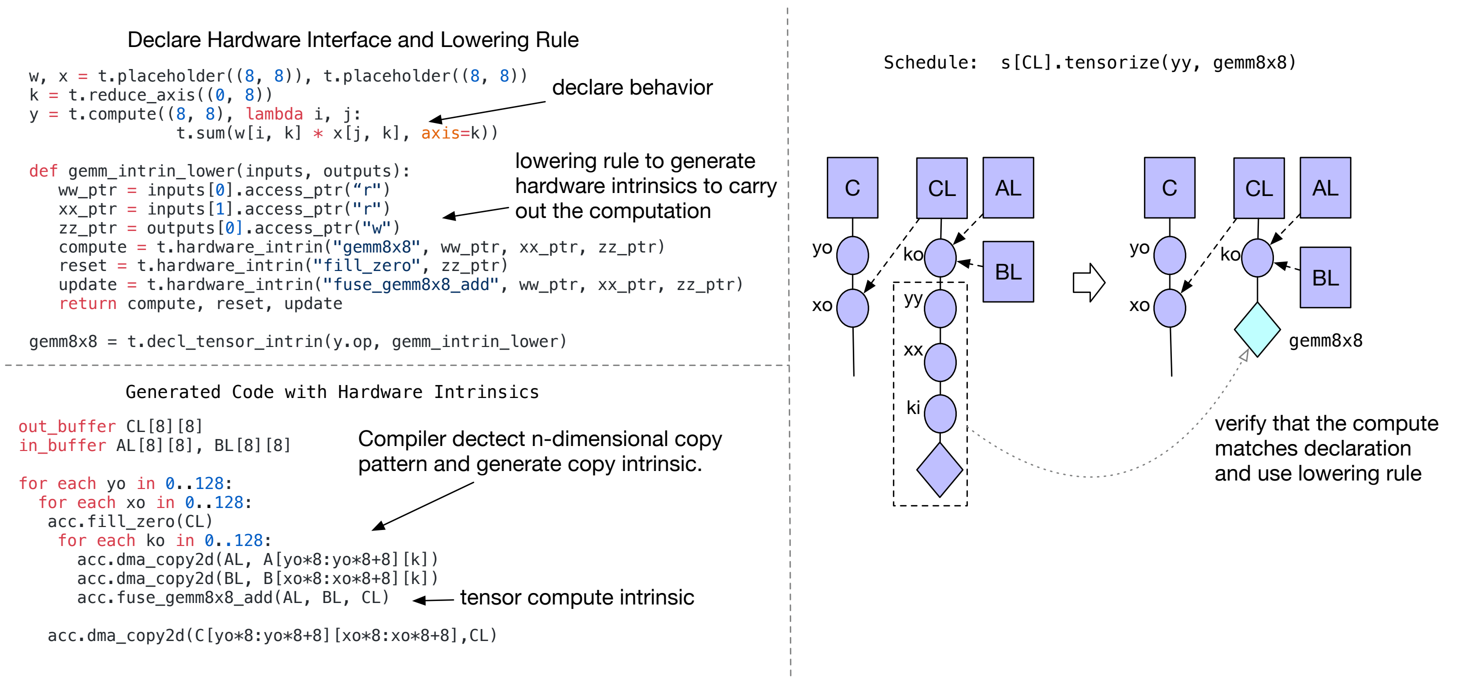
Figure 14: Matrix multiplication performance comparison using Halide, TVM with and without cooperation, and cuBLAS on Nvidia K80. By adopting the new schedule primitive, we can generate a kernel that is close to vendor-optimized performance.

We call this model *shared-nothing nested parallelism*, as one working thread cannot look at the data of its sibling within the same parallel computation stage. Interactions between sibling threads happen at the join stage, when the subtasks are done and next stage can consume the data produced by the previous stage. This programming model does not enable threads to *cooperate* with each other in order to perform collective task within the same parallel stage.

Figure 12 provides a matrix multiplication example demonstrating this limitation. Matrix multiplication on a GPU requires division of work into tiles that are distributed among thread groups, and then assigned to each individual thread. Under shared-nothing nested parallelism, each thread will have to independently fetch the data needed for its tile of the computation during the reduction phase. [1]

A better alternative to the shared-nothing approach is to fetch data cooperatively across threads. This pattern is well known in GPU programming using languages like CUDA, OpenCL and Metal, but has not been implemented into a schedule primitive. We introduce the concept of *memory scopes* to the schedule space, so that a stage can be marked as shared. Without memory scopes, automatic scope inference will mark the relevant stage as thread-local, as shown in Figure 12. The shared task needs to compute the dependencies of all the working threads in the group. We can efficiently schedule the input data loading by distributing the loading task among the same group of threads using the thread binding primitive. It's worth noting that we are forced to use the same threads to work on the division of

[1] It is possible to have a shared loading stage across threads before the reduction, but that requires loading every dependency used in the reduction, using too much memory.


Declare Hardware Interface and Lowering Rule

```
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)
```

Generated Code with Hardware Intrinsics

```
out_buffer CL[8][8]
in_buffer AL[8][8], BL[8][8]

for each yo in 0..128:
  for each xo in 0..128:
   acc.fill_zero(CL)
    for each ko in 0..128:
      acc.dma_copy2d(AL, A[yo*8:yo*8+8][k])
      acc.dma_copy2d(BL, B[xo*8:xo*8+8][k])
      acc.fuse_gemm8x8_add(AL, BL, CL)

   acc.dma_copy2d(C[yo*8:yo*8+8][xo*8:xo*8+8],CL)
```

```
Schedule:  s[CL].tensorize(yy, gemm8x8)
```

declare behavior
lowering rule to generate hardware intrinsics to carry out the computation
Compiler dectect n-dimensional copy pattern and generate copy intrinsic.
tensor compute intrinsic
C
CL
AL
BL
yo
xo
ko
yy
xx
ki
C
CL
AL
BL
yo
xo
ko
gemm8x8
verify that the compute matches declaration and use lowering rule

Figure 15: Tensorization helps us target specialized accelerators. We use the tensor expression language to describe hardware intrinsics. This approach decouples hardware lowering from scheduling and makes it easy to target new specialized accelerators.

shared workload, so threads can *persist* across the load and compute stages. This improved implementation requires additional compiler support. Specifically, the bound inference algorithm needs to be able to deduce the bounds of the shared task by merging the tasks of all the cooperative threads together. Additionally, memory synchronization barriers need to be properly inserted to guarantee that shared loaded data is visible to the consumers.

Figure 14 compares the performance of GPU kernels generated with shared nothing nested parallelism vs. with cooperation. We also compare TVM to Halide, which adopts a shared-nothing nested parallel approach. We find that it is critical to adopt these new schedule primitives to obtain optimal performance on GPUs. Finally, in addition to being useful to GPUs, memory scopes allow us to tag special memory buffers and create special lowering rules when targeting specialized deep learning accelerators.

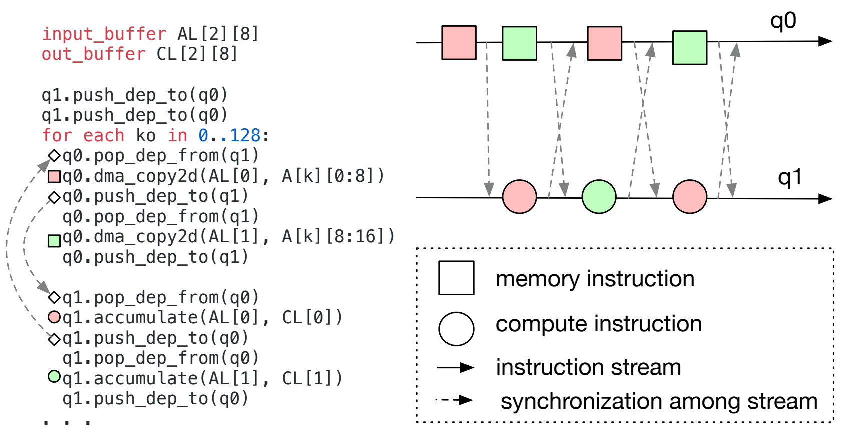
3.4 Tensorization: Generalizing the Hardware Interface

Deep learning workloads have high arithmetic intensity that can be typically decomposed into tensor operators like matrix-matrix multiplication or 1D convolution. These natural decompositions have led to the recent trend of adding tensor compute primitives that go beyond vector instructions. The emerging compute intrinsics are quite diverse, including examples like matrix-matrix multiplication [17], matrix-vector product [1] and 1D convolution [9]. These new primitives create new challenges for the tensor operator schedule: the schedule must use these primitives to benefit from acceleration. We dub this the *tensorization* problem, analogous to the vectorization problem for SIMD architectures.

Tensorization differs significantly from vectorization. The inputs to the tensor compute primitives are multi-dimensional, with fixed or variable lengths, and dictate different data layouts. More importantly, we cannot resort to a fixed set of primitives, as new deep learning accelerators are emerging with their own flavors of tensor instructions. Therefore, we need a solution that is *future proof* to support new generations of specialized accelerators.

To solve this challenge, we *separate the hardware interface from the schedule*. Specifically, we introduce a tensor intrinsic declaration mechanism. We can use the tensor expression language to declare the behavior of each new hardware intrinsic, as well as the lowering rule associated to it. Additionally, we introduce a *tensorize* schedule primitive to replace a unit of computation with the corresponding tensor intrinsics. The compiler matches the computation pattern with a hardware declaration, and lowers it to the corresponding the hardware intrinsic. Figure 15 shows an example of tensorization.

The tensor expression language describes both the users' intended compute description, and the abstractions that the hardware exposes. Tensorization decouples the schedule from specific hardware primitives, making TVM easy to extend to support new hardware

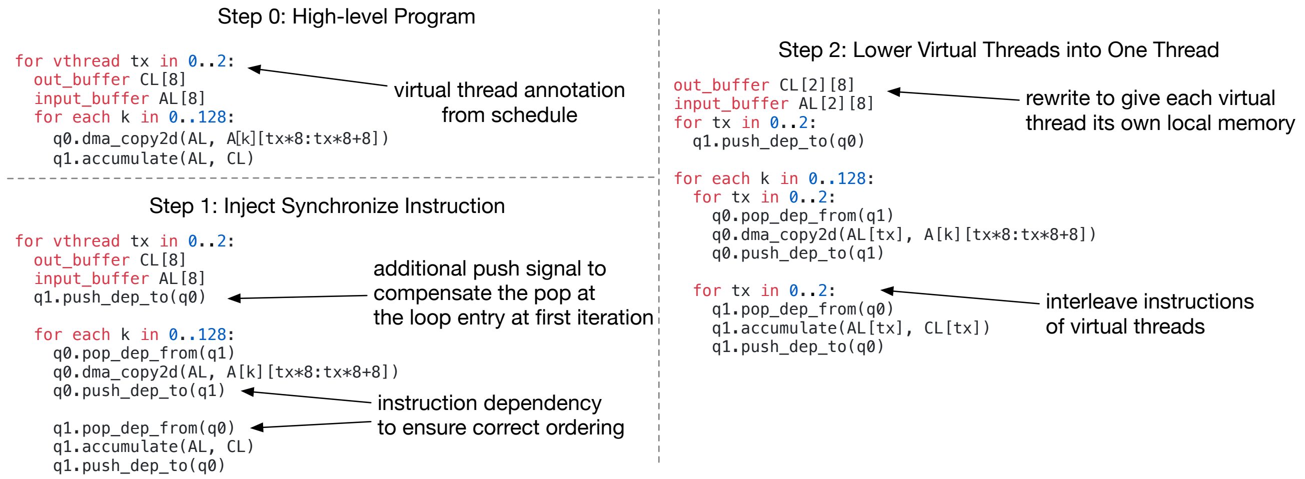


```
input_buffer AL[2][8]
out_buffer CL[2][8]

q1.push_dep_to(q0)
q1.push_dep_to(q0)
for each ko in 0..128:
◇q0.pop_dep_from(q1)
🟥q0.dma_copy2d(AL[0], A[k][0:8])
◇q0.push_dep_to(q1)
  q0.pop_dep_from(q1)
🟩q0.dma_copy2d(AL[1], A[k][8:16])
  q0.push_dep_to(q1)

◇q1.pop_dep_from(q0)
🔴q1.accumulate(AL[0], CL[0])
◇q1.push_dep_to(q0)
  q1.pop_dep_from(q0)
🟢q1.accumulate(AL[1], CL[1])
  q1.push_dep_to(q0)
. . .
```

Figure 16: Low-level program with explicit synchronization. This example demonstrates how to compute $C_j = \sum_k A_{k,j}$. The hardware contains two instruction streams that need to be explicitly synchronized. Carefully crafting the synchronization messages enables execution overlap between instruction streams.



Step 0: High-level Program

```
for vthread tx in 0..2:
  out_buffer CL[8]
  input_buffer AL[8]
  for each k in 0..128:
    q0.dma_copy2d(AL, A[k][tx*8:tx*8+8])
    q1.accumulate(AL, CL)
```

Step 1: Inject Synchronize Instruction

```
for vthread tx in 0..2:
  out_buffer CL[8]
  input_buffer AL[8]
  q1.push_dep_to(q0)

  for each k in 0..128:
    q0.pop_dep_from(q1)
    q0.dma_copy2d(AL, A[k][tx*8:tx*8+8])
    q0.push_dep_to(q1)

    q1.pop_dep_from(q0)
    q1.accumulate(AL, CL)
    q1.push_dep_to(q0)
```

additional push signal to compensate the pop at the loop entry at first iteration
instruction dependency to ensure correct ordering

Step 2: Lower Virtual Threads into One Thread

```
out_buffer CL[2][8]
input_buffer AL[2][8]
for tx in 0..2:
  q1.push_dep_to(q0)

for each k in 0..128:
  for tx in 0..2:
    q0.pop_dep_from(q1)
    q0.dma_copy2d(AL[tx], A[k][tx*8:tx*8+8])
    q0.push_dep_to(q1)

  for tx in 0..2:
    q1.pop_dep_from(q0)
    q1.accumulate(AL[tx], CL[tx])
    q1.push_dep_to(q0)
```

rewrite to give each virtual thread its own local memory
interleave instructions of virtual threads

Figure 17: Lowering procedure to transform a high-level parallel program with virtual threads to explicit synchronize program model.

architectures. The generated code of tensorized schedule aligns with common practices found in high performance computing: break complex operations down into a repeated sequence of micro-kernel calls. Consequently, we can also use the *tensorize* primitive to take advantage of handcrafted assembly micro-kernels, which can be beneficial in some platforms. For example, tensorizing half-precision GEMM to a 4×4 handcrafted micro-kernel on an AMD Vega GPU can yield more than $1.5\times$ speedup over the best non-tensorized version.

3.5 Compiler Support for Latency Hiding

Latency hiding refers to the process of overlapping memory operations with computation to maximize memory and compute utilization. It requires different strategies depending on the hardware back-end that is being targeted. On CPUs, memory latency hiding is achieved implicitly with simultaneous multithreading [11] or hardware prefetching techniques [16, 8]. GPUs rely on rapid context switching of many warps of threads to maximize the utilization of functional units [25]. Specialized deep learning accelerators, on the other hand, usually favor leaner control and offload this problem to the compiler stack.

Figure 16 demonstrates how a compiler is expected to explicitly handle data dependences on a pipelined deep learning accelerator that follows a decoupled-access/execute philosophy [22, 17]. We assume that the hardware pipeline consists of memory and compute stages that can execute concurrently where each stage is controlled by a independent instruction stream. Explicit synchronization

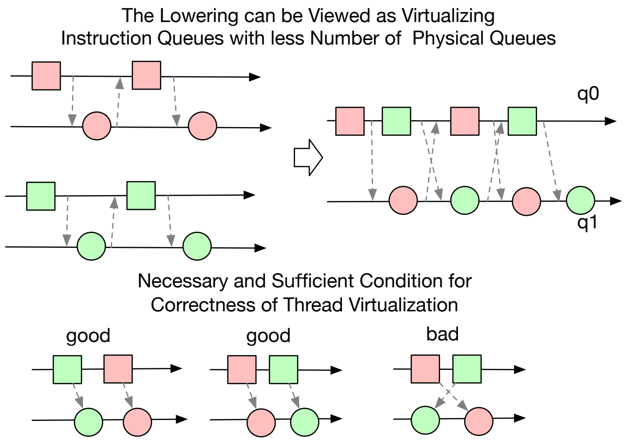

The Lowering can be Viewed as Virtualizing
Instruction Queues with less Number of Physical Queues
q0
q1
Necessary and Sufficient Condition for
Correctness of Thread Virtualization
good
good
bad

Figure 18: The lowering process can be viewed as virtualizing thread-local instruction streams onto a lesser amount of physical instruction streams in hardware. The lowering is correct as long as the ordering between each sender/receiver pair in the original virtual stream is preserved after lowering.

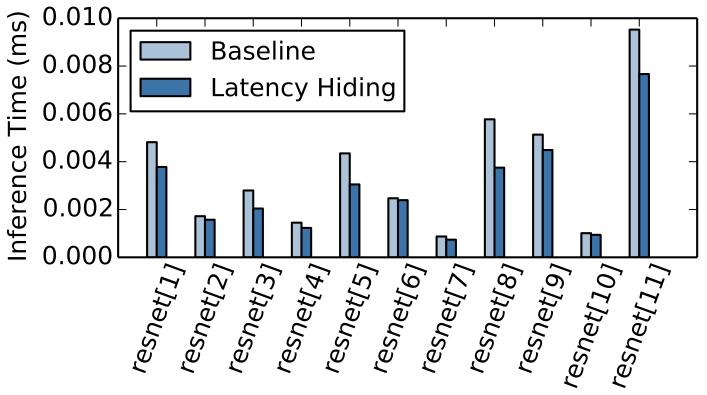


Figure 19: The effect of latency hiding on a FPGA-based hardware accelerator design running ResNet inference.

instructions are used to send a signal between pipeline stages to indicate when a given task is completed so that the next dependent stage can start to consume, or overwrite data. This type of explicit dependency tracking is implemented in hardware with FIFO queues, and gives direct control to the programmer and compiler over how hardware tasks execute. By using low-level synchronization primitives exposed by the hardware, we can effectively hide latency.

Programming hardware at such a low-level, with explicit synchronization operations, is a difficult and painstaking task. To reduce the burden on the programmer, we provide a virtual threading schedule primitive that lets the programmer specify a high-level data parallel program that TVM automatically lowers to a low-level explicit data dependence program. The lowering procedure is shown in Figure 17. The algorithm starts with a high-level parallel program and then inserts necessary synchronization instructions to guarantee correct execution order of the operations within each thread. Finally, the operations of all virtual threads are interleaved into one single thread.

Correctness of Virtual Thread Lowering An incorrect interleaving of virtual threads can lead to a deadlock. Conceptually, before lowering, each thread has its own private virtual instruction streams. The lowering process maps the instructions of these virtual instruction streams into the limited physical instruction streams. This procedure requires correctness guarantees, as the physical message queue between the physical instruction streams needs to be shared among threads to pass dependencies. We can prove that the lowering is correct as long as ordering is preserved according to the rules displayed in Figure 18. The following theorem gives a necessary and sufficient condition for the correctness of this lowering process.

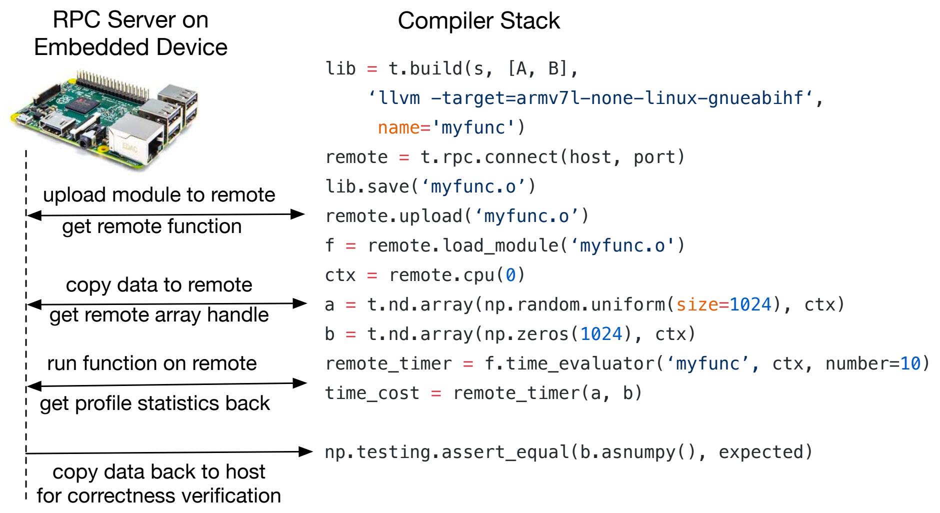
Theorem 3.1. *Let $<$ be the partial order of the instructions after lowering, $S = \{(x_i, y_i)\}$ be set of all pairs of push pop instructions before lowering. Every push message sent by the sender gets received by the corresponding pop instruction (lowering is correct), if and only if*

$$\begin{aligned}(x' > x) \rightarrow (y' > y), (x' < x) \rightarrow (y' < y)\\ \forall (x, y), (x', y') \in S\end{aligned} \tag{1}$$

In other words, the relative order of receiver of the synchronization message need to be the same as its sender for each send–receive pair. demonstrates this condition.

Proof. Proof by contradiction. Let a be the first sender in a physical queue to send its message to wrong receiver d. Then $\exists (a, b), (c, d) \in S$.

- $a < c$ since a is the first sender who sent the wrong message.
- $b < d$ because of the theorem condition



```
lib = t.build(s, [A, B],
    ‘llvm -target=armv7l-none-linux-gnueabihf‘,
     name='myfunc')
remote = t.rpc.connect(host, port)
lib.save(‘myfunc.o’)
remote.upload(‘myfunc.o’)
f = remote.load_module(‘myfunc.o')
ctx = remote.cpu(0)
a = t.nd.array(np.random.uniform(size=1024), ctx)
b = t.nd.array(np.zeros(1024), ctx)
remote_timer = f.time_evaluator(‘myfunc’, ctx, number=10)
time_cost = remote_timer(a, b)

np.testing.assert_equal(b.asnumpy(), expected)
```

Figure 20: RPC support in TVM runtime to make it easy to cross compile, deploy, and profile embedded devices.

- This means b pops a message from the queue before d from some sender h, and $h < a$ due to the FIFO property of message queue.
- This contradicts the fact that a is the first sender in the queue to send to the wrong receiver (h also send to wrong receiver).

$\square$

This condition can be enforced by iterating through the virtual threads in round-robin fashion when each virtual instruction stream is lowered down to a physical instruction stream. This condition not only applies to the case of two physical instruction streams, but can also be used to virtualize the threads into an arbitrary number of physical instruction streams.

Hardware Evaluation of Latency Hiding We demonstrate the effectiveness of latency hiding on a custom hardware accelerator design which we describe in more depth in subsection 5.3. We run each layer of ResNet on the accelerator, and use TVM to generate two schedules: one with latency hiding, and one without. The schedule with latency hiding applies virtual threading on the program to hide memory access latencies. The results are shown in Figure 19. Overall latency hiding achieves anywhere between 7% up to 54% latency reduction on each kernel by hiding some of the latency of loading data into the accelerator. In terms of compute resources, no latency hiding leads to at best 52% utilization, whereas latency hiding yields up to 74% utilization.

4 Code Generation and Runtime Support

After optimizing the schedule, the remaining task is to generate code that can run on the target platform and facilitate the deployment and profiling of the generated kernels.

Code Generation For a specific tuple of data-flow declaration, axis relation hyper-graph, and schedule tree, we can generate lowered code by iteratively traversing the schedule tree, inferring the dependent bounds of the input tensors (using the axis relation hyper-graph), and generating the loop nest in the low-level code. The code is lowered to an in-memory representation of an imperative C style loop program. We reuse passes from Halide for common lowering primitives like storage flattening and unrolling, and add GPU/accelerator-specific transformations such as synchronization point detection, virtual thread injection, and module generation. Finally, the loop program is transformed into LLVM or CUDA/Metal/OpenCL source code.

Runtime Support For GPU programs, we build the host and device modules separately and provide a runtime module system to launch kernels using their corresponding driver APIs. We build a driver library for the FPGA-based deep learning accelerator used in our evaluation with a C runtime API that can construct instructions and push them to the target accelerator for execution. Our code generation algorithm then translates the accelerator program to a series of calls into the runtime API.

H/W	IC	OC	K	S	NNPack (ms)	TVM (ms)	**Speedup**
224	3	64	7	2	44.06	21.93	2.01
56	64	64	3	1	24.42	21.45	1.14
56	64	64	1	1	7.69	3.56	2.16
56	64	128	3	2	16.06	12.62	1.27
56	64	128	1	2	3.09	1.86	1.66
28	128	128	3	1	28.78	21.36	1.35
28	128	256	3	2	14.47	15.36	0.94
28	128	256	1	2	2.2	2.15	1.03
14	256	256	3	1	56.44	23.48	2.4
14	256	512	3	2	23.22	28.56	0.81
14	256	512	1	2	3.37	2.54	1.33
7	512	512	3	1	179.81	62.48	2.88

Table 1: Comparison of TVM and NNPack on convolution operators of ResNet on Raspberry Pi. H/W for height and width, IC for input channels, OC for output channels, K for kernel size, S for stride size.

Autotuning This paper focuses on providing a novel optimization framework to enable performance-competitive compilation for deep learning systems. The optimizations that we present give the compiler a large space of schedules to explore, in order to generate performance-competitive code. We explore preliminary automatic scheduling techniques and a pass to automatically inline the injective operations. Complex operations such as high dimensional convolution, matrix multiplication, and depthwise convolution are autotuned with a collection of schedule templates. We believe combining our optimizations with more sophisticated search techniques [20] can yield further improvements in the future.

Remote Deployment Profiling TVM also includes infrastructure to make profiling and autotuning easier on embedded devices. Traditionally, targeting an embedded device for tuning requires cross-compiling on the host side, copying to the target device, and timing the execution. This process has to be done manually and prevents the automation of the compile, run, and profile autotuning flow. We provide remote function call support in our compiler stack: through the RPC interface, we can compile the program on a host compiler, upload to remote embedded devices, run the function remotely, and access the results in the same script on the host (Figure 20). We find this approach to be very helpful in optimizing workloads on embedded devices such as the Raspberry Pi and FPGA-based accelerators.

5 Evaluation

We evaluate TVM on three types of platforms—an embedded CPU, a server-class GPU, and a deep learning accelerator implemented on a low-power FPGA based SoC. The benchmarks are based on real world deep learning inference workloads including ResNet [12] and MobileNet [15]. We compare our approach with existing deep learning frameworks including MxNet [7] and TensorFlow [2] that rely on highly engineered vendor-specific libraries.

5.1 Raspberry Pi Evaluation

We evaluated the performance of TVM on a Raspberry Pi 3B (Quad Core 1.2GHz). We use Apache MXNet (commit: 35ceea) as our baseline system. MXNet uses state-of-the-art operators in NNPack (commit: 57fc2c) for convolution and OpenBLAS for matrix multiplication. We carefully tuned the baseline for the best performance. We also enabled Winograd convolution in the NNPack implementation for 3x3 convolutions and multithreading.

Table 1 and Table 2 show the comparison between TVM tensor operators against hand-optimized ones for ResNet and MobileNet. We observe that TVM generates operators that outperform the hand-optimized NNPack version for both neural network workloads. The result also demonstrates TVM 's ability to quickly optimize emerging tensor operators, such as depthwise [15] convolution operators which are not supported in existing DNN libraries. Finally, Table 3 shows an end-to-end comparison of the two network workloads, where TVM outperforms baselines by 2x and 12x for ResNet and MobileNet respectively.

H/W	C	M	K	S	MXKernel (ms)	TVM (ms)	**Speedup**
112	32	1	3	1	35.52	4.33	8.21
112	64	1	3	2	20.83	3.49	5.97
56	128	1	3	1	35.54	4.17	8.53
56	128	1	3	2	10.66	1.68	6.34
28	256	1	3	1	19.43	2.13	9.13
28	256	1	3	2	6.81	0.9	7.59
14	512	1	3	1	12.07	1.43	8.43
14	512	1	3	2	4.1	0.44	9.39
7	1024	1	3	1	7.48	0.99	7.57

Table 2: Comparison of TVM and MXKernel (native operator in MXNet) on depthwise convolution operators of MobileNet on Raspberry Pi. H/W for height and width, C for channels, M for multiplier, K for kernel size, S for stride size. Because depthwise convolution architecture is relatively new and is not yet optimized by a DNN library, we pick the naive implementation found in the MXNet. TVM generates operator that out-performs the implementation in MXNet.

Workload	MXNet(ms)	TVM (ms)	**Speedup**
ResNet18	1390	567	2.4
MobileNet	2862	209	12

Table 3: End-to-End Experiment Results on Raspberry Pi

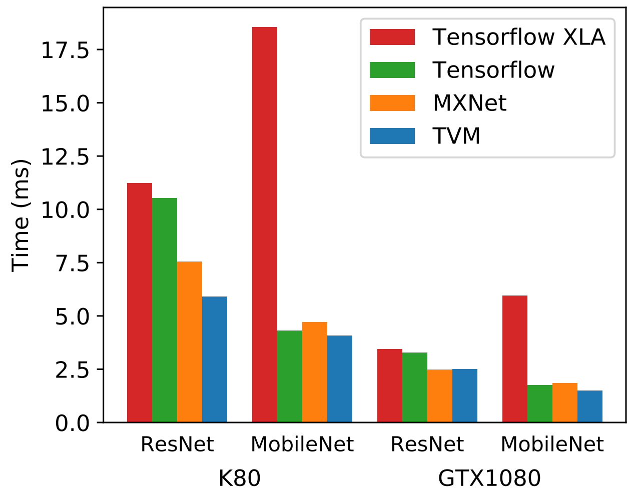


Figure 21: GPU end-to-end comparison of ResNet and MobileNet workloads among TVM , MXNet, Tensorflow, and Tensorflow XLA on NVIDIA Tesla K80 and GTX 1080.

5.2 GPU Evaluation

We first compare the end-to-end performance of the ResNet and MobileNet workloads among TVM , MXNet, Tensorflow, and Tensorflow XLA on the NVIDIA Tesla K80 and GTX 1080 GPUs. MXNet and Tensorflow both use cuBLAS v8 and cuDNN v7 for the ResNet workload, and implement their own versions of depthwise convolution in the MobileNet workload as this operation is relatively new and is not yet supported by the latest libraries. On the other hand, Tensorflow XLA uses JIT compilation. Figure 21 shows that TVM outperforms other frameworks in end-to-end execution time under almost all configurations except for the ResNet workload on GTX 1080 where TVM is marginally slower (1.6%) than MXNet. TVM reduces the inference time by 22% and 5% over the second best implementations of ResNet and MobileNet on the Tesla K80, and is 7.7% faster for MobileNet on GTX 1080.

We perform a breakdown comparison of stages in Resnet and MobileNet on Nvidia Tesla K80, shown in Table 4 and Table 5. We can see that TVM generates competitive GPU kernels compared to the vendor-specific ones one. This is because TVM has a large

H/W	IC	OC	K	S	cuDNN (ms)	TVM (ms)	**Speedup**
224	3	64	7	2	0.44	0.45	0.98
56	64	64	3	1	0.42	0.47	0.90
56	64	64	1	1	0.23	0.06	3.74
56	64	128	3	2	0.32	0.38	0.86
56	64	128	1	2	0.21	0.10	2.08
28	128	128	3	1	0.44	0.41	1.08
28	128	256	3	2	0.40	0.37	1.08
28	128	256	1	2	0.22	0.08	2.67
14	256	256	3	1	0.56	0.56	1.01
14	256	512	3	2	0.56	0.47	1.20
14	256	512	1	2	0.24	0.10	2.47
7	512	512	3	1	0.75	0.55	1.36

Table 4: Comparison of TVM and cuDNN v7 on Conv2D operators of ResNet on Nvidia Tesla K80. H/W for height and width, IC for input channels, OC for output channels, K for kernel size, S for stride size.

H/W	C	M	K	S	MXKernel (ms)	TVM (ms)	**Speedup**
112	32	1	3	1	0.34	0.12	2.91
112	64	1	3	2	0.26	0.34	0.75
56	128	1	3	1	0.36	0.18	1.98
56	128	1	3	2	0.21	0.17	1.25
28	256	1	3	1	0.20	0.07	2.71
28	256	1	3	2	0.17	0.05	3.65
14	512	1	3	1	0.17	0.04	4.72
14	512	1	3	2	0.17	0.03	5.45
7	1024	1	3	1	0.16	0.03	5.06

Table 5: Comparison of TVM and MXNet on depthwise Conv2D operators on MobileNet running on Nvidia Tesla K80. H/W for height and width, C for channels, M for multiplier, K for kernel size, S for stride size.

design space to search through to generate optimized schedules. More importantly, for new operations such as depthwise convolutions (Table 5), TVM can generate well-optimized GPU kernels.

AMD GPU Case Study To highlight the ability of TVM to target multiple GPU backends, we experimented with porting existing schedules to an AMD Vega FE Air (`gfx900`). When starting from a default GPU schedule template, we are within an order of magnitude of GTX 1080 performance, achieving 10.2 ms and 3.3 ms end-to-end times for ResNet and MobileNet respectively. Quickly tuning the schedule in a matter of hours further improved ResNet performance by roughly 40% and MobileNet performance by roughly 10%. These results highlight that despite AMD and NVIDIA GPUs having ostensibly different architectures with different instructions, TVM is general enough to capture their similarities: mainly in their memory hierarchy and organization of threads. Still, we expect further performance improvements with AMD GPU-specific structural changes to the default schedule.

5.3 FPGA Accelerator Evaluation

Vanilla Inference Tensor Accelerator We demonstrate how TVM tackles accelerator-specific code generation on a generic inference accelerator design we prototyped on an FPGA. We introduce the Vanilla Inference Tensor Accelerator (VITA) which distills characteristics from previous accelerator proposals [19, 9, 17] into a minimalist hardware architecture. We use VITA in this evaluation to demonstrate TVM 's ability to generate highly efficient schedules that can target specialized accelerators. Figure 22 shows the high-level hardware organization of the VITA architecture. VITA is programmed as a tensor processor to efficiently execute operations with high compute intensity (e.g, matrix multiplication, high dimensional convolution). VITA can perform load/store operations to bring blocked 3-dimensional tensors from DRAM to into a contiguous region of SRAM. VITA also provides specialized on-chip memories for network parameters, layer inputs (narrow data type), and layer outputs (wide data type). Finally, VITA provides explicit synchronization control over successive load, compute, and store operations to maximize the overlap between memory and compute operations.

Methodology We implement the VITA design on a low-power PYNQ board which incorporates an ARM Cortex A7 dual core CPU clocked at 667MHz and an Artix-7 based FPGA fabric. On the modest FPGA resources, we implement a 16×16 matrix-vector unit clocked at 200MHz that performs products of 8-bit values and accumulates them into a 32-bit register every cycle. The theoretical peak throughput of this flavor of the VITA design lies around 102.4GOPS/s. We allocate 32kB of resources for activation storage, 32kB for parameter storage, 32kB for microcode buffers, and 128kB for the register file. These on-chip buffers are nowhere near large enough to provide enough on-chip storage for a single layer of ResNet, and therefore enable a case study on effective memory reuse and latency hiding.

End-to-end ResNet Evaluation We leverage TVM to generate ResNet inference kernels on the PYNQ platform and offload as many layers as possible to VITA. We utilize TVM to generate both schedules for the CPU only and CPU+FPGA implementation. Due to its

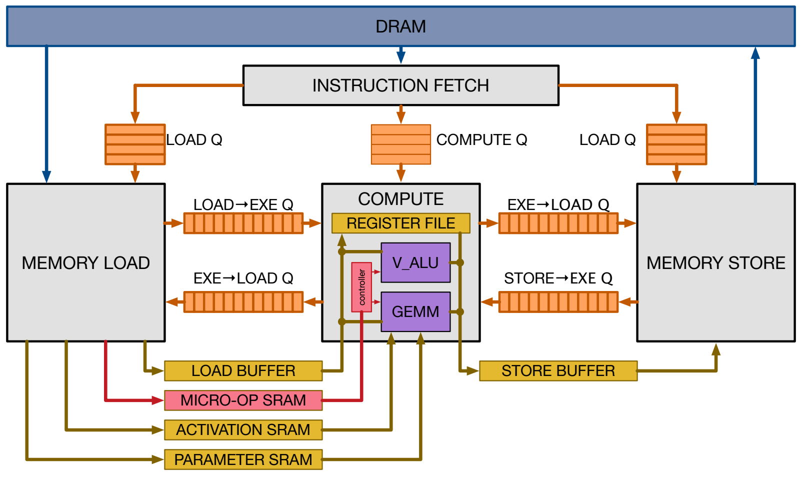


Figure 22: VITA Hardware Design Overview.

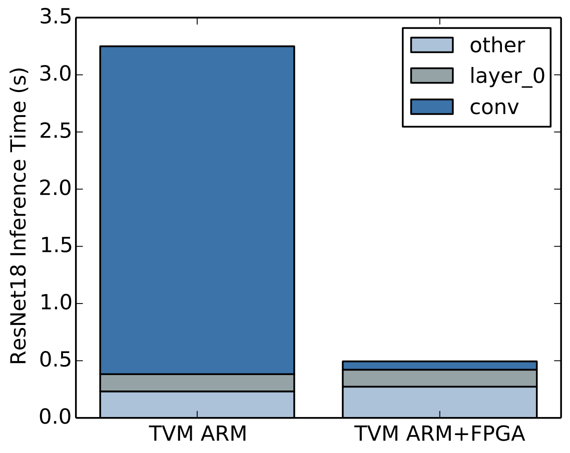


Figure 23: We offload convolutions in the ResNet workload to an FPGA-based accelerator. The grayed-out bars correspond to layers that cannot be accelerated by the FPGA and therefore have to run on the CPU. The FPGA can provide a 40x acceleration on offloaded convolution layers over the ARM Cortex A9.

shallow convolution depth, the first ResNet convolution layer could not be efficiently offloaded on the FPGA and is instead computed on the CPU. All other convolution layers in ResNet, however are amenable to efficient offloading. Operations like residual layers, batch normalization, and activations are also performed on the CPU since VITA does not support these operations.

Figure 23 shows the ResNet inference time breakdown between the CPU-only execution and the CPU+FPGA execution. Most of the computation is spent on the convolution layers that can be offloaded to VITA. For those convolution layers, the achieved speedup is 40×. Unfortunately, the overall performance of the FPGA accelerated system is bottlenecked by the sections of the workload that have to be executed on the CPU due to Amdahl's law. We envision that extending the VITA design to incorporate support for these other operators will help reduce inference time even further. This FPGA-based experiment showcases TVM's ability to adapt to new architectures and the hardware intrinsics that they expose.

6 Related Work

Deep learning frameworks [3, 6, 7, 4] provide convenient interfaces for users to express deep learning workloads, and deploy them easily on different hardware back-ends. While existing frameworks currently depend on vendor specific tensor operator libraries to execute their workloads, they can leverage TVM 's stack to generate optimized code a larger hardware devices.

High-level computation graph DSLs are a typical way to represent and perform high-level optimizations. Tensorflow's XLA [3] and the recently introduced DLVM [26] fall into this category. The representations of computation graphs in these works are similar, and a high-level computation graph DSL is also used in this paper. While graph level representations are a good fit for high-level optimizations, they are too high-level to optimize tensor operators under a diverse set of hardware back-ends. Prior work relies on specific lowering rules to directly generate low-level LLVM or resorts to vendor crafted libraries. These approaches require significant engineering effort for each hardware back-end and operator-variant combination.

Halide [21] introduced the principle of separation between compute and scheduling. We adopt Halide's insight and reuse its existing useful scheduling primitives in our compiler. The tensor operator scheduling is also related other works on DSL for GPUs [23, 14] as well as works on polyhedral-based loop transformation [5, 24]. TACO [18] introduces a generic way to generate sparse tensor operators on CPU. We specifically focus on solving the new scheduling challenges of deep learning workloads for GPUs and specialized accelerators. Our scheduling primitives can be potentially adopted to the optimization pipeline in these works. More importantly, we provide an end-to-end stack that can directly take descriptions from deep learning frameworks, and jointly optimize together with the high-level stack.

Despite the trend domain specific accelerators for deep learning [17, 10], it is yet unclear how a compilation stack can be built to effectively target these devices. The VITA design used in the evaluation provides a generic way to summarize the properties of these accelerators, and enables a concrete case study on how such compilation to accelerators can be done. This paper provides a generic solution to effectively target the specialized accelerators via tensorization and compiler-driven latency hiding.

7 Acknowledgements

We would like to thank Ras Bodik and James Bornholt for their thorough feedback on earlier versions of this paper. This work was partially supported by the NSF under grant #1518703.

8 Conclusion

Our work provides an end-to-end stack to solve fundamental optimization challenges across a diverse set of hardware back-ends. We hope our work can encourage more studies of programming languages, compilation, and open new opportunities for hardware co-design techniques for deep learning systems. We plan to open-source our compiler stack and the VITA design to encourage further researches in this direction.

References

[1] Nvidia tesla v100 gpu architecture: The worldś most advanced data center gpu, 2017.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[4] Amit Agarwal, Eldar Akchurin, Chris Basoglu, Guoguo Chen, Scott Cyphers, Jasha Droppo, Adam Eversole, Brian Guenter, Mark Hillebrand, Ryan Hoens, Xuedong Huang, Zhiheng Huang, Vladimir Ivanov, Alexey Kamenev, Philipp Kranen, Oleksii Kuchaiev, Wolfgang Manousek, Avner May, Bhaskar Mitra, Olivier Nano, Gaizka Navarro, Alexey Orlov, Marko Padmilac, Hari Parthasarathi, Baolin Peng, Alexey Reznichenko, Frank Seide, Michael L. Seltzer, Malcolm Slaney, Andreas Stolcke, Yongqiang Wang, Huaming Wang, Kaisheng Yao, Dong Yu, Yu Zhang, and Geoffrey Zweig. An introduction to computational networks and the computational network toolkit. Technical Report MSR-TR-2014-112, August 2014.

[5] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F. Donaldson, Jeroen Ketema, Javed Absar, Sven van Haastregt, Alexey Kravets, Anton Lokhmotov, Robert David, and Elnar Hajiyev. Pencil: A platform-neutral compute intermediate language for accelerator programming. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, pages 138–149, Washington, DC, USA, 2015. IEEE Computer Society.

[6] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[7] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, , and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *Neural Information Processing Systems, Workshop on Machine Learning Systems (LearningSys'15)*, 2015.

[8] Tien-Fu Chen and Jean-Loup Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[9] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 367–379, Piscataway, NJ, USA, 2016. IEEE Press.

[10] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 609–622, Washington, DC, USA, 2014. IEEE Computer Society.

[11] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: a platform for next-generation processors. *IEEE Micro*, 17(5):12–19, Sept 1997.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. *arXiv preprint arXiv:1603.05027*, 2016.

[13] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. Darkroom: Compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph.*, 33(4):144:1–144:11, July 2014.

[14] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 556–571, New York, NY, USA, 2017. ACM.

[15] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[16] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[17] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM.

[18] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, October 2017.

[19] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. Pudiannao: A polyvalent machine learning accelerator. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 369–381, New York, NY, USA, 2015. ACM.

[20] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.*, 35(4):83:1–83:11, July 2016.

[21] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[22] James E. Smith. Decoupled access/execute computer architectures. In *Proceedings of the 9th Annual Symposium on Computer Architecture*, ISCA '82, pages 112–119, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

[23] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Lift: A functional data-parallel ir for high-performance gpu code generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO ’17, pages 74–85, Piscataway, NJ, USA, 2017. IEEE Press.

[24] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, January 2013.

[25] Vasily Volkov. *Understanding Latency Hiding on GPUs*. PhD thesis, University of California at Berkeley, 2016.

[26] Richard Wei, Vikram Adve, and Lane Schwartz. Dlvm: A modern compiler infrastructure for deep learning systems. *CoRR*, abs/1711.03016, 2017.