Halide

Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines

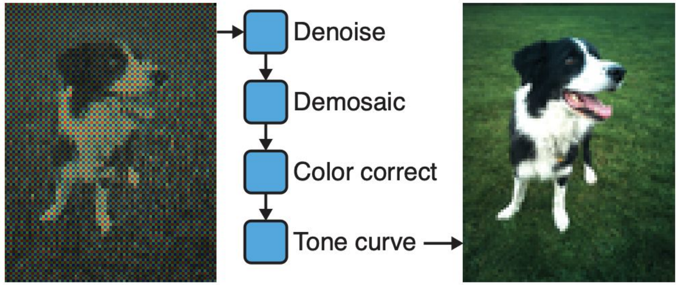
Paper presentaion for 15-849
Bohan Hou

Overview

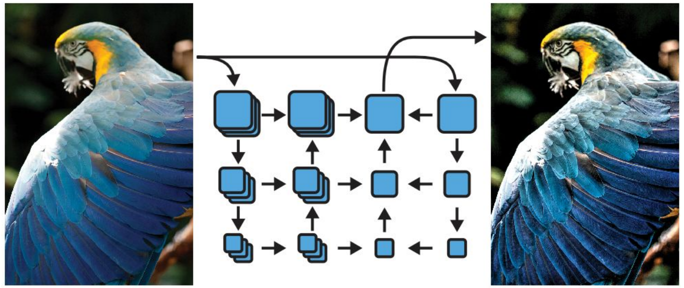
- Image Processing Pipeline Optimization
    - Multiple optimization techniques
    - Engineering Complexity

- Key idea: decouple algorithm from schedule
    - algorithm: what is computed
    - schedule: when and where to compute

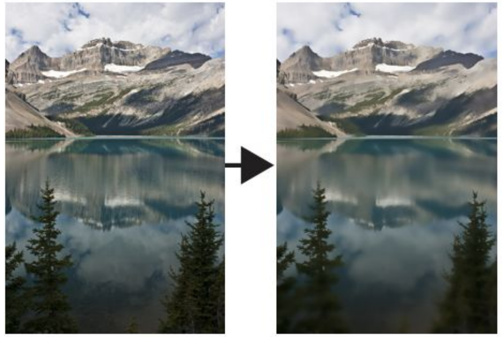
Problem: Image Processing Pipeline Optimization



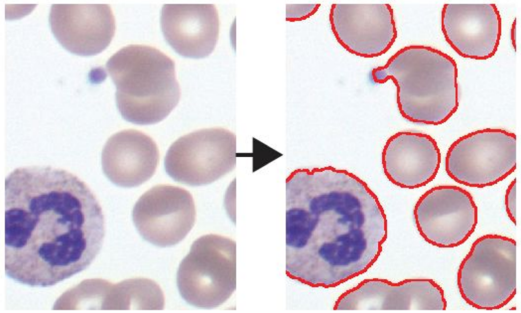
Camera Raw Pipeline



Local Laplacian Filter



Bilateral Grid

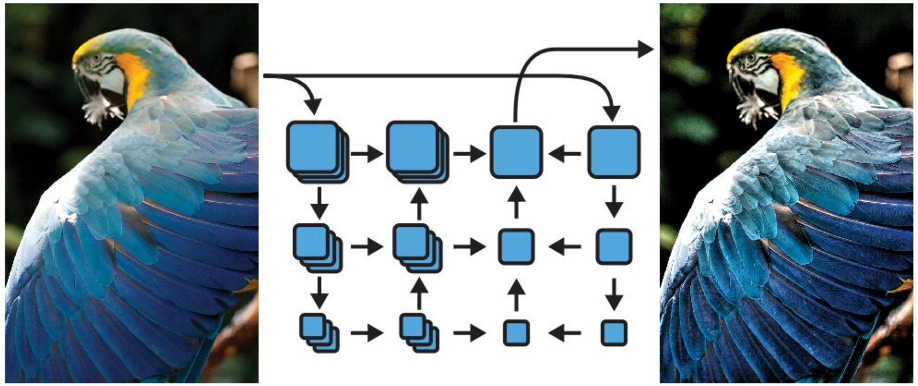


Snake Image Segmentation

Challenge: Engineering Complexity

Example: Local Laplacian Filters

- 1500 lines of expert-optimized C++
- Multi-threaded, SSE SIMD insts
- 3 months of work
- 10x faster than reference C code



Used in Adobe PhotoShop
Camara Raw / Lightroom

Note: data from https://halide-lang.org/assets/lectures/Halide_CVPR_intro.pdf

Challenge: Engineering Complexity

Example: 3x3 blur

(a) Clean C++ : 9.94 ms per megapixel

```
void blur(const Image &in, Image &blurred) {
 Image tmp(in.width(), in.height());

 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;
}
```

Techniques:

- Tiling
- Fusing
- Vectorization
- Multi-threading
- Redundant Computation

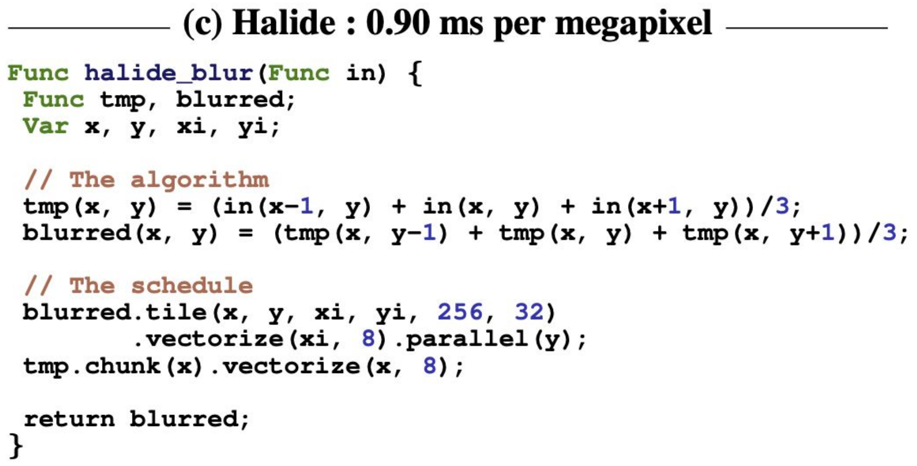
Lacks readability, portablity, modularity

(b) Fast C++ (for x86) : 0.90 ms per megapixel

```
void fast_blur(const Image &in, Image &blurred) {
 __m128i one_third = _mm_set1_epi16(21846);
 #pragma omp parallel for
 for (int yTile = 0; yTile < in.height(); yTile += 32) {
  __m128i a, b, c, sum, avg;
  __m128i tmp[(256/8)*(32+2)];
  for (int xTile = 0; xTile < in.width(); xTile += 256) {
   __m128i *tmpPtr = tmp;
   for (int y = -1; y < 32+1; y++) {
    const uint16_t *inPtr = &(in(xTile, yTile+y));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_loadu_si128((__m128i*)(inPtr-1));
     b = _mm_loadu_si128((__m128i*)(inPtr+1));
     c = _mm_load_si128((__m128i*)(inPtr));
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(tmpPtr++, avg);
     inPtr += 8;
   }}
   tmpPtr = tmp;
   for (int y = 0; y < 32; y++) {
    __m128i *outPtr = (__m128i *)(&(blurred(xTile, yTile+y)));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_load_si128(tmpPtr+(2*256)/8);
     b = _mm_load_si128(tmpPtr+256/8);
     c = _mm_load_si128(tmpPtr++);
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(outPtr++, avg);
}}}}}
```

11x faster on CPU

Method: Decouple Algorithm from Schedule



(c) Halide : 0.90 ms per megapixel

```
Func halide_blur(Func in) {
 Func tmp, blurred;
 Var x, y, xi, yi;

 // The algorithm
 tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
 blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;

 // The schedule
 blurred.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
 tmp.chunk(x).vectorize(x, 8);

 return blurred;
}
```

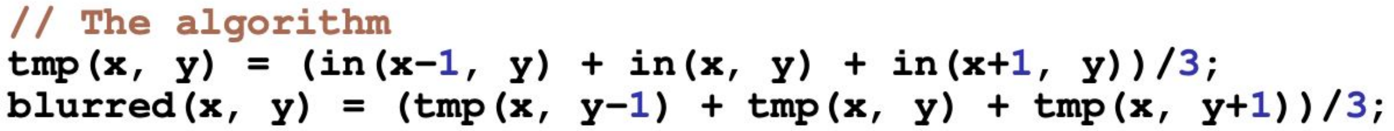
Algorithm: *what* is computed

Schedule: *when* are *where* it's computed

- Easy for programmers to build pipelines
- Easy for programmers to specify & explore optimizations
- Easy for compilers to generate fast code

Algorithm: Pure functional

- Declarative specification
- Pipeline stages are pure functions from coordinates to values
- No explicit bounds
- No loops or traversal orders
- Only feed forward pipelines



```
// The algorithm
tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;
```

Algorithm of 3x3 blur

Algorithm: Pure functional

Reduction functions

- initial value function: specify the initial value of each output point
- reduction function: specify the update rules of output points
- reduction domain: specify the reduction order

```
UniformImage in(UInt(8), 2);
Func histogram, cdf, out;
RDom r(0, in.width(), 0, in.height()), ri(0, 255);
Var x, y, i;

histogram(in(r.x, r.y))++;
cdf(i) = 0;
cdf(ri) = cdf(ri-1) + histogram(ri);
out(x, y) = cdf(in(x, y));
```

Algorithm of histograms

Schedule

Schedule defines intra-stage order, inter-stage interleaving

For each stage:

1) In which order should we compute its values?
2) When should we compute its inputs?
3) How to map onto parallel excution resources like SIMD units and GPU blocks?

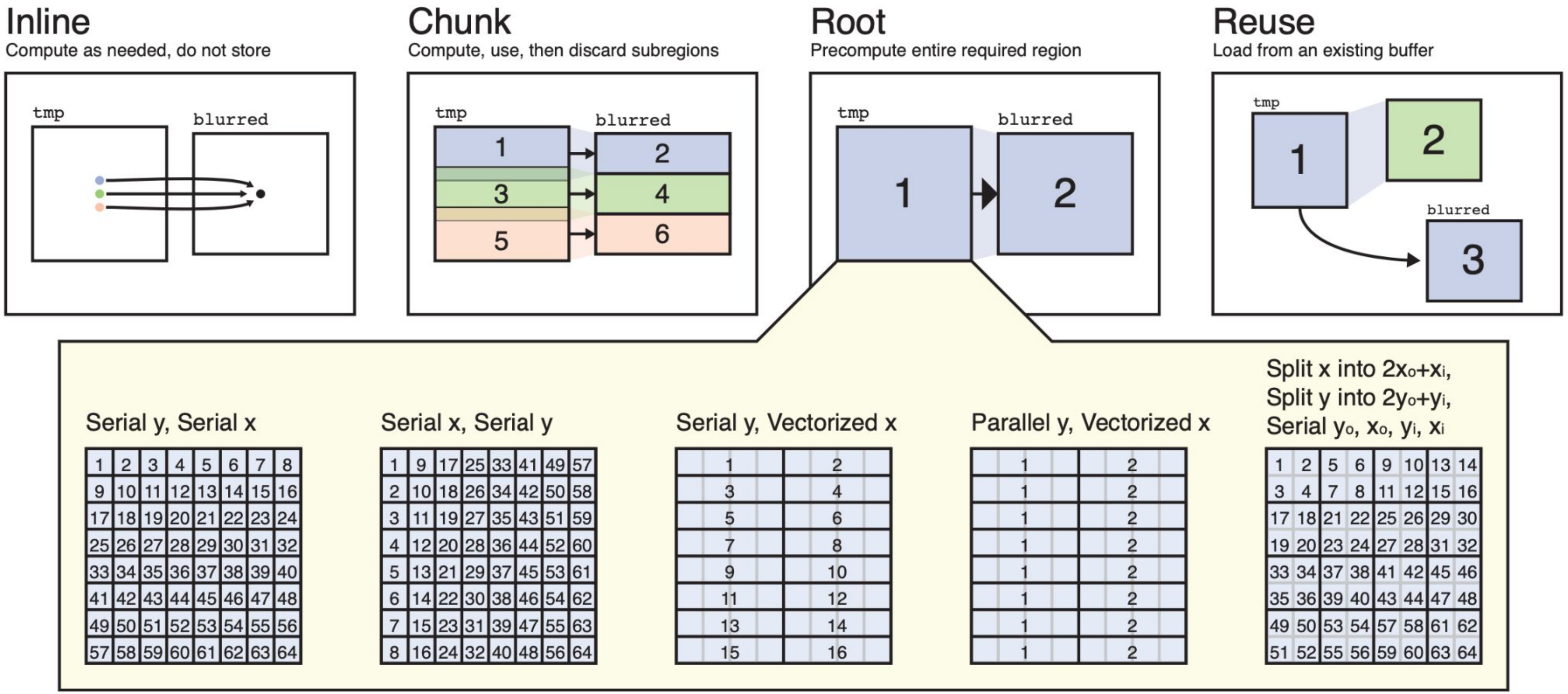
```
// The schedule
blurred.tile(x, y, xi, yi, 256, 32)
       .vectorize(xi, 8).parallel(y);
tmp.chunk(x).vectorize(x, 8);
```

▭ traversal order

▭ parallel execution

▭ producer consumer relation

Schedule: intra-stage order, inter-stage interleaving



Serial y, Serial x

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

Serial x, Serial y

1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63
8	16	24	32	40	48	56	64

Serial y, Vectorized x

1	2
3	4
5	6
7	8
9	10
11	12
13	14
15	16

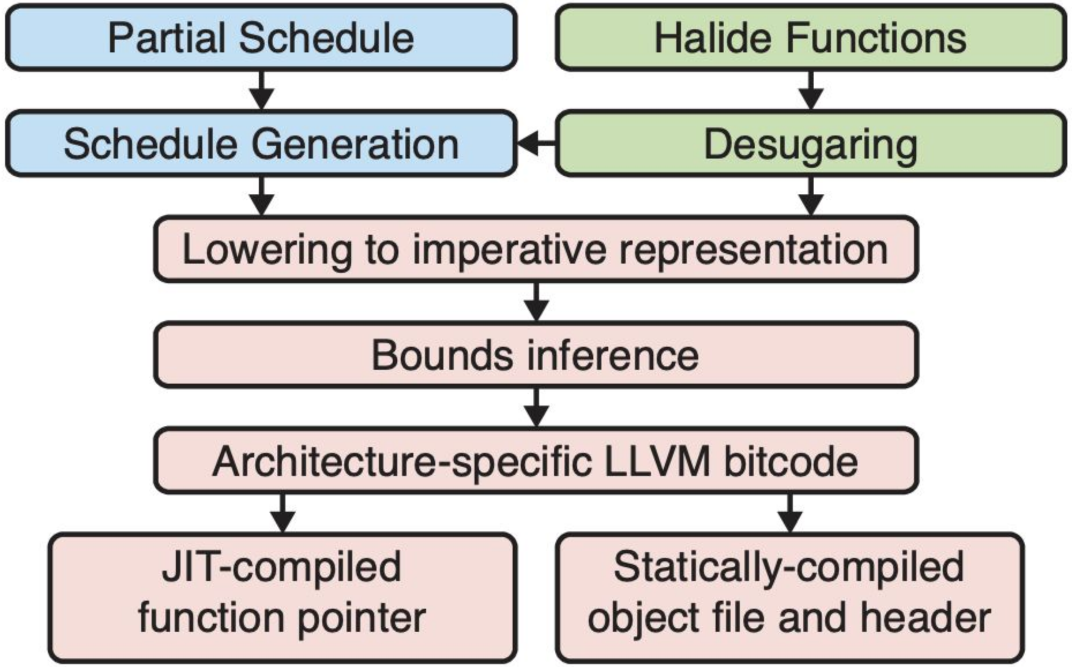
Parallel y, Vectorized x

1	2
1	2
1	2
1	2
1	2
1	2
1	2
1	2

Split x into $2x_o+x_i$, Split y into $2y_o+y_i$, Serial y_o, x_o, y_i, x_i

1	2	5	6	9	10	13	14
3	4	7	8	11	12	15	16
17	18	21	22	25	26	29	30
19	20	23	24	27	28	31	32
33	34	37	38	41	42	45	46
35	36	39	40	43	44	47	48
49	50	53	54	57	58	61	62
51	52	55	56	59	60	63	64

Compiler Implementation


Partial Schedule
Halide Functions
Schedule Generation
Desugaring
Lowering to imperative representation
Bounds inference
Architecture-specific LLVM bitcode
JIT-compiled function pointer
Statically-compiled object file and header

Lowering:

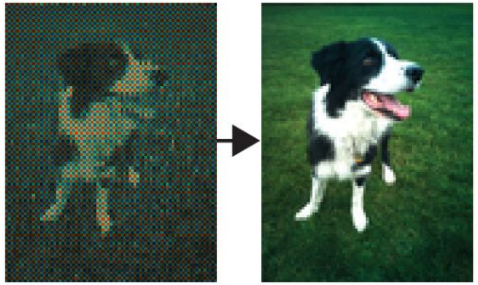
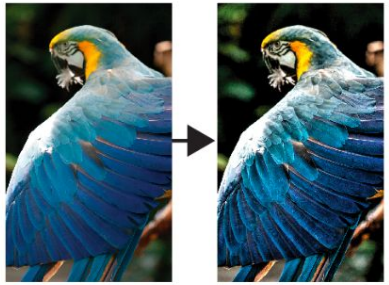
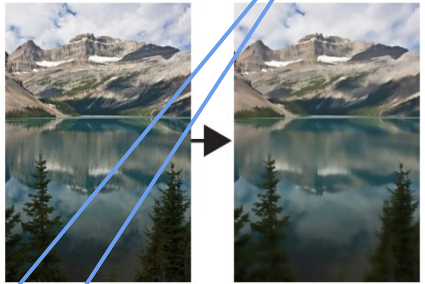
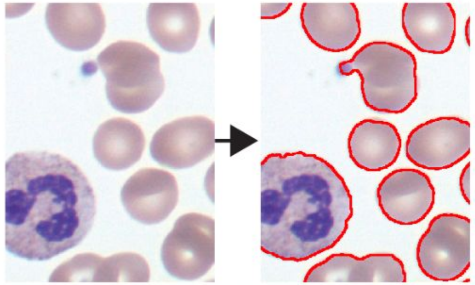
- Generate loop nests
- Allocate storage for function realizations

Bounds inference:

- Replace function call with load/store to the storages

Evaluation

Reduced engineering effort
Better Readability
Better Performance

Camera Raw Pipeline	**Local Laplacian Filter**	**Bilateral Grid**	***Snake* Image Segmentation**
Optimized NEON ASM: 463 lines	**C++, OpenMP+IPP:** 262 lines	**Tuned C++:** 122 lines	**Vectorized MATLAB:** 67 lines
Nokia N900: 772 ms	Quad-core x86: 335 ms	Quad-core x86: 472ms	Quad-core x86: 3800 ms
Halide algorithm: 145 lines	**Halide algorithm:** 62 lines	**Halide algorithm:** 34 lines	**Halide algorithm:** 148 lines
schedule: 23 lines	**schedule:** 7 lines	**schedule:** 6 lines	**schedule:** 7 lines
Nokia N900: 741 ms	Quad-core x86: 158 ms	Quad-core x86: 80 ms	Quad-core x86: 55 ms
2.75x shorter	3.7x shorter	3x shorter	2.2x longer
5% *faster* than tuned assembly	2.1x faster	5.9x faster	70x faster

Porting to new platforms does not change the algorithm code, only the schedule

Quad-core x86: 51 ms	CUDA GPU: 48 ms (7x)	CUDA GPU: 11 ms (42x)	CUDA GPU: 3 ms (1250x)
		Hand-written CUDA: 23 ms [Chen et al. 2007]	

Portability

Thanks

Discussion Questions

- Can we design a framwork to automatically schedule a given algorithm?

- Is Halide language Turing complete? Is there any useful operator that can not be expressed by Halide? How can we improve it then?

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

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, Arvind Krishnamurthy

Presented by Ashwin Venkatram

7 February 2022

Contents

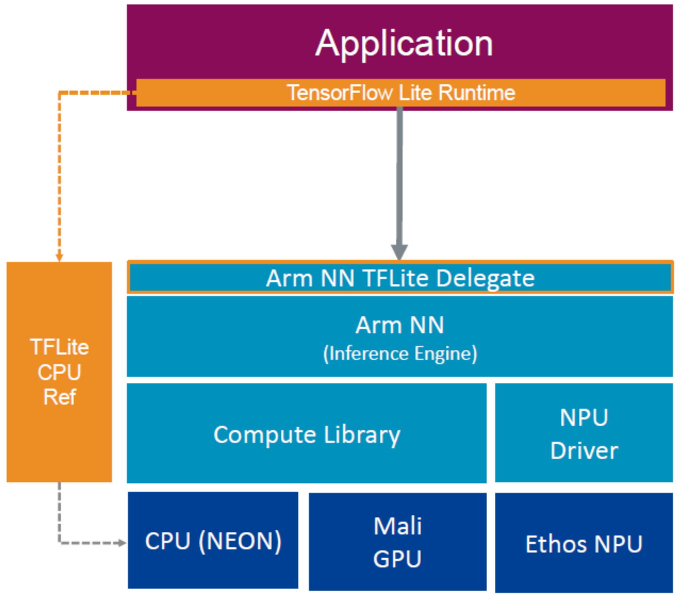
- Motivation for DL Compilers & Landscape of HW accelerators
- DL Compiler System Architecture
  - Front-end IR
  - Back-end IR
- TVM Performance
- Discussion Questions

Motivation for DL Compilers & Current State

Case Study: Real-time Anomaly Detection

Arm NN TFLite Delegate

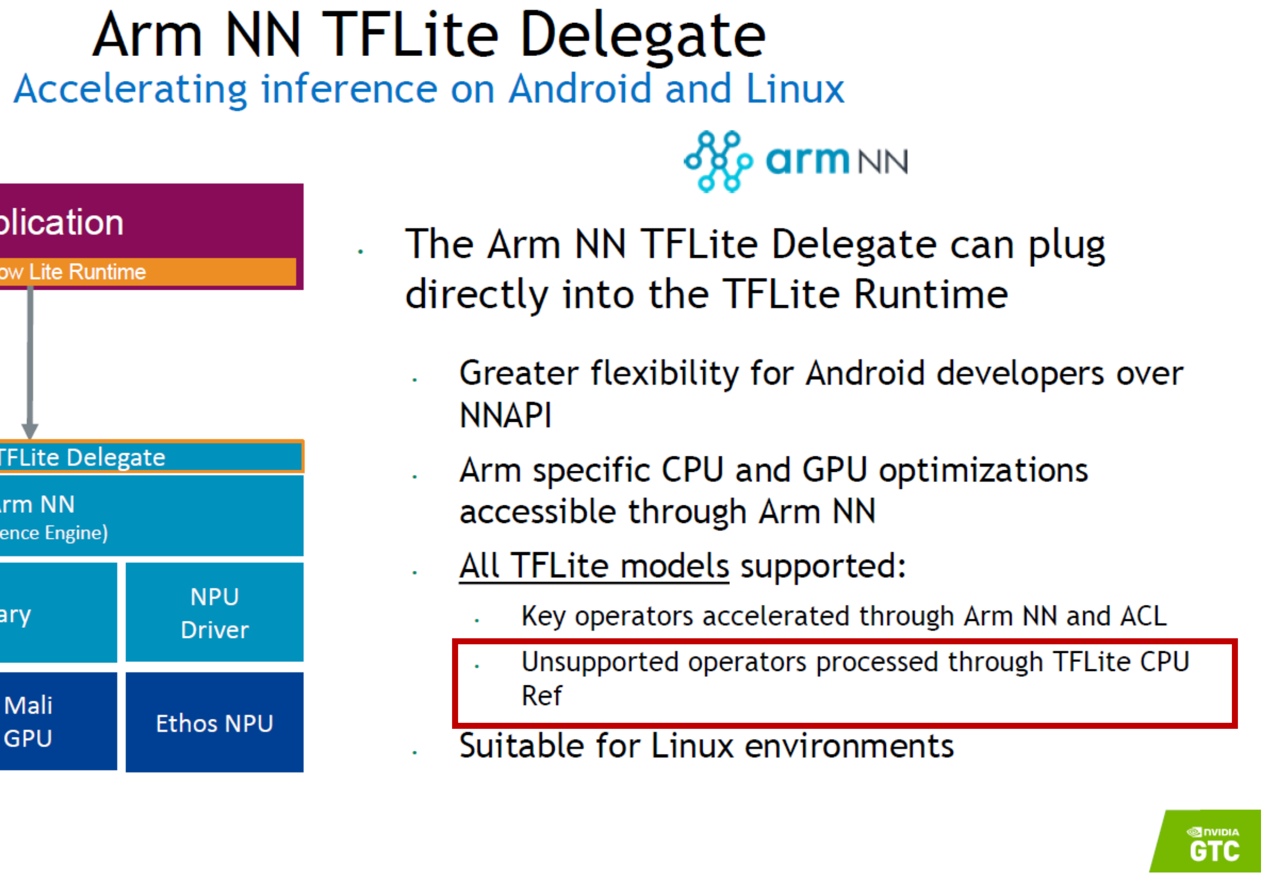
Accelerating inference on Android and Linux



- The Arm NN TFLite Delegate can plug directly into the TFLite Runtime
  - Greater flexibility for Android developers over NNAPI
  - Arm specific CPU and GPU optimizations accessible through Arm NN
  - All TFLite models supported:
    - Key operators accelerated through Arm NN and ACL
    - Unsupported operators processed through TFLite CPU Ref
  - Suitable for Linux environments

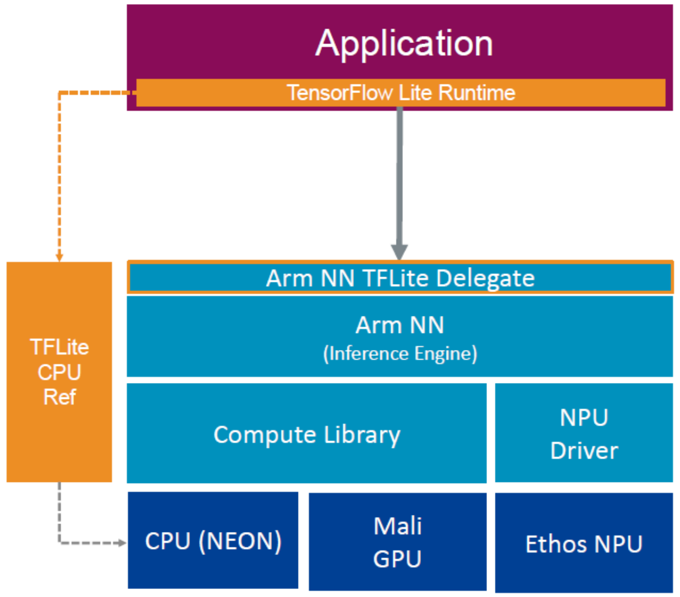
NVIDIA GTC Nov 2021 Conference Presentation
Scalable, Accelerated Hardware-agnostic ML inference with NVIDIA Triton and ARM NN

Case Study: Real-time Anomaly Detection



Arm NN TFLite Delegate

Accelerating inference on Android and Linux



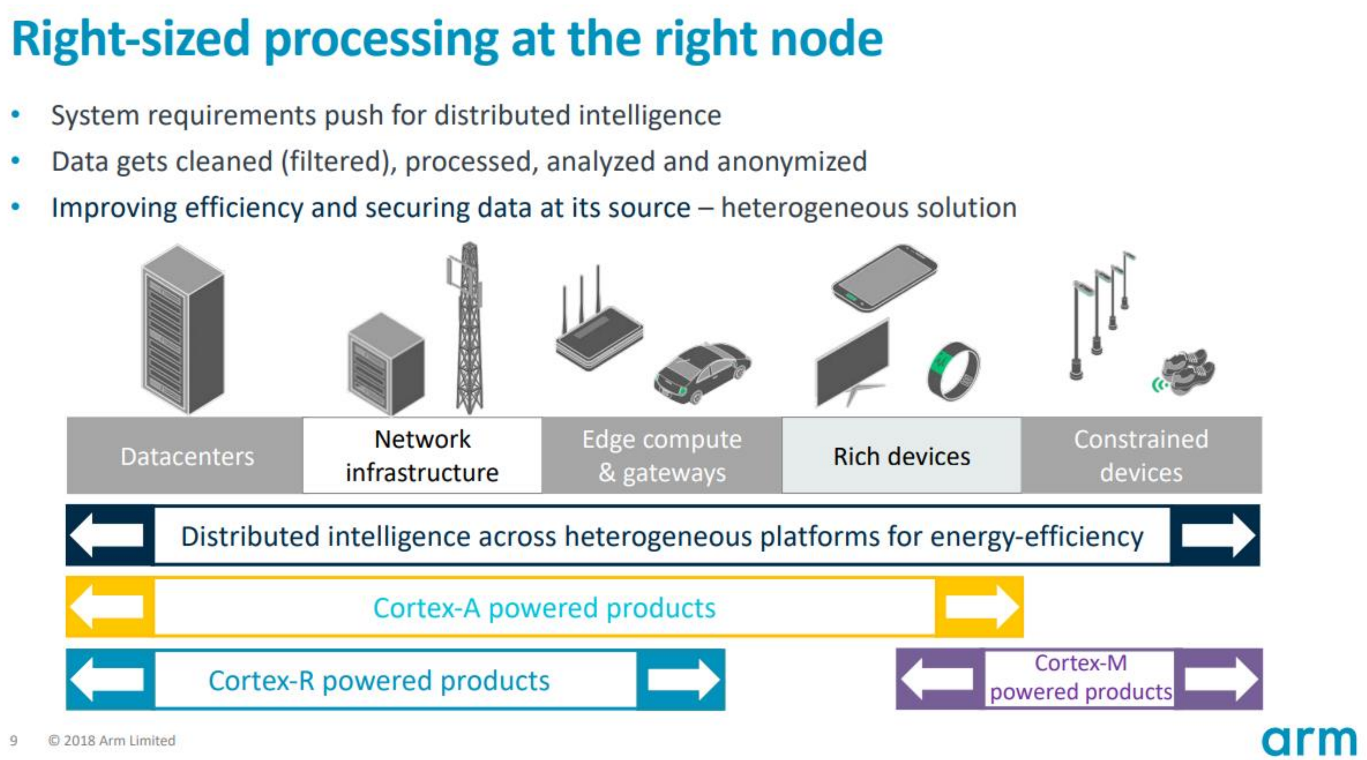
- The Arm NN TFLite Delegate can plug directly into the TFLite Runtime
  - Greater flexibility for Android developers over NNAPI
  - Arm specific CPU and GPU optimizations accessible through Arm NN
  - <u>All TFLite models</u> supported:
    - Key operators accelerated through Arm NN and ACL
    - Unsupported operators processed through TFLite CPU Ref
  - Suitable for Linux environments

NVIDIA GTC Nov 2021 Conference Presentation
Scalable, Accelerated Hardware-agnostic ML inference with NVIDIA Triton and ARM NN

System can only perform as best as vendor provided optimized kernels and implementation of operators!

Latency gated by vendor provided handcrafted kernels

Heterogenous Hardware Targets



Right-sized processing at the right node

- System requirements push for distributed intelligence
- Data gets cleaned (filtered), processed, analyzed and anonymized
- Improving efficiency and securing data at its source – heterogeneous solution

Datacenters
Network infrastructure
Edge compute & gateways
Rich devices
Constrained devices
Distributed intelligence across heterogeneous platforms for energy-efficiency
Cortex-A powered products
Cortex-R powered products
Cortex-M powered products
9
© 2018 Arm Limited
arm

ARM Developer Conference: Nov 2019 Tech Talk

- Varied HW targets available within the ARM ecosystem

- What about Intel CPU, NVIDIA GPUs, Xilinx SoC FPGA + ARM cores, dedicated accelerators?
  - GPU: TensorRT, FPGA: Xilinx Vitis-AI

- How would a ML developer deploy to a different target and benchmark? Tedious to cross-compile, deploy, measure and feedback manually. Can this be automated?

- Would deployment optimization need to be re-done for each target independently on vendor provided library?

Compatibility with Hardware Microarchitecture

- Hardware accelerators primarily differ in:
  - Data layout
  - Memory layout

- Optimizations to be done by compiler stack and not within hardware accelerator (excluding CPU), enabling leaner silicon design with software defined optimization

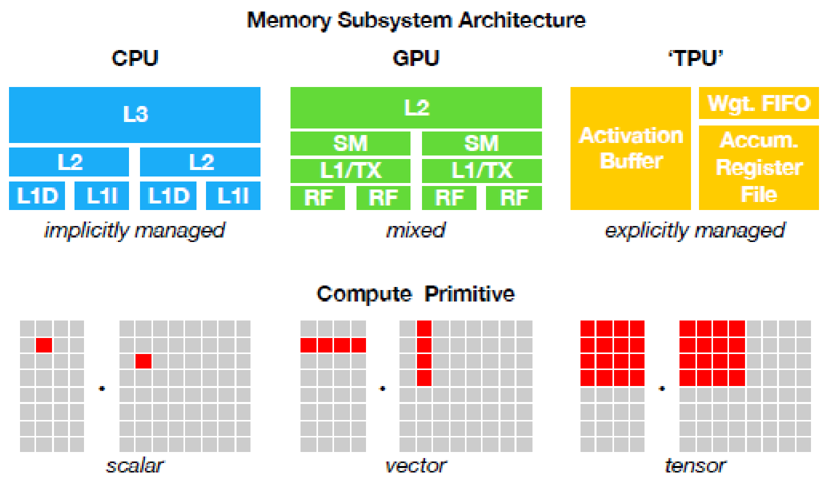


Figure 1: CPU, GPU and TPU-like accelerators require different on-chip memory architectures and compute primitives. This divergence must be addressed when generating optimized code.

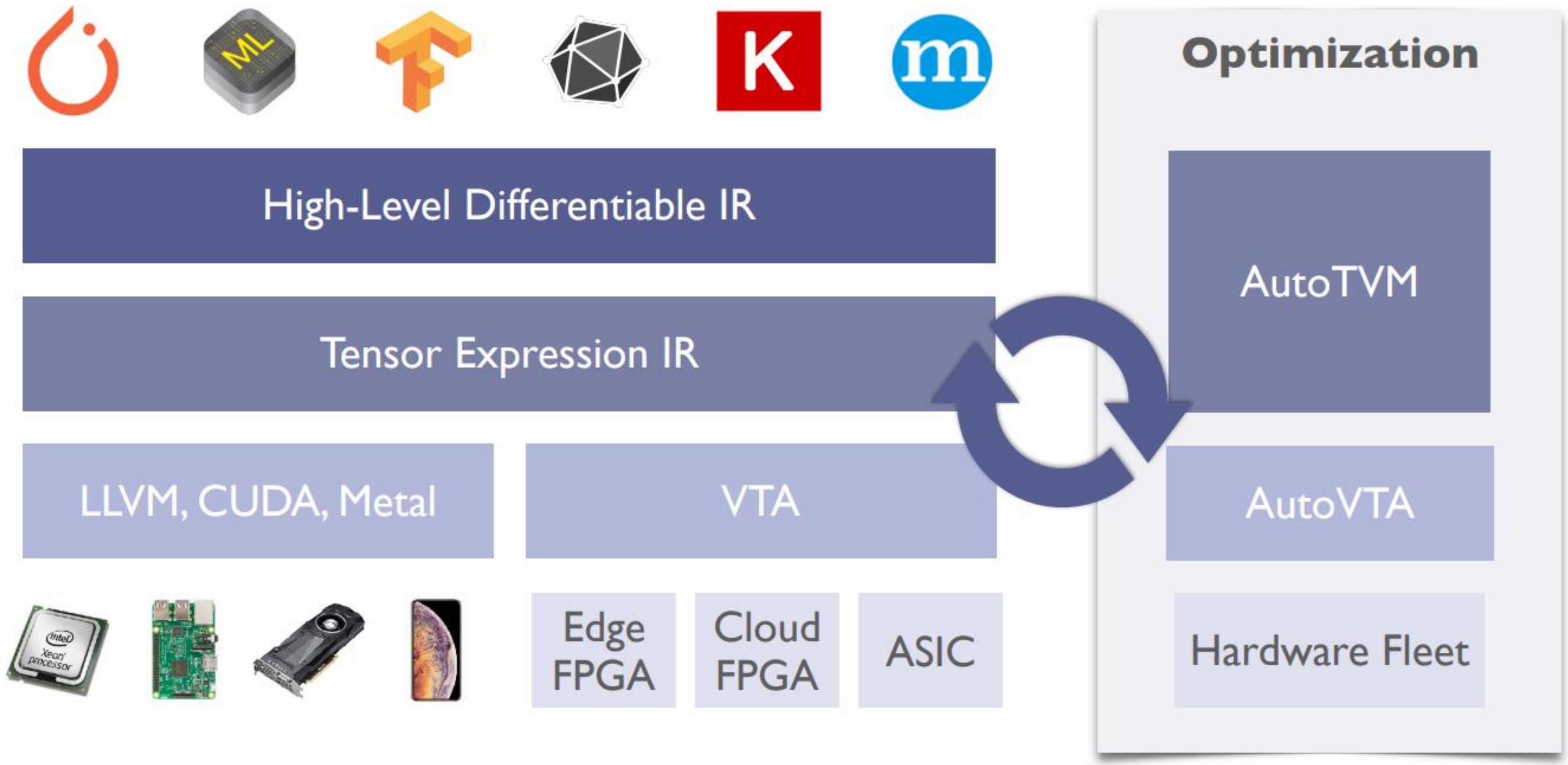
TVM: An Automated End-to-End Optimizing Compiler for Deep Learning

Current State Pain-Points

- ML System Designer’s difficult choices:
  - Avoiding graph optimizations that yield new operations not supported in predefined operator library
  - Using unoptimized implementation of new operators used in models
  - Deploying models to production is manual
  - Requires re-optimization of model operator performance for each desired hardware target using vendor specific runtime libraries
- Solution Motivation:
  - Deep Learning compilers take a high-level IR from existing frameworks and generate low-level optimized code
  - Model architecture agnostic and hardware target agnostic to enable running heterogeneous models on heterogeneous hardware accelerators

DL Compiler System Architecture

TVM: Learning-based Deep Learning Compiler



Tianqi Chen – TVM Conference Overview Presentation
https://sampl.cs.washington.edu/tvmconf/slides/Tianqi-Chen-TVM-Stack-Overview.pdf

DL Compiler System Architecture

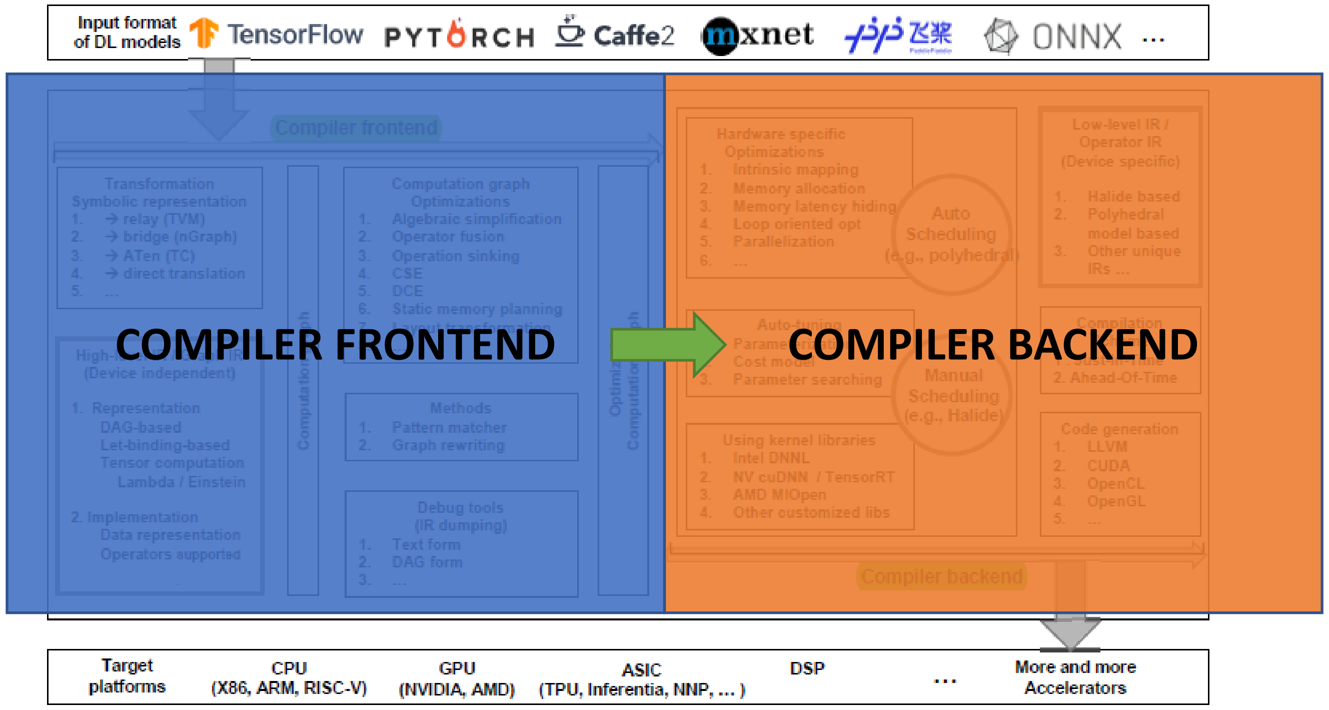


Fig. 2. The overview of commonly adopted design architecture of DL compilers.

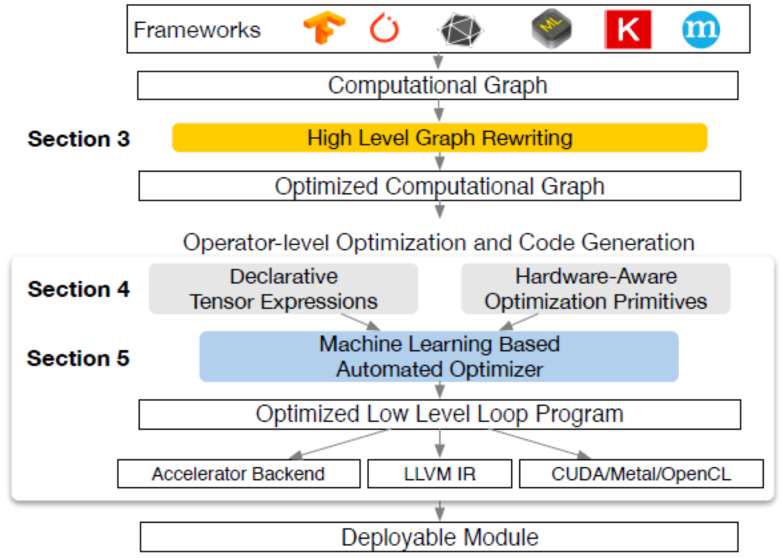

Frameworks
Computational Graph
Section 3
High Level Graph Rewriting
Optimized Computational Graph
Operator-level Optimization and Code Generation
Section 4
Declarative Tensor Expressions
Hardware-Aware Optimization Primitives
Section 5
Machine Learning Based Automated Optimizer
Optimized Low Level Loop Program
Accelerator Backend
LLVM IR
CUDA/Metal/OpenCL
Deployable Module

Figure 2: System overview of TVM. The current stack supports descriptions from many deep learning frameworks and exchange formats, such as CoreML and ONNX, to target major CPU, GPU and specialized accelerators.

The Deep Learning Compiler: A Comprehensive Survey
https://arxiv.org/abs/2002.03794

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

TVM Key Innovations

1. Tensor Expression Language
2. Automated Program Optimization Framework via GBT and a ML based cost model
3. Graph Re-writer + Automatic code generation

Note: This paper discusses #2 – #4

End-User Example

TVM API to get a deployable module

```
import tvm as t
# Use keras framework as example, import model
graph, params = t.frontend.from_keras(keras_model)
target = t.target.cuda()
graph, lib, params = t.compiler.build(graph, target, params)
```

Compiled runtime module contains:

- Graph - the final optimized computational graph
- Lib - generated operators
- Params - module parameters

Deploying model to the target back-end

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

Note: TVM supports multiple deployment back-ends in languages such as C++, Java and Python.

DL Compiler System Architecture

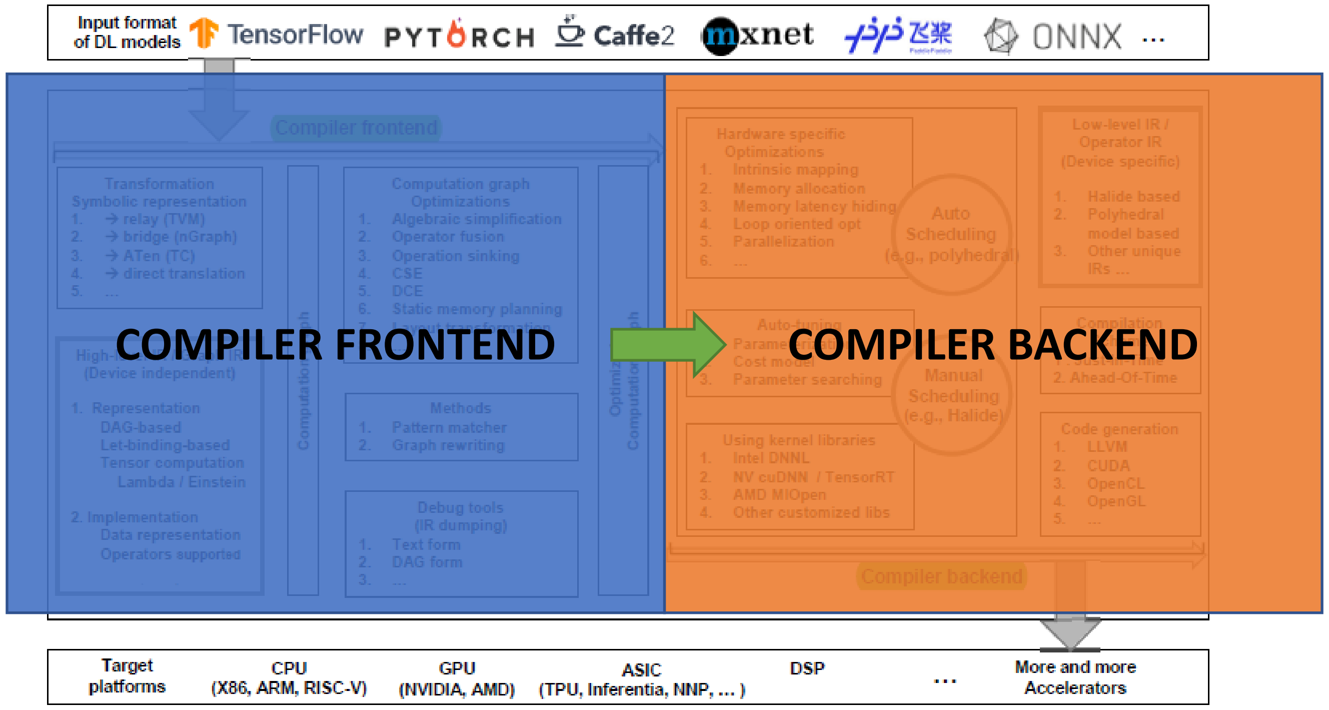
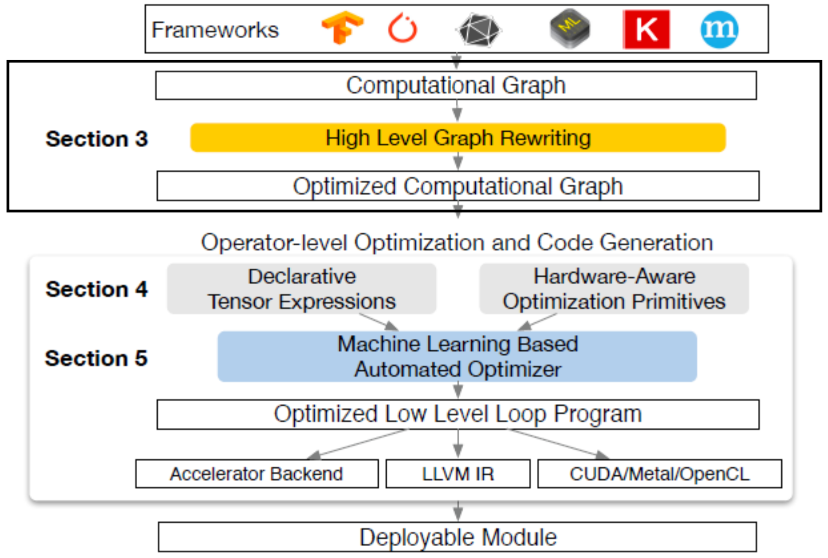


Fig. 2. The overview of commonly adopted design architecture of DL compilers.



Frameworks
Computational Graph
Section 3
High Level Graph Rewriting
Optimized Computational Graph
Operator-level Optimization and Code Generation
Section 4
Declarative Tensor Expressions
Hardware-Aware Optimization Primitives
Section 5
Machine Learning Based Automated Optimizer
Optimized Low Level Loop Program
Accelerator Backend
LLVM IR
CUDA/Metal/OpenCL
Deployable Module

Figure 2: System overview of TVM. The current stack supports descriptions from many deep learning frameworks and exchange formats, such as CoreML and ONNX, to target major CPU, GPU and specialized accelerators.

The Deep Learning Compiler: A Comprehensive Survey
https://arxiv.org/abs/2002.03794

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

DL Compiler System Architecture

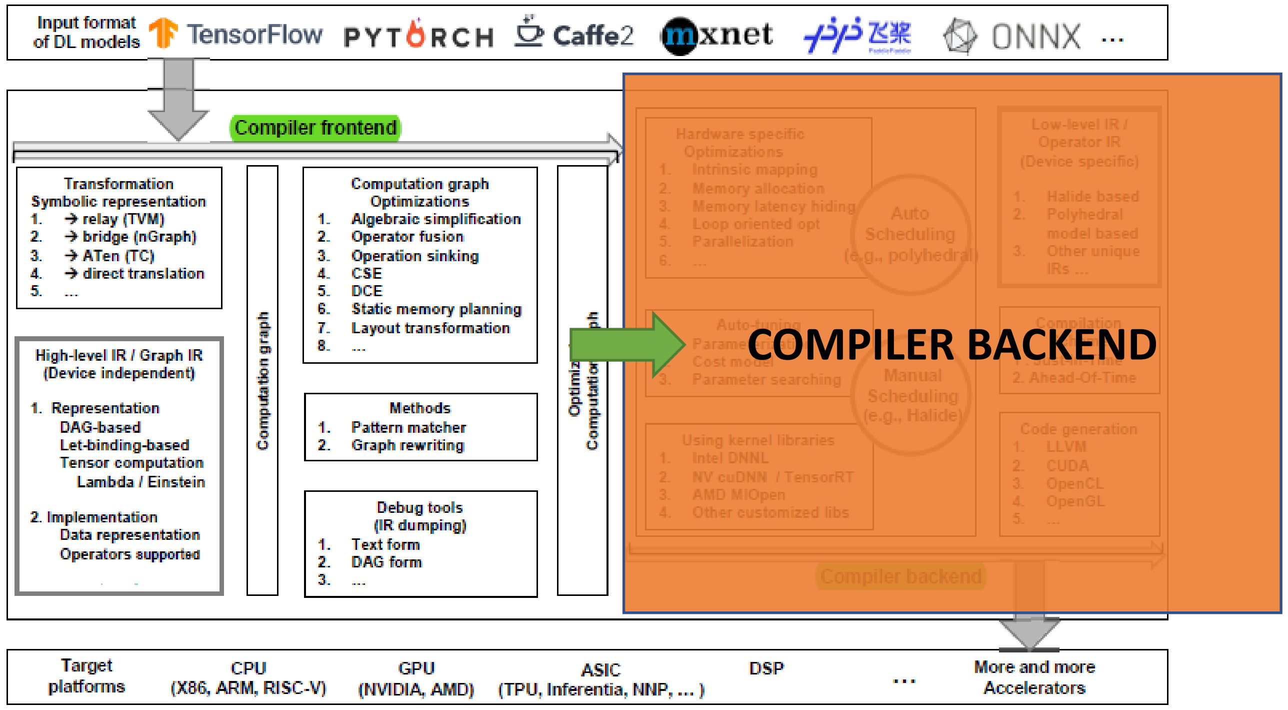
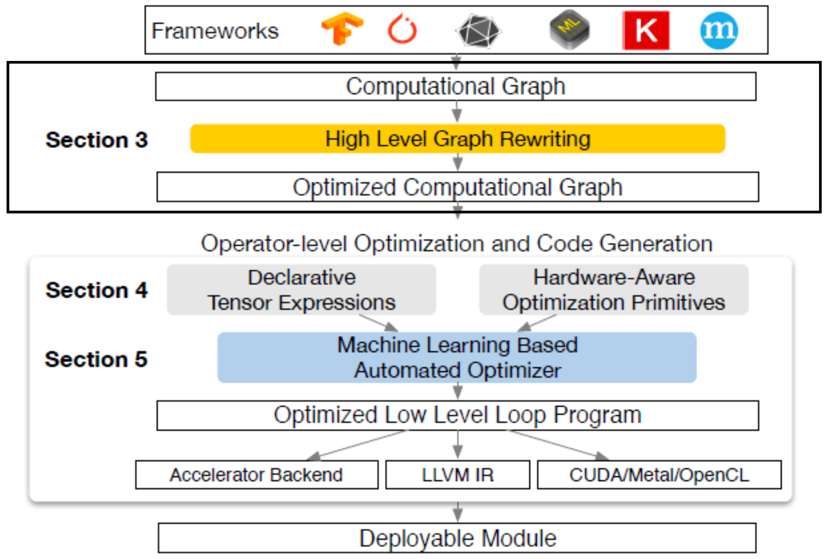


Fig. 2. The overview of commonly adopted design architecture of DL compilers.



Frameworks
Computational Graph
Section 3
High Level Graph Rewriting
Optimized Computational Graph
Operator-level Optimization and Code Generation
Section 4
Declarative Tensor Expressions
Hardware-Aware Optimization Primitives
Section 5
Machine Learning Based Automated Optimizer
Optimized Low Level Loop Program
Accelerator Backend
LLVM IR
CUDA/Metal/OpenCL
Deployable Module

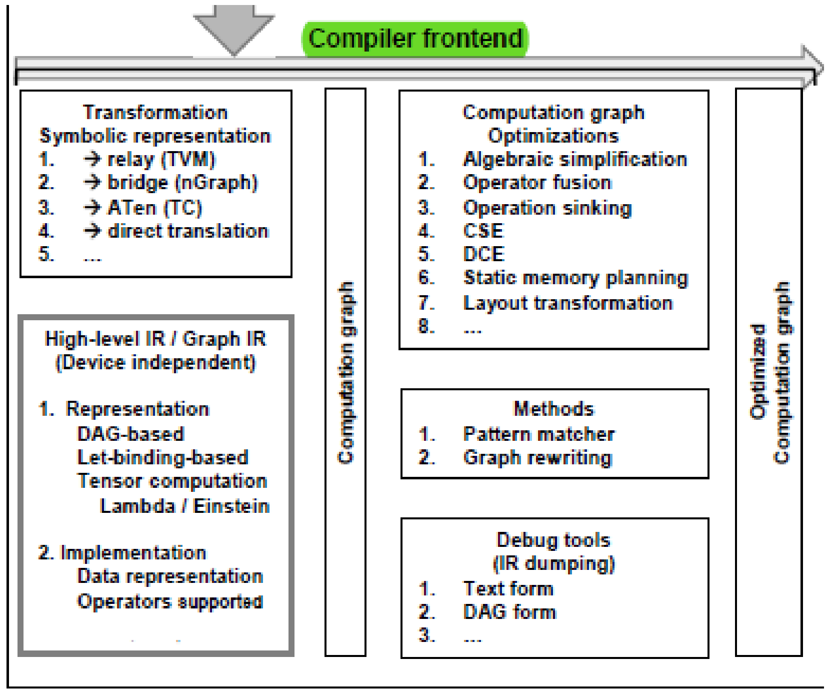
Figure 2: System overview of TVM. The current stack supports descriptions from many deep learning frameworks and exchange formats, such as CoreML and ONNX, to target major CPU, GPU and specialized accelerators.

The Deep Learning Compiler: A Comprehensive Survey
https://arxiv.org/abs/2002.03794

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

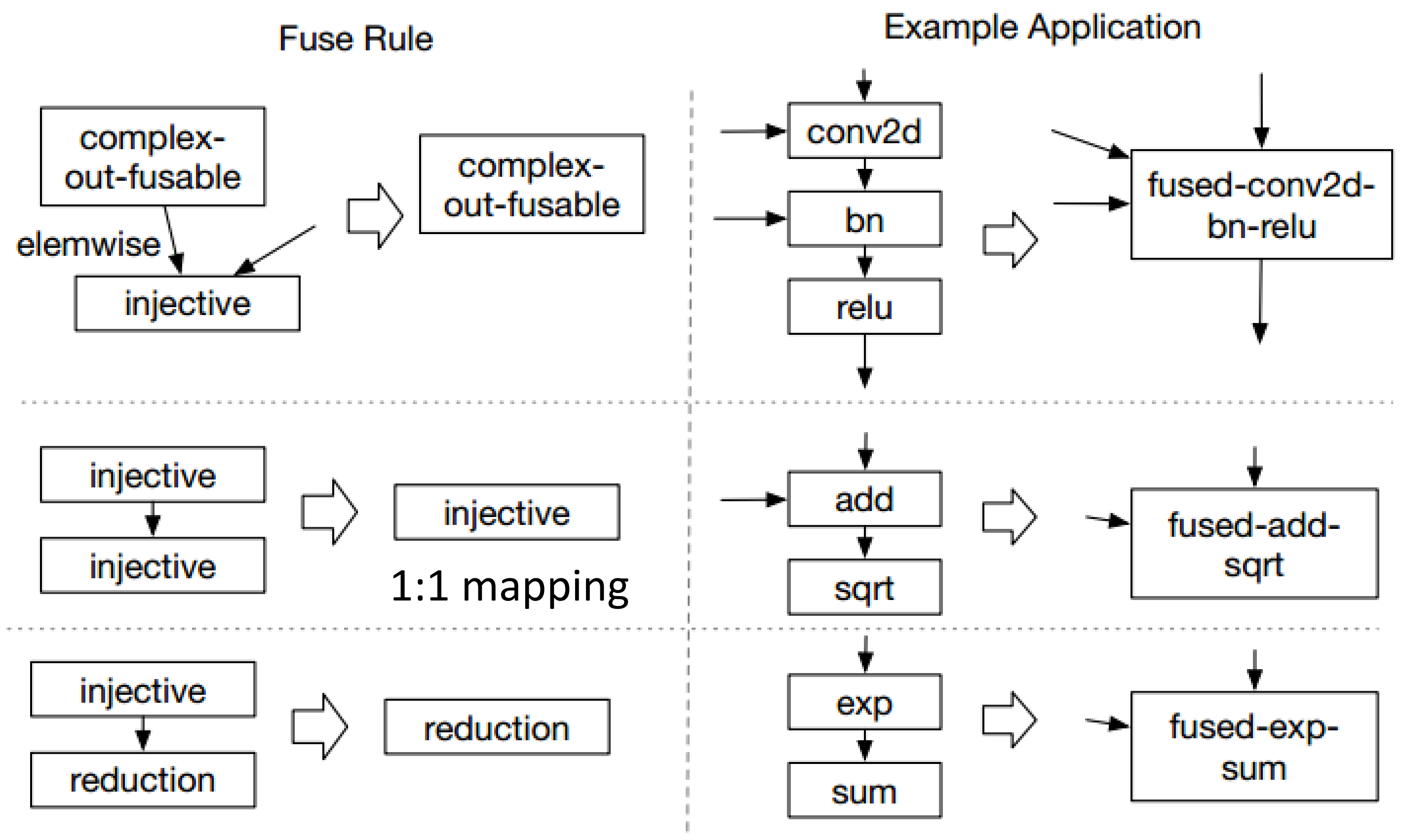
High-Level IR Optimizations

- Computation graph optimizations
  - Operator Fusion (aka kernel fusion)
  - Constant-folding (pre-compute graph sections that are statically determined)
  - Static memory planning (pre-allocation to hold intermediate tensors)
  - Data layout transformations (required to take advantage of hardware accelerator memory layout and re-sizing of operation kernel)



The Deep Learning Compiler: A Comprehensive Survey
https://arxiv.org/abs/2002.03794

3 Rules for Operator Fusion



https://www.programmerall.com/article/68731401786/

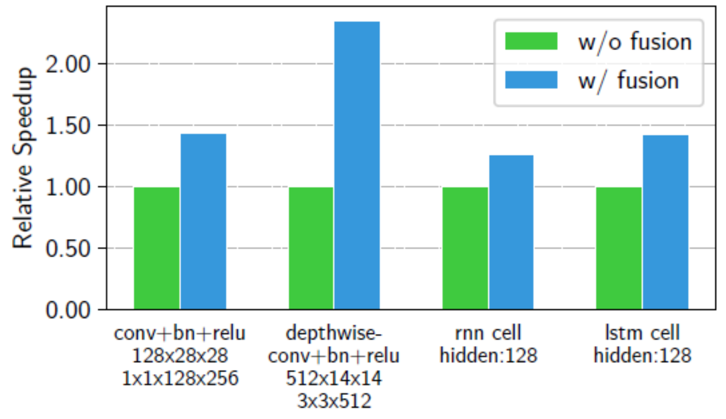
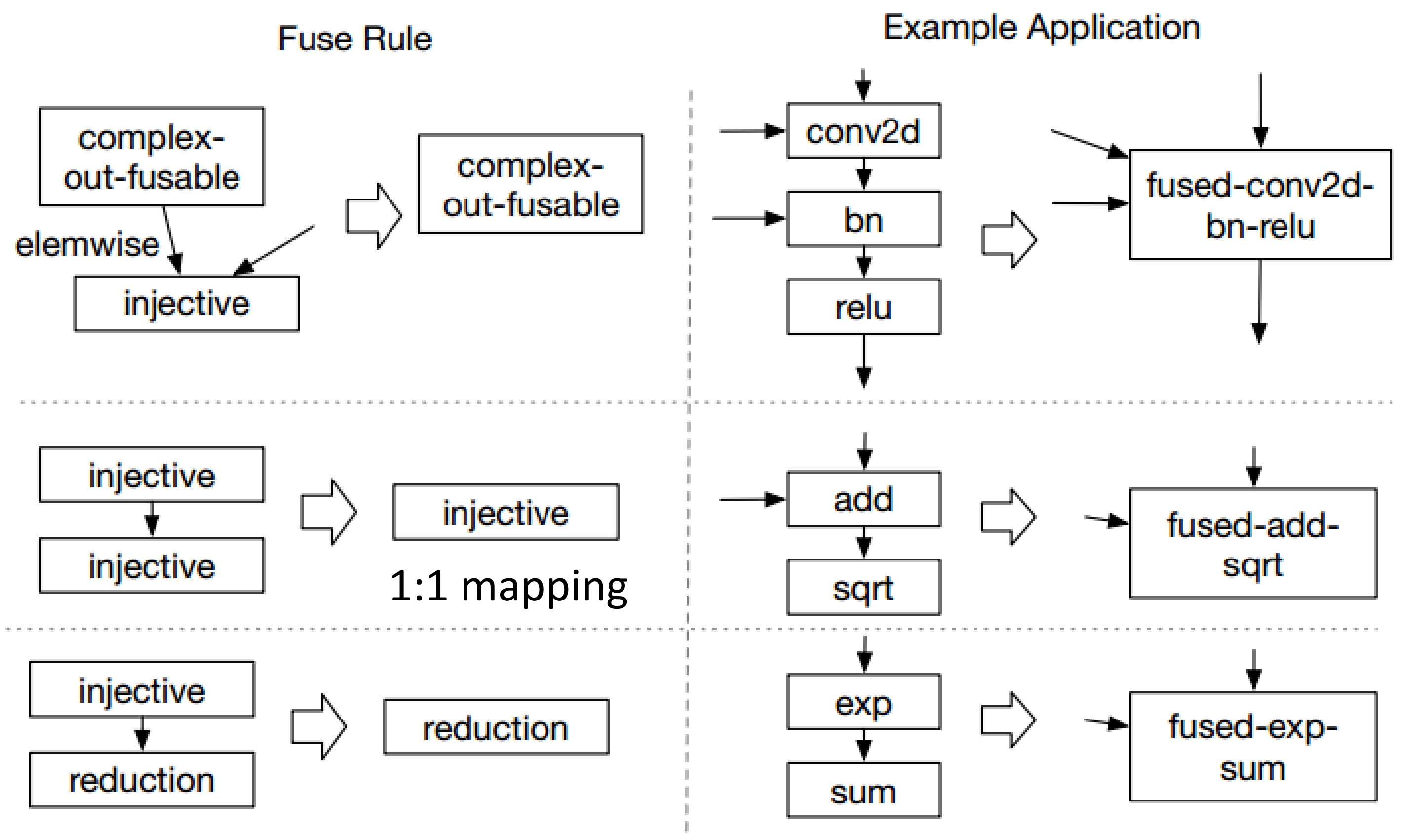


Figure 4: Performance comparison between fused and non-fused operations. TVM generates both operations. Tested on NVIDIA Titan X.

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

3 Rules for Operator Fusion



https://www.programmerall.com/article/68731401786/

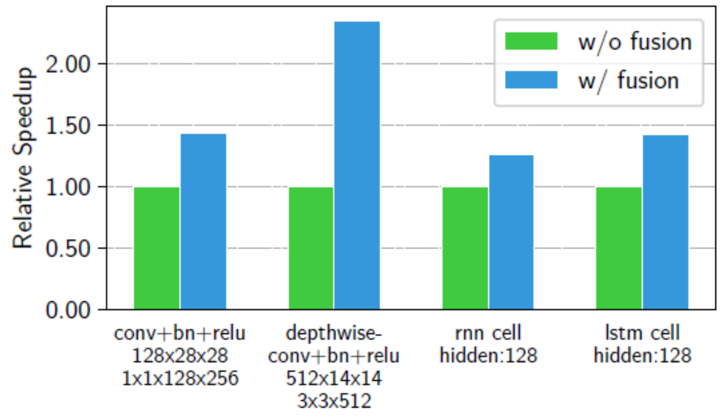


Figure 4: Performance comparison between fused and non-fused operations. TVM generates both operations. Tested on NVIDIA Titan X.

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

Operator fusion results in new operations that may not be supported by handcrafted kernels in vendor provided libraries. TVM proposes a code generation approach and searches the space of suggested operator kernels to pick the best one.

DL Compiler System Architecture

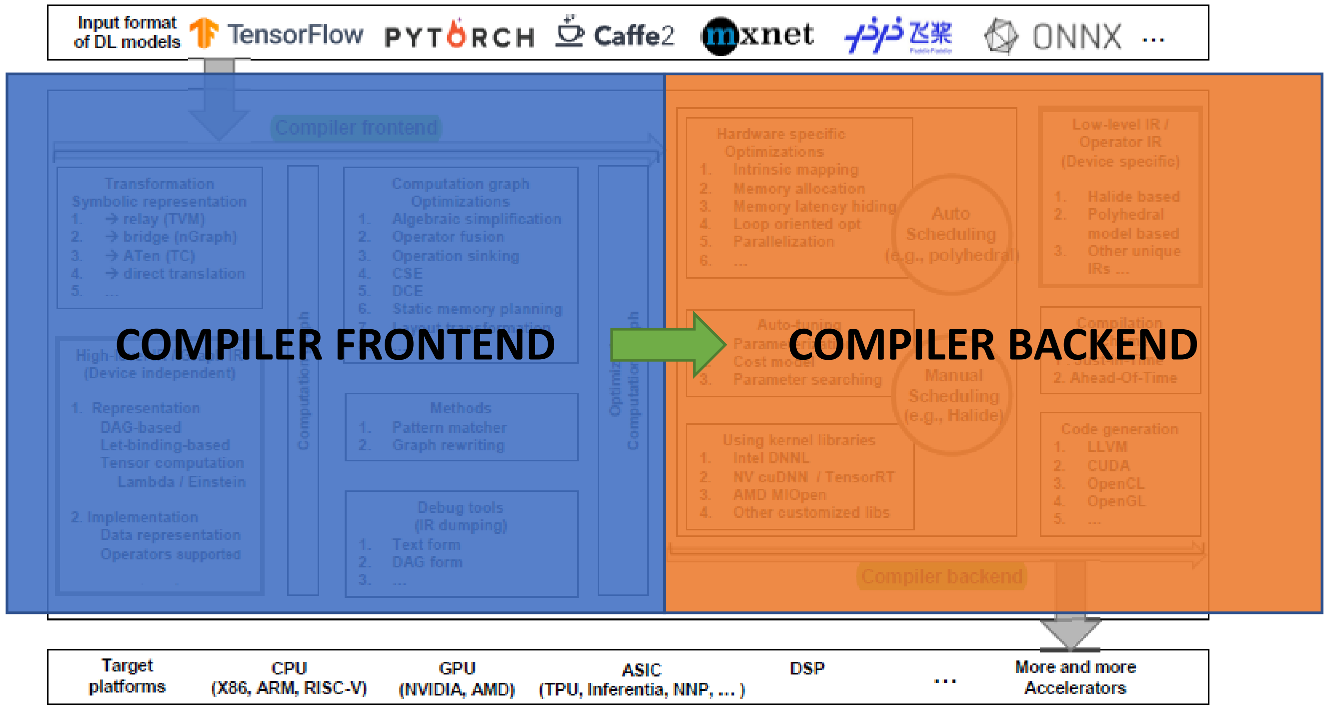


Fig. 2. The overview of commonly adopted design architecture of DL compilers.

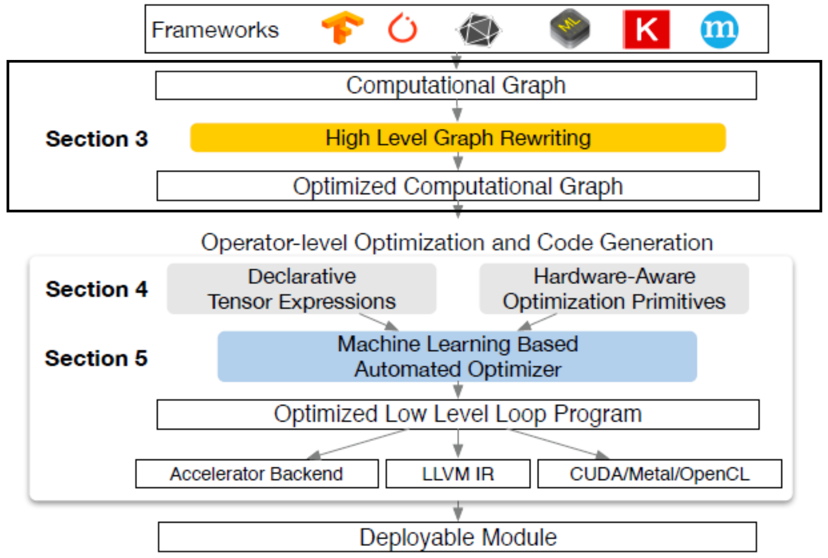

Frameworks
Section 3
Computational Graph
High Level Graph Rewriting
Optimized Computational Graph
Operator-level Optimization and Code Generation
Section 4
Declarative Tensor Expressions
Hardware-Aware Optimization Primitives
Section 5
Machine Learning Based Automated Optimizer
Optimized Low Level Loop Program
Accelerator Backend
LLVM IR
CUDA/Metal/OpenCL
Deployable Module

Figure 2: System overview of TVM. The current stack supports descriptions from many deep learning frameworks and exchange formats, such as CoreML and ONNX, to target major CPU, GPU and specialized accelerators.

The Deep Learning Compiler: A Comprehensive Survey
https://arxiv.org/abs/2002.03794

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

DL Compiler System Architecture

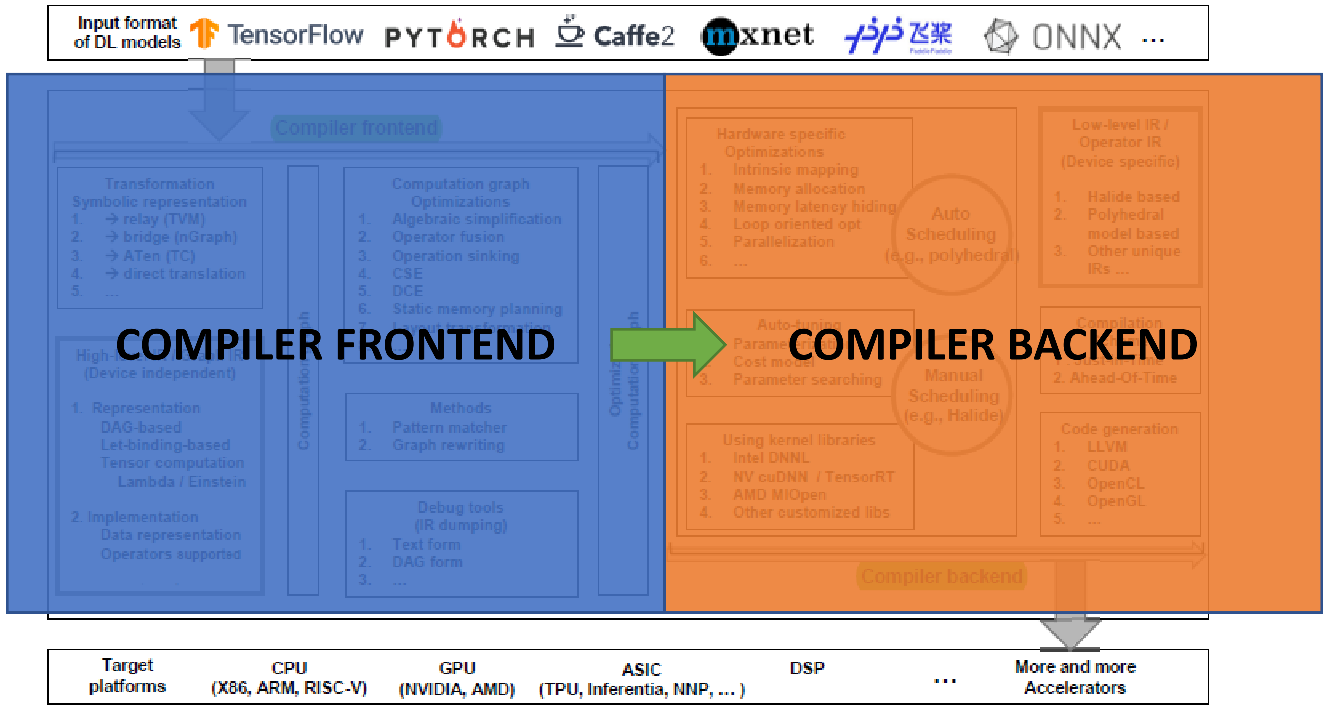


Fig. 2. The overview of commonly adopted design architecture of DL compilers.

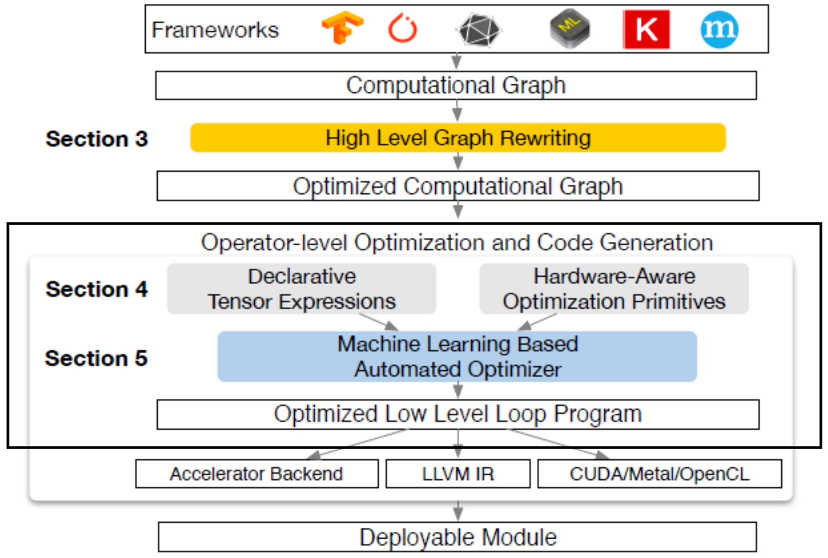

Frameworks
Computational Graph
Section 3
High Level Graph Rewriting
Optimized Computational Graph
Operator-level Optimization and Code Generation
Section 4
Declarative Tensor Expressions
Hardware-Aware Optimization Primitives
Section 5
Machine Learning Based Automated Optimizer
Optimized Low Level Loop Program
Accelerator Backend
LLVM IR
CUDA/Metal/OpenCL
Deployable Module

Figure 2: System overview of TVM. The current stack supports descriptions from many deep learning frameworks and exchange formats, such as CoreML and ONNX, to target major CPU, GPU and specialized accelerators.

The Deep Learning Compiler: A Comprehensive Survey
https://arxiv.org/abs/2002.03794

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

DL Compiler System Architecture

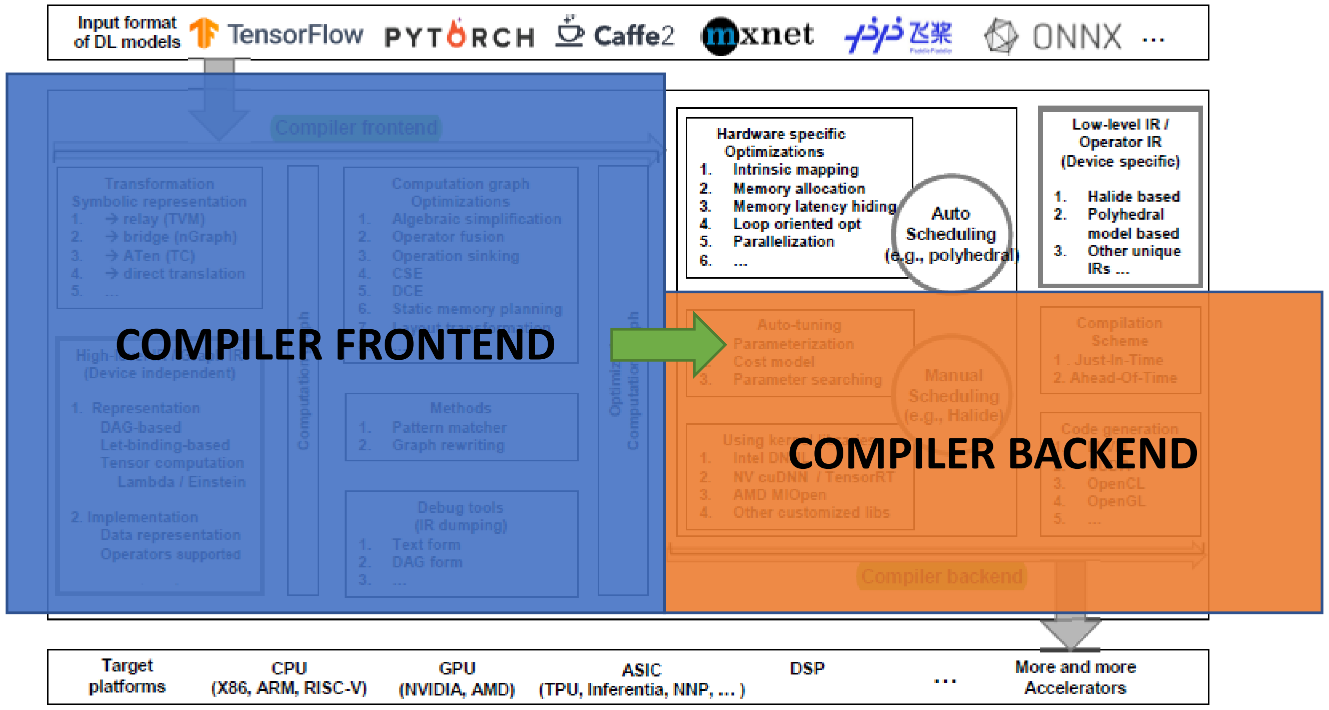
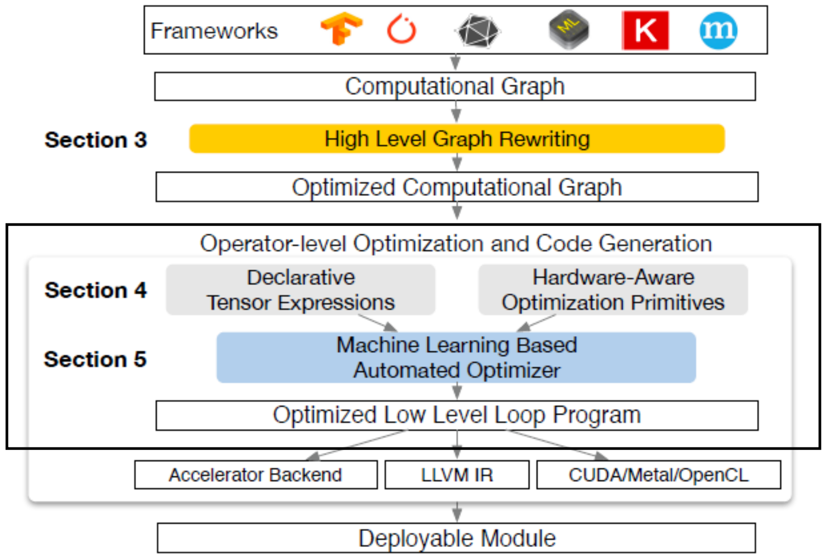


Fig. 2. The overview of commonly adopted design architecture of DL compilers.



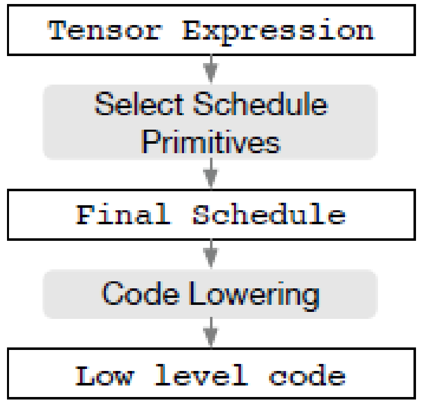
Frameworks
Computational Graph
Section 3
High Level Graph Rewriting
Optimized Computational Graph
Operator-level Optimization and Code Generation
Section 4
Declarative Tensor Expressions
Hardware-Aware Optimization Primitives
Section 5
Machine Learning Based Automated Optimizer
Optimized Low Level Loop Program
Accelerator Backend
LLVM IR
CUDA/Metal/OpenCL
Deployable Module

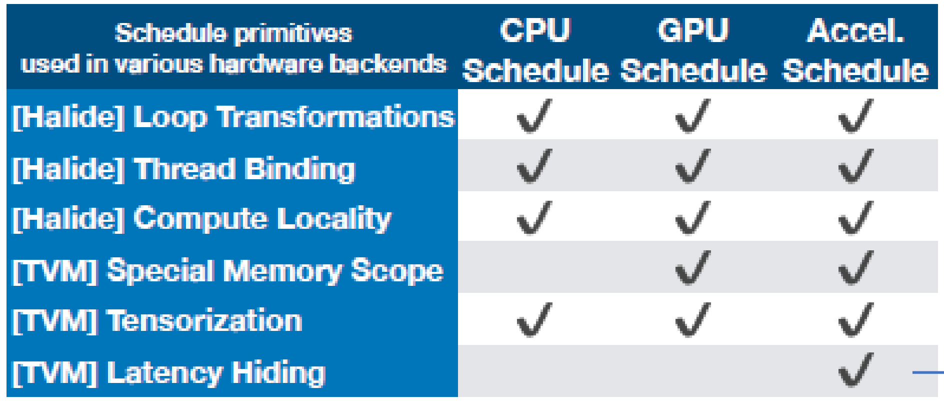
Figure 2: System overview of TVM. The current stack supports descriptions from many deep learning frameworks and exchange formats, such as CoreML and ONNX, to target major CPU, GPU and specialized accelerators.

The Deep Learning Compiler: A Comprehensive Survey
https://arxiv.org/abs/2002.03794

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

Generating Tensor Operations





Schedule primitives used in various hardware backends	CPU Schedule	GPU Schedule	Accel. Schedule
[Halide] Loop Transformations	✓	✓	✓
[Halide] Thread Binding	✓	✓	✓
[Halide] Compute Locality	✓	✓	✓
[TVM] Special Memory Scope		✓	✓
[TVM] Tensorization	✓	✓	✓
[TVM] Latency Hiding			✓

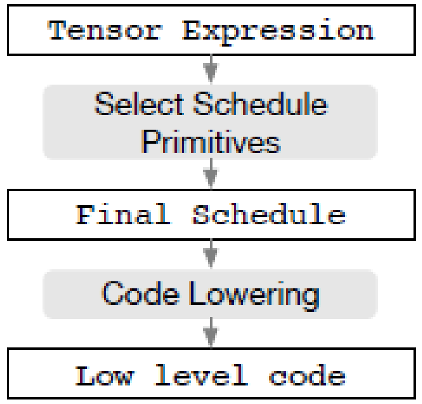
Only for TPU like accelerators

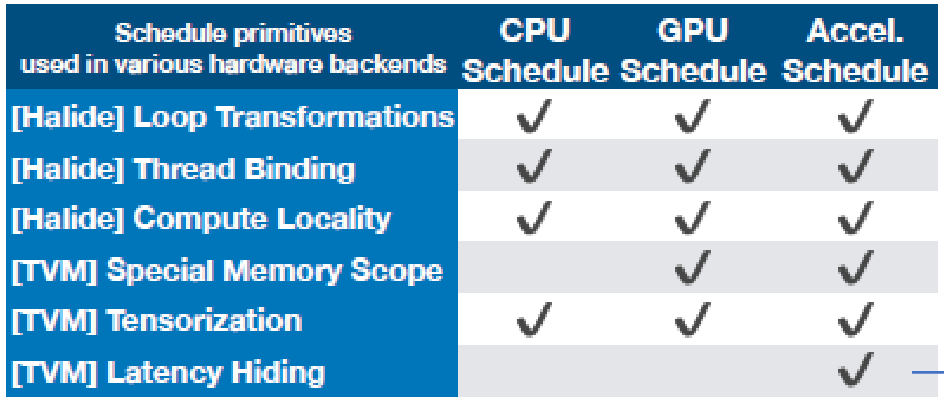
Builds on Halide's design style of decoupling scheduling from computation rules

Generating Tensor Operations

User-defined expression to schedule each operation

Requires knowledge of target hardware and operator semantics





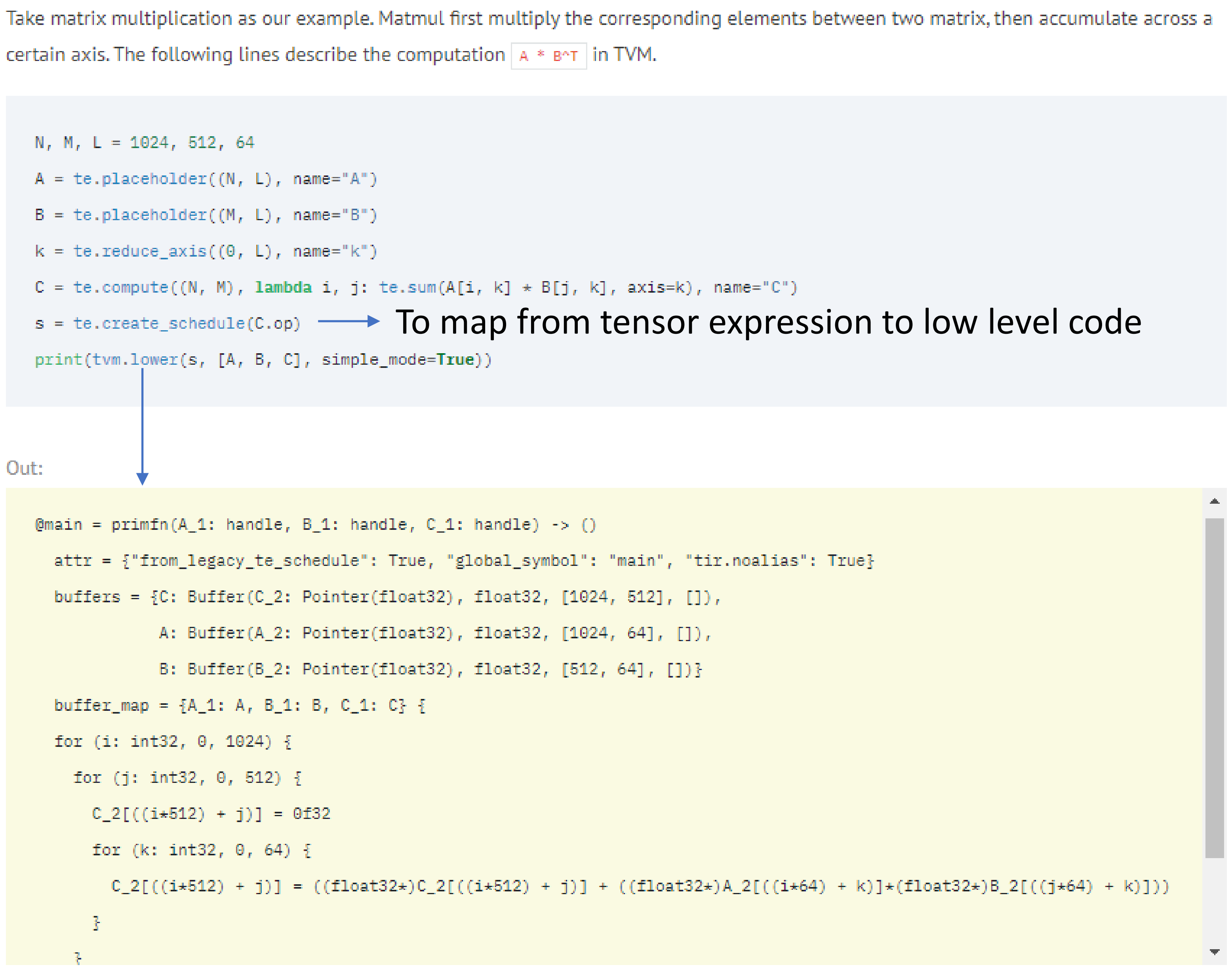
Schedule primitives used in various hardware backends	CPU Schedule	GPU Schedule	Accel. Schedule
[Halide] Loop Transformations	✓	✓	✓
[Halide] Thread Binding	✓	✓	✓
[Halide] Compute Locality	✓	✓	✓
[TVM] Special Memory Scope		✓	✓
[TVM] Tensorization	✓	✓	✓
[TVM] Latency Hiding			✓

Only for TPU like accelerators

Builds on Halide’s design style of decoupling scheduling from computation rules

Tensorize Example

Matrix Mult



Take matrix multiplication as our example. Matmul first multiply the corresponding elements between two matrix, then accumulate across a certain axis. The following lines describe the computation `A * B^T` in TVM.

```python
N, M, L = 1024, 512, 64
A = te.placeholder((N, L), name="A")
B = te.placeholder((M, L), name="B")
k = te.reduce_axis((0, L), name="k")
C = te.compute((N, M), lambda i, j: te.sum(A[i, k] * B[j, k], axis=k), name="C")
s = te.create_schedule(C.op)
print(tvm.lower(s, [A, B, C], simple_mode=True))
```

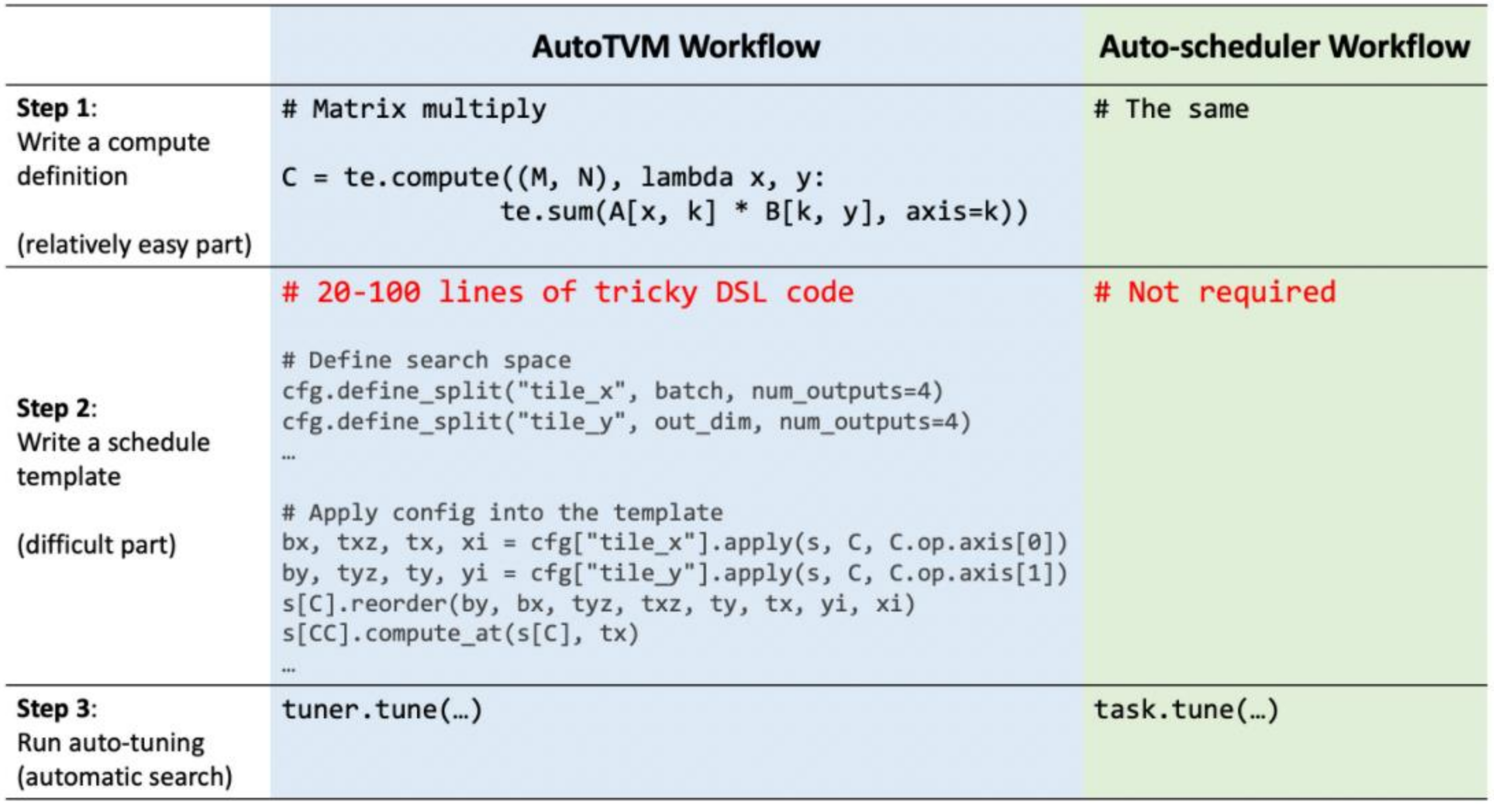
To map from tensor expression to low level code

Out:

```
@main = primfn(A_1: handle, B_1: handle, C_1: handle) -> ()
  attr = {"from_legacy_te_schedule": True, "global_symbol": "main", "tir.noalias": True}
  buffers = {C: Buffer(C_2: Pointer(float32), float32, [1024, 512], []),
             A: Buffer(A_2: Pointer(float32), float32, [1024, 64], []),
             B: Buffer(B_2: Pointer(float32), float32, [512, 64], [])}
  buffer_map = {A_1: A, B_1: B, C_1: C} {
  for (i: int32, 0, 1024) {
    for (j: int32, 0, 512) {
      C_2[((i*512) + j)] = 0f32
      for (k: int32, 0, 64) {
        C_2[((i*512) + j)] = ((float32*)C_2[((i*512) + j)] + ((float32*)A_2[((i*64) + k)]*(float32*)B_2[((j*64) + k)]))
      }
    }
```

https://tvm.apache.org/docs//how_to/work_with_schedules/tensorize.html

Solution to AutoTVM schedule template



	AutoTVM Workflow	Auto-scheduler Workflow

Step 1:
Write a compute definition
 	
(relatively easy part)
	`# Matrix multiply`
	
	`C = te.compute((M, N), lambda x, y:`
	`            te.sum(A[x, k] * B[k, y], axis=k))`
		`# The same`

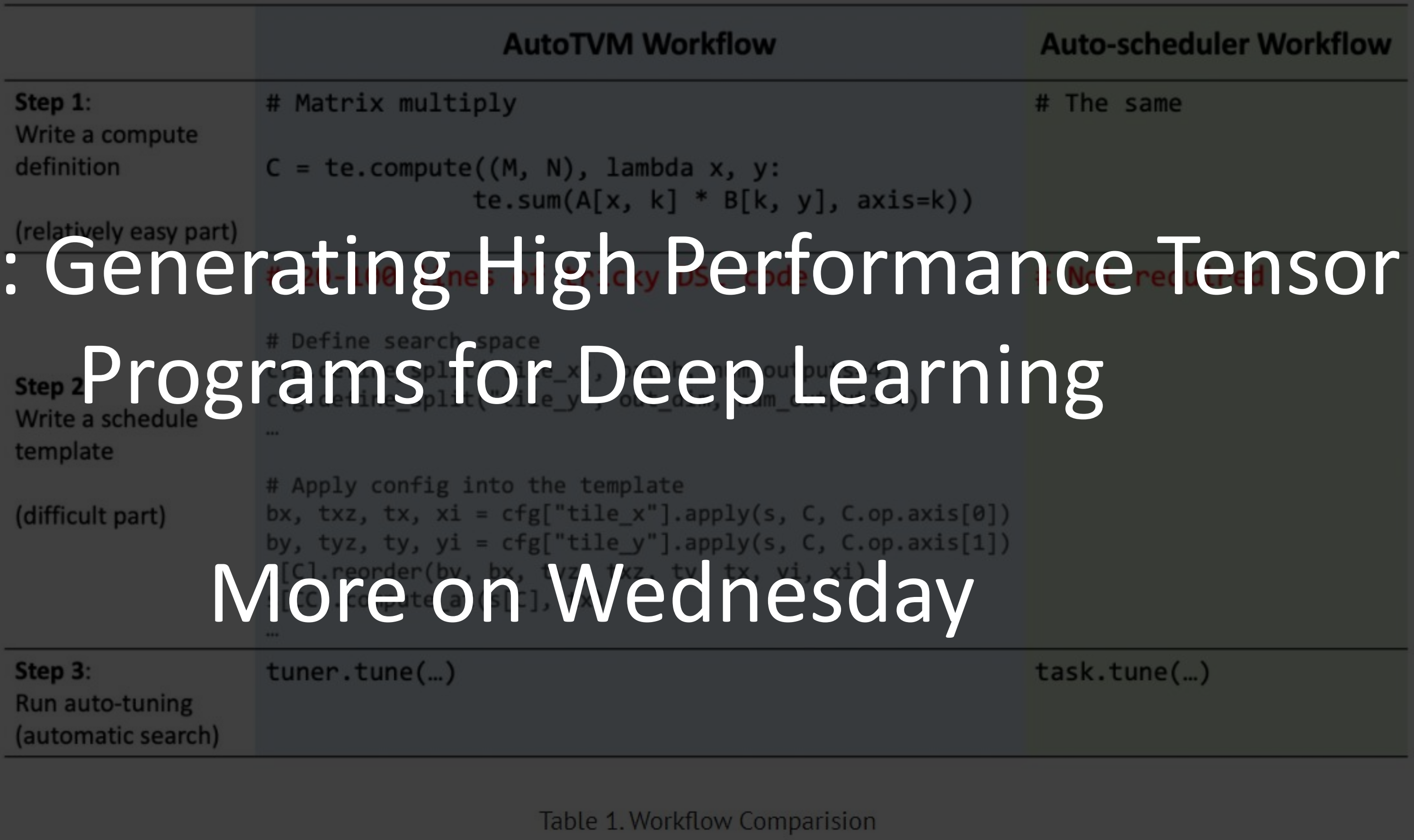
Step 2:
Write a schedule template
 	
(difficult part)
	`# 20-100 lines of tricky DSL code`
	
	`# Define search space`
	`cfg.define_split("tile_x", batch, num_outputs=4)`
	`cfg.define_split("tile_y", out_dim, num_outputs=4)`
	`…`
	
	`# Apply config into the template`
	`bx, txz, tx, xi = cfg["tile_x"].apply(s, C, C.op.axis[0])`
	`by, tyz, ty, yi = cfg["tile_y"].apply(s, C, C.op.axis[1])`
	`s[C].reorder(by, bx, tyz, txz, ty, tx, yi, xi)`
	`s[CC].compute_at(s[C], tx)`
	`…`
		`# Not required`

Step 3: Run auto-tuning (automatic search)	`tuner.tune(…)`	`task.tune(…)`

Table 1. Workflow Comparision

https://tvm.apache.org/2021/03/03/intro-auto-scheduler

Solution to AutoTVM schedule template



Ansor: Generating High Performance Tensor Programs for Deep Learning

More on Wednesday

	AutoTVM Workflow	Auto-scheduler Workflow

Step 1:
Write a compute definition
 	
(relatively easy part)
	`# Matrix multiply`
	
	`C = te.compute((M, N), lambda x, y:`
	`        te.sum(A[x, k] * B[k, y], axis=k))`
		`# The same`

Step 2:
Write a schedule template
 	
(difficult part)
	`# 20-100 lines of tricky DSL code`
	
	`# Define search space`
	`cfg.define_split("tile_x", [illegible], num_outputs=4)`
	`cfg.define_split("tile_y", out_dim, num_outputs=4)`
	`…`
	
	`# Apply config into the template`
	`bx, txz, tx, xi = cfg["tile_x"].apply(s, C, C.op.axis[0])`
	`by, tyz, ty, yi = cfg["tile_y"].apply(s, C, C.op.axis[1])`
	`s[C].reorder(by, bx, tyz, txz, ty, tx, yi, xi)`
	`s[C].compute_at(s[C], [illegible])`
	`…`
		`# Not required`

Step 3: Run auto-tuning (automatic search)	`tuner.tune(…)`	`task.tune(…)`

Table 1. Workflow Comparision

https://tvm.apache.org/2021/03/03/intro-auto-scheduler

Low Level IR & Hardware Specific **Tensorize** Optimizations

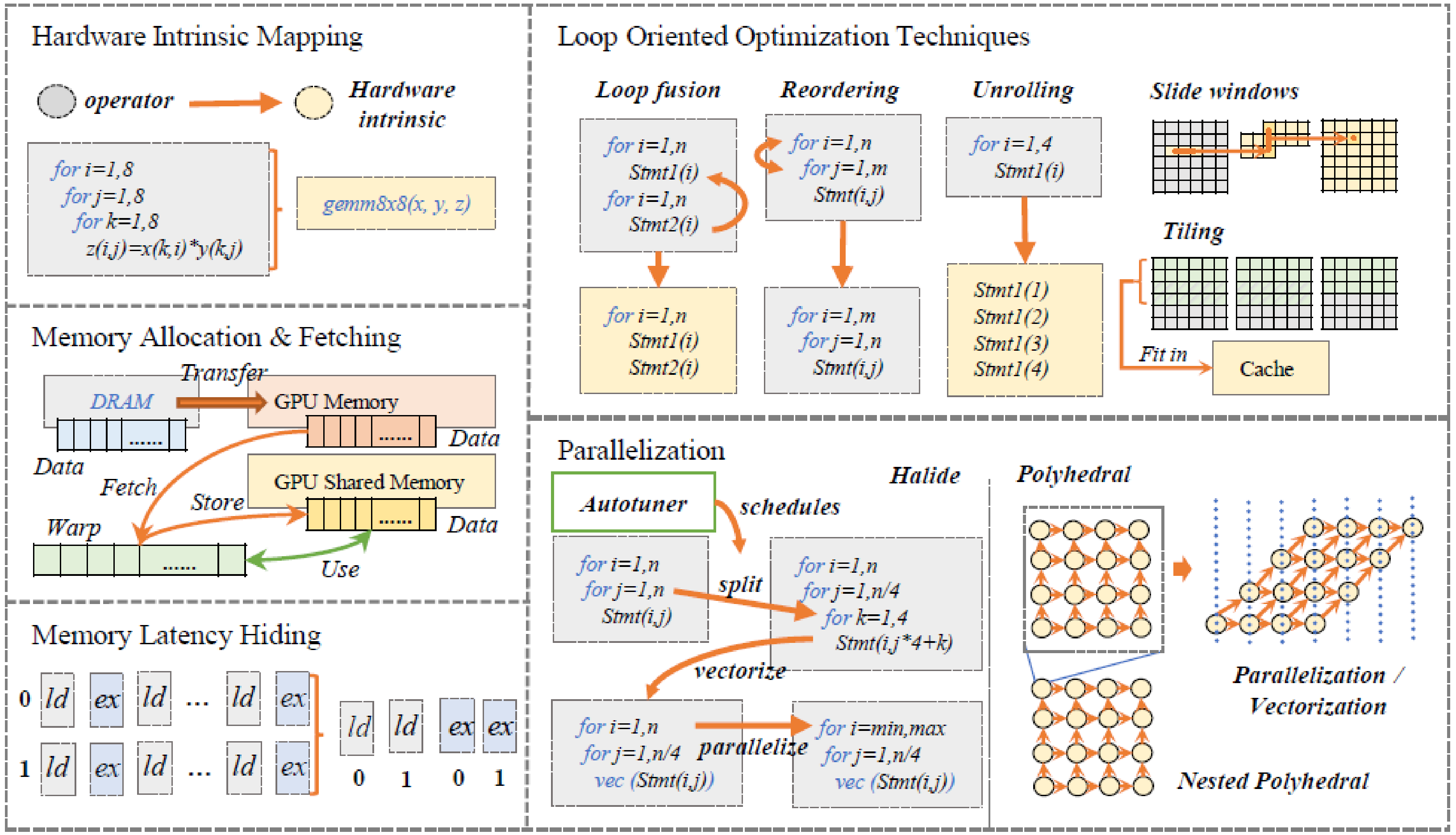
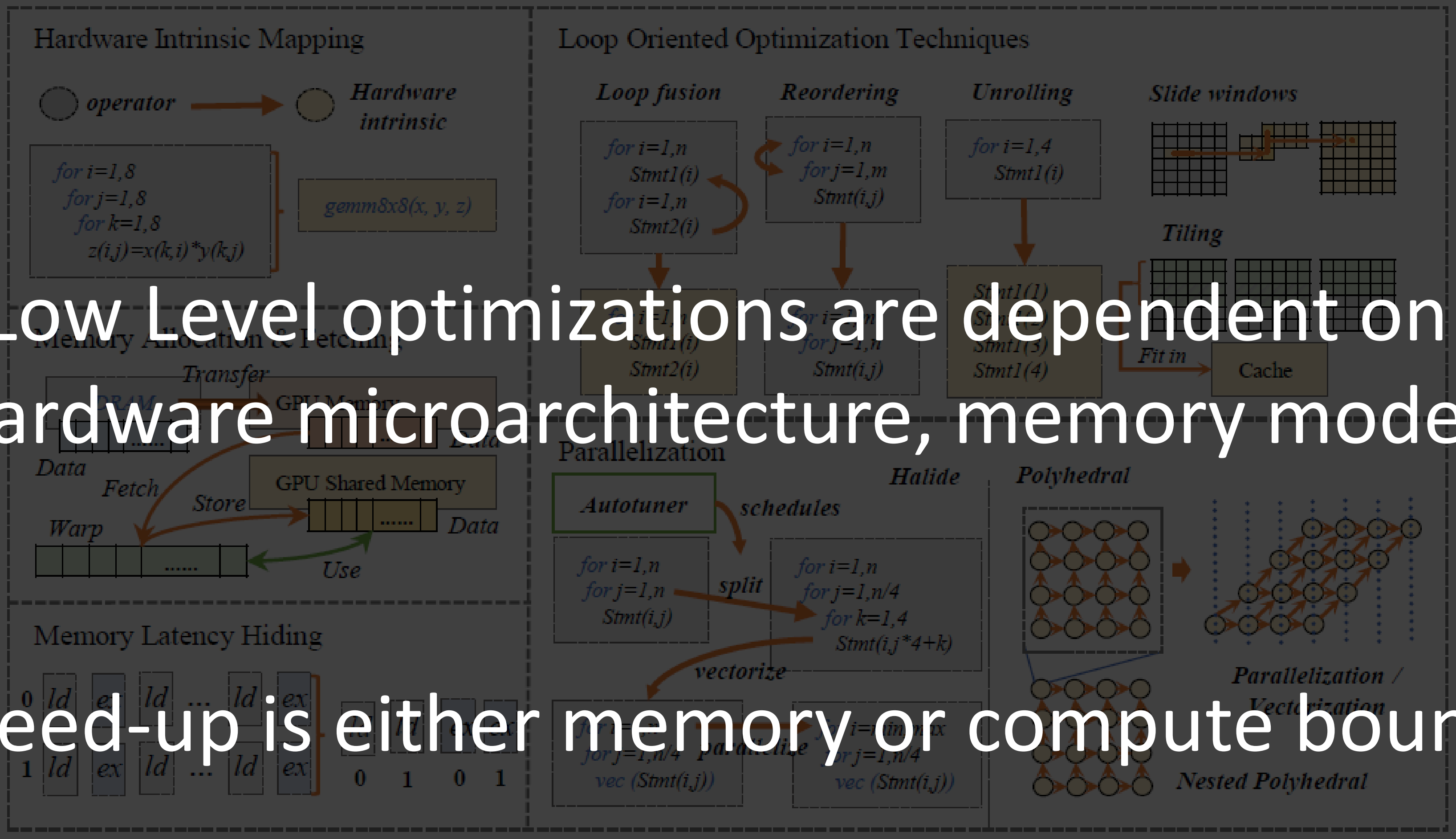


Fig. 4. Overview of hardware-specific optimizations applied in DL compilers.

The Deep Learning Compiler: A Comprehensive Survey
https://arxiv.org/abs/2002.03794

Low Level IR & Hardware Specific **Tensorize** Optimizations



```
for i=1,8
  for j=1,8
    for k=1,8
      z(i,j)=x(k,i)*y(k,j)
```

```
for i=1,n
  Stmt1(i)
for i=1,n
  Stmt2(i)
```

```
for i=1,n
  for j=1,m
    Stmt(i,j)
```

```
for i=1,4
  Stmt1(i)
```

```
for i=1,n
  for j=1,n
    Stmt(i,j)
```

```
for i=1,n
  for j=1,n/4
    for k=1,4
      Stmt(i,j*4+k)
```

Low Level optimizations are dependent on hardware microarchitecture, memory model

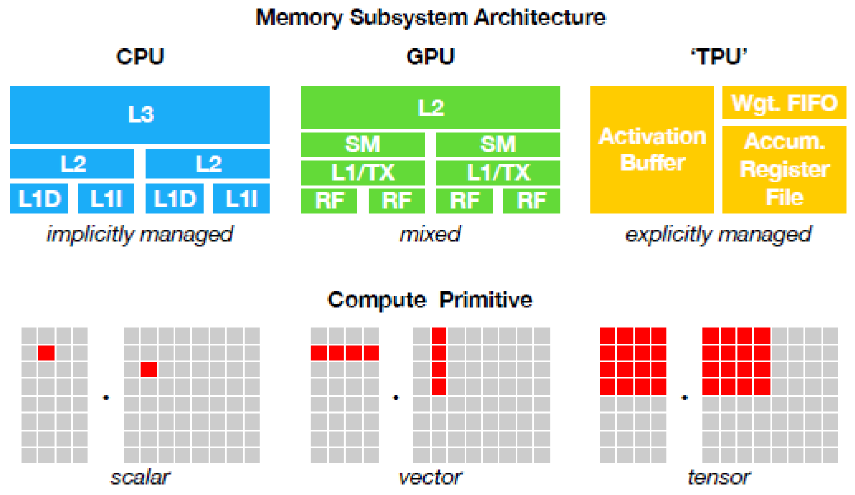
Speed-up is either memory or compute bound

Fig. 4. Overview of hardware-specific optimizations applied in DL compilers.

The Deep Learning Compiler: A Comprehensive Survey
https://arxiv.org/abs/2002.03794

Tensorize Example: Matrix Mult

- Nested loop parallelism (exploiting HW multi-thread hierarchy)
- Tiling & caching of data in (shared) memory while ensuring memory scope definition + atomic synchronization (if necessary)
- Lowering to hardware via ISA specification



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

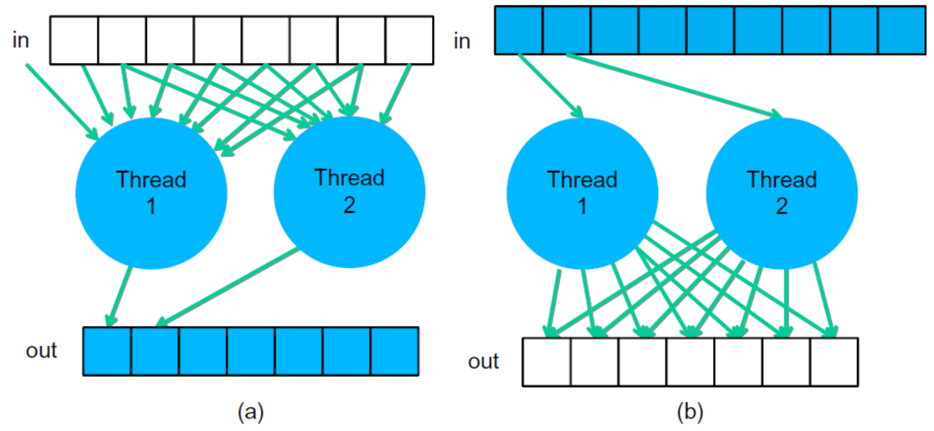


FIGURE 13.1

(a) Gather and (b) scatter based thread arrangements.

Programming Massively Parallel Processors – David B. Kirk

Special abstraction in Tensorization to support 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

Special abstraction in Tensorization to support Hardware Intrinsic

- Leverages the tensor expression language to **explicitly declare** the **behaviour** of the **HW intrinsic and the lowering rule**
- Enables **integration** of **new intrinsic operations** supported by **custom accelerators**, or **hand-crafted micro-kernels**
- Accepts **inputs of arbitrary dimensions**, matching to the **data layout required** by HW accelerator
- **Decouples scheduling from specific hardware primitive (Halide scheduling primitive extended within TVM)**

DL Compiler System Architecture

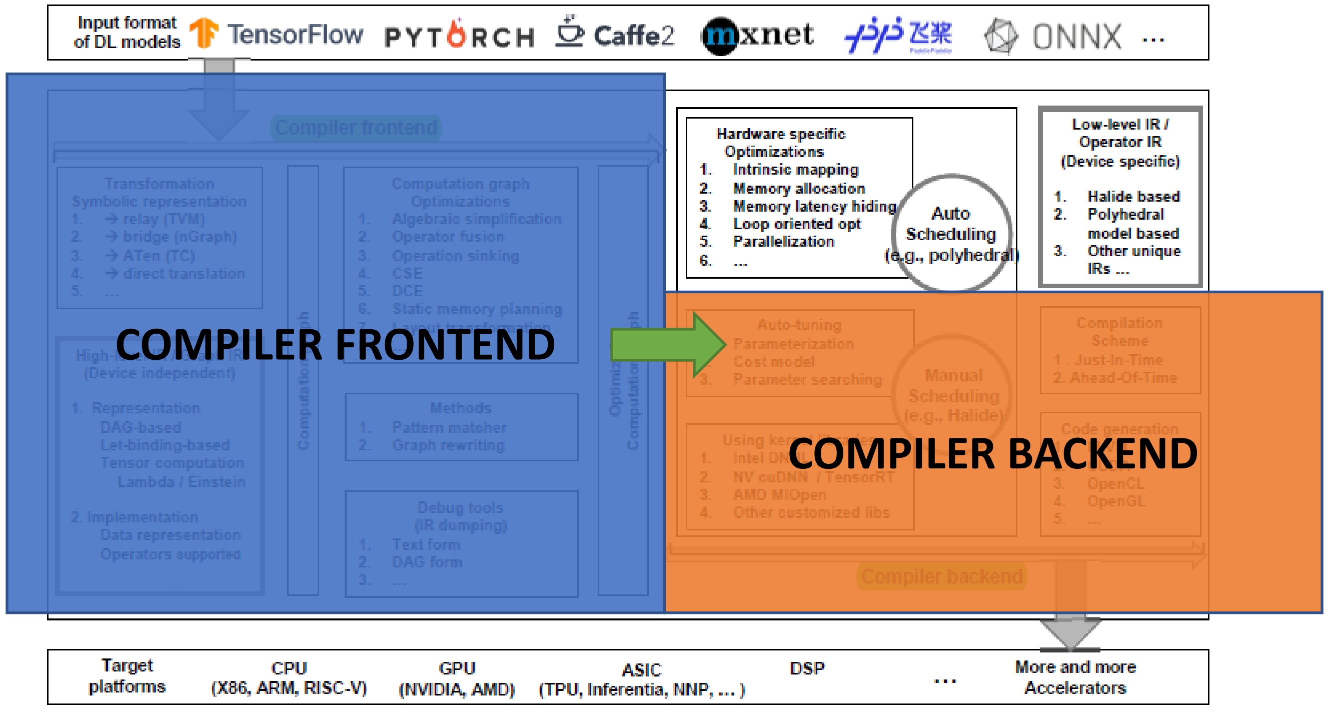
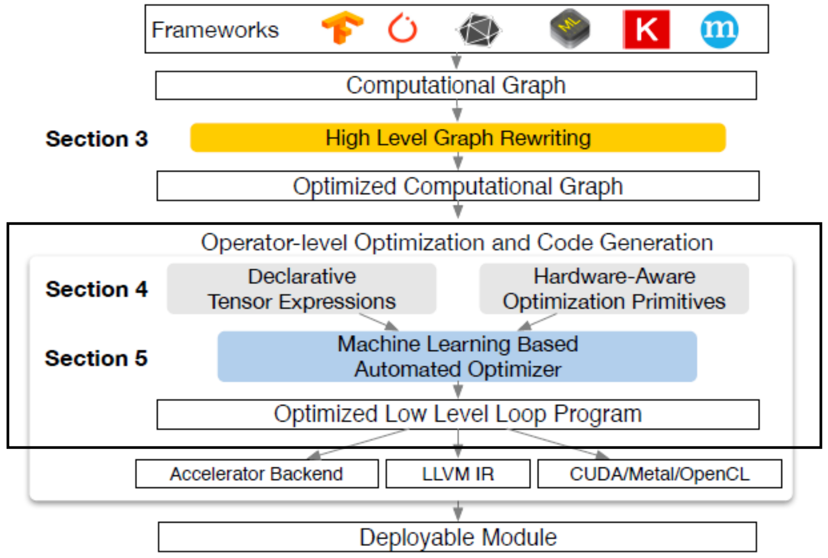


Fig. 2. The overview of commonly adopted design architecture of DL compilers.



Frameworks
Computational Graph
Section 3
High Level Graph Rewriting
Optimized Computational Graph
Operator-level Optimization and Code Generation
Section 4
Declarative Tensor Expressions
Hardware-Aware Optimization Primitives
Section 5
Machine Learning Based Automated Optimizer
Optimized Low Level Loop Program
Accelerator Backend
LLVM IR
CUDA/Metal/OpenCL
Deployable Module

Figure 2: System overview of TVM. The current stack supports descriptions from many deep learning frameworks and exchange formats, such as CoreML and ONNX, to target major CPU, GPU and specialized accelerators.

The Deep Learning Compiler: A Comprehensive Survey
https://arxiv.org/abs/2002.03794

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

DL Compiler System Architecture

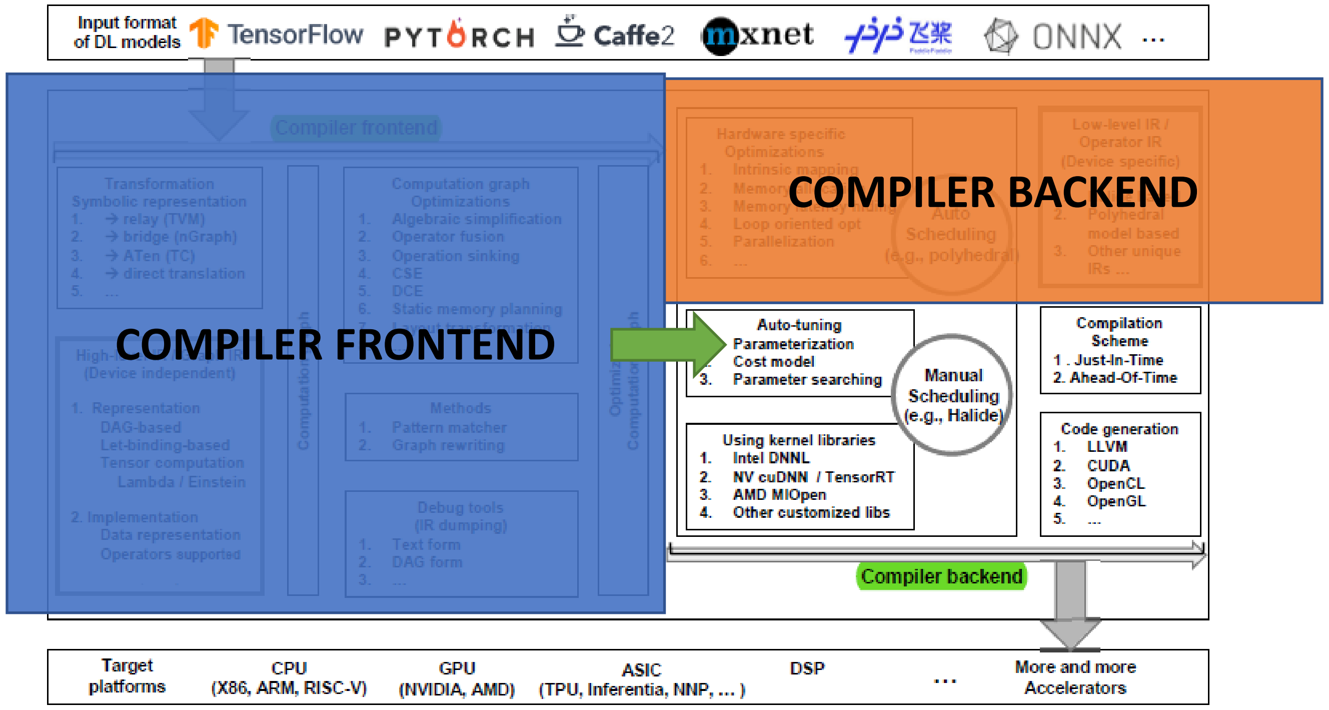


Fig. 2. The overview of commonly adopted design architecture of DL compilers.

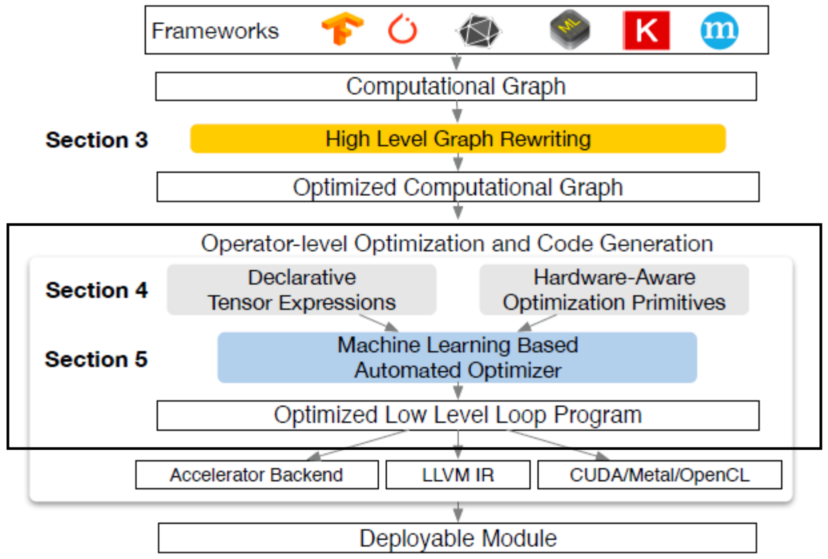

Frameworks
Computational Graph
Section 3
High Level Graph Rewriting
Optimized Computational Graph
Operator-level Optimization and Code Generation
Section 4
Declarative Tensor Expressions
Hardware-Aware Optimization Primitives
Section 5
Machine Learning Based Automated Optimizer
Optimized Low Level Loop Program
Accelerator Backend
LLVM IR
CUDA/Metal/OpenCL
Deployable Module

Figure 2: System overview of TVM. The current stack supports descriptions from many deep learning frameworks and exchange formats, such as CoreML and ONNX, to target major CPU, GPU and specialized accelerators.

The Deep Learning Compiler: A Comprehensive Survey
https://arxiv.org/abs/2002.03794

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

Auto-tune: Optimizing Operator Kernels

Automated Schedule Optimizer

Goal: To find optimal operator implementation

1. **Schedule explorer** – develops search space by proposing promising new configurations based on factors such as (developer may define):
    1. Modifying loop order
    2. Optimizing for memory hierarchy
    3. Tiling size
    4. Loop unrolling factor
2. **ML cost model** – predicts performance of a given configuration and is updated through repetitive benchmarking on hardware target with varied workloads

ML Based Cost Model

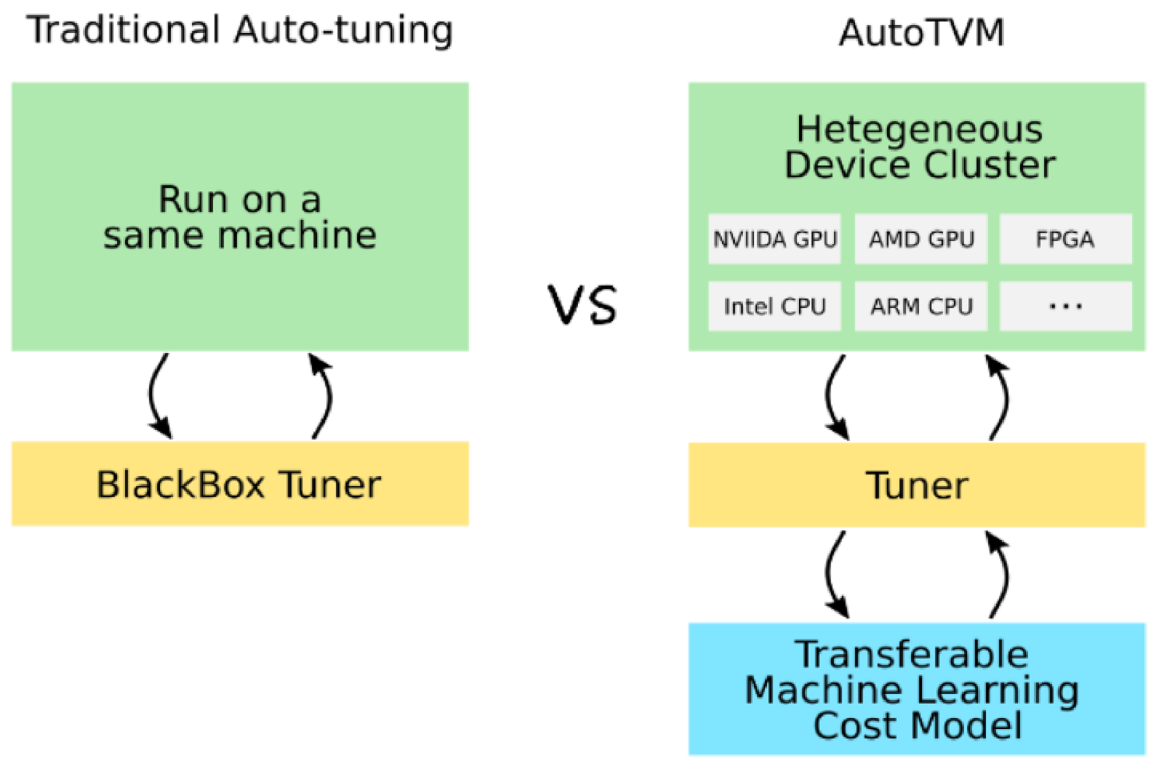
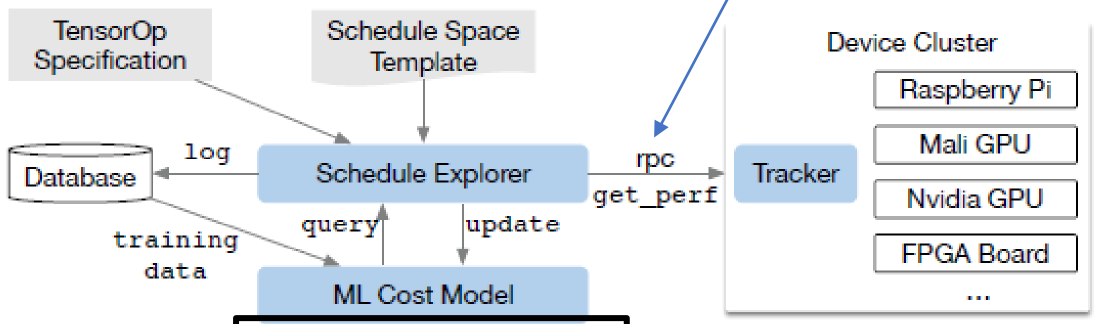


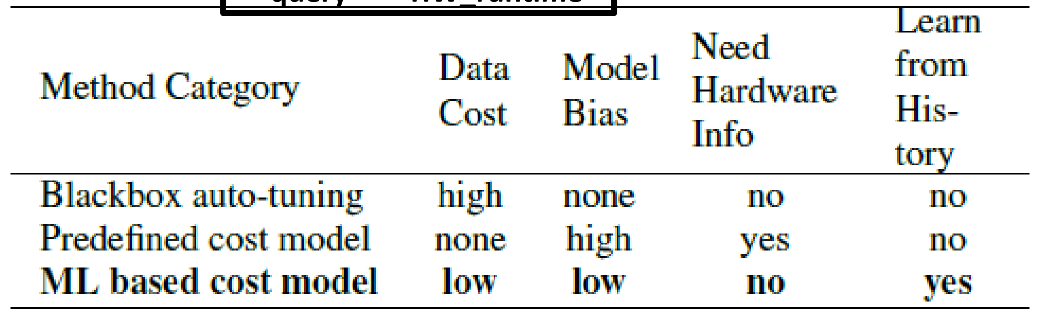
Figure 2. Comparison of Traditional Auto-tuning and AutoTVM

https://tvm.apache.org/2018/10/03/auto-opt-all

Via RPC interface with remote device over IP or cross-compiled target via JTAG and retrieve performance metrices



$t_{query} < t_{HW\_runtime}$



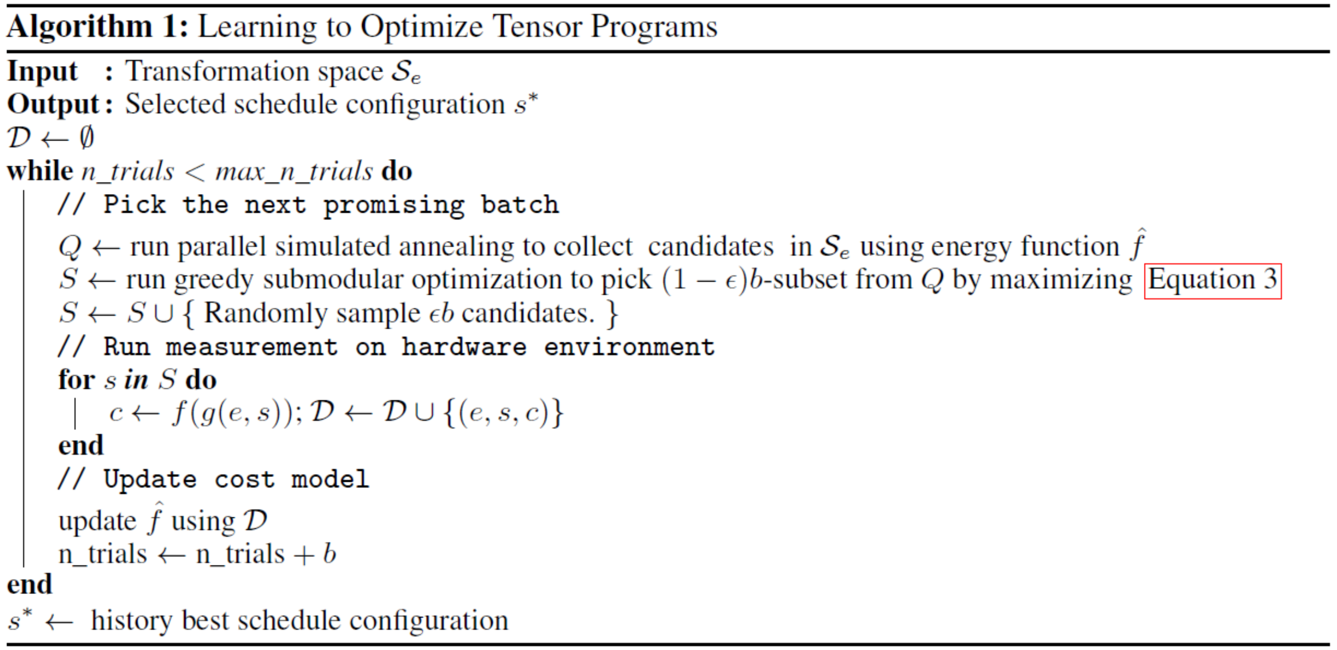
Method Category	Data Cost	Model Bias	Need Hardware Info	Learn from His-tory
Blackbox auto-tuning	high	none	no	no
Predefined cost model	none	high	yes	no
ML based cost model	**low**	**low**	**no**	**yes**

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

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

ML Based Cost Model

- To approximate target hardware performance when considering candidate kernel designs
- GBT and TreeRNN based search to evaluate features: memory access count, memory buffer re-use ratio, loop structure [vectorized, unrolled, parallel], etc



```
Algorithm 1: Learning to Optimize Tensor Programs
Input  : Transformation space $\mathcal{S}_e$
Output : Selected schedule configuration $s^*$
$\mathcal{D} \leftarrow \emptyset$
while $n\_trials < max\_n\_trials$ do
    // Pick the next promising batch
    $Q \leftarrow$ run parallel simulated annealing to collect candidates in $\mathcal{S}_e$ using energy function $\hat{f}$
    $S \leftarrow$ run greedy submodular optimization to pick $(1-\epsilon)b$-subset from $Q$ by maximizing Equation 3
    $S \leftarrow S \cup \{$ Randomly sample $\epsilon b$ candidates. $\}$
    // Run measurement on hardware environment
    for $s$ in $S$ do
        $c \leftarrow f(g(e,s)); \mathcal{D} \leftarrow \mathcal{D} \cup \{(e,s,c)\}$
    end
    // Update cost model
    update $\hat{f}$ using $\mathcal{D}$
    n_trials $\leftarrow$ n_trials $+ b$
end
$s^* \leftarrow$ history best schedule configuration
```

More in Wednesday's talk: Learning to Optimize Tensor Programs

TVM vs Competitors: GPU, ARM Hardware Speed-up

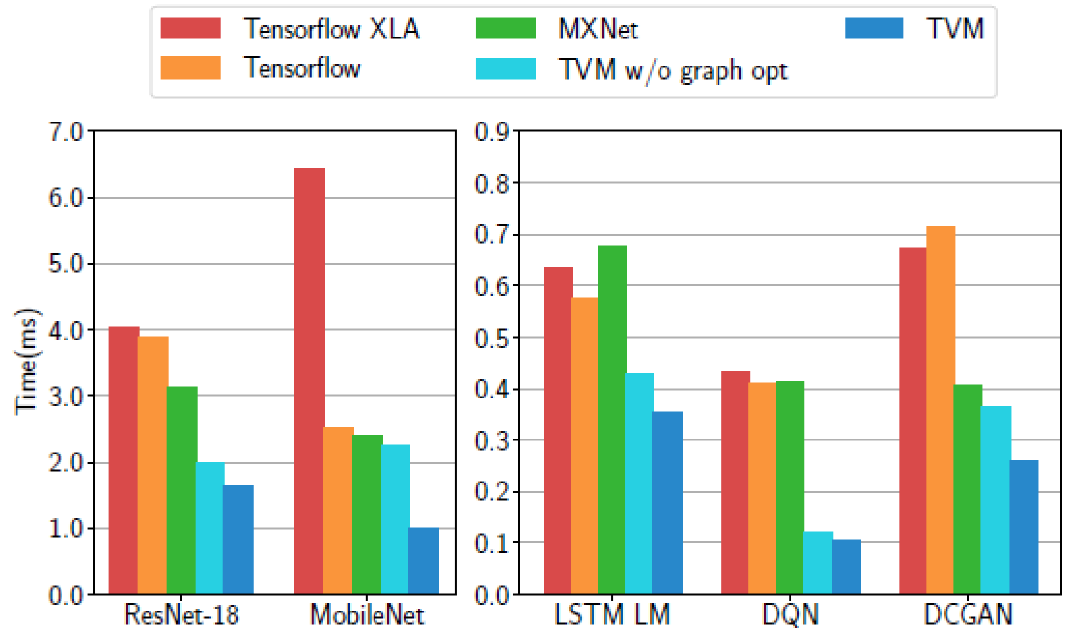


Figure 14: GPU end-to-end evaluation for TVM, MXNet, Tensorflow, and Tensorflow XLA. Tested on the NVIDIA Titan X.

1.6X to 3.8X Speed-up compared to baselines

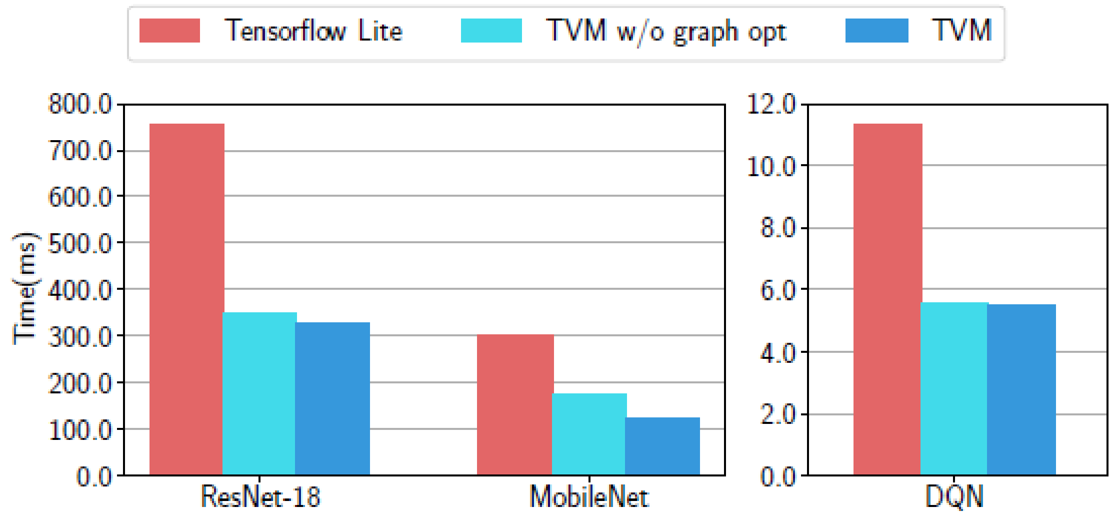


Figure 16: ARM A53 end-to-end evaluation of TVM and TFLite.

TFLite handcrafted kernels compared to TVM AutoTune

TVM vs ARM Compute Library Speed-up

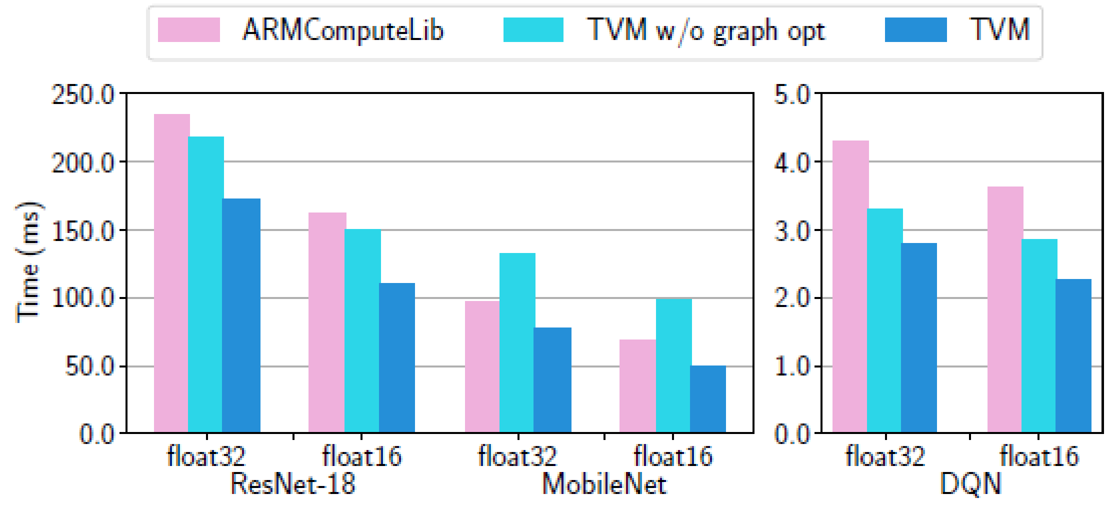


Figure 19: End-to-end experiment results on Mali-T860MP4. Two data types, float32 and float16, were evaluated.

ARM Compute Library with ARMNN integration is ARM's optimized kernels for operator execution on their hardware targets (Cortex and Mali). TVM achieves 1.2X to 1.6X speedup on the Mali GPU with FP32 and FP16.

DL Compiler System Architecture

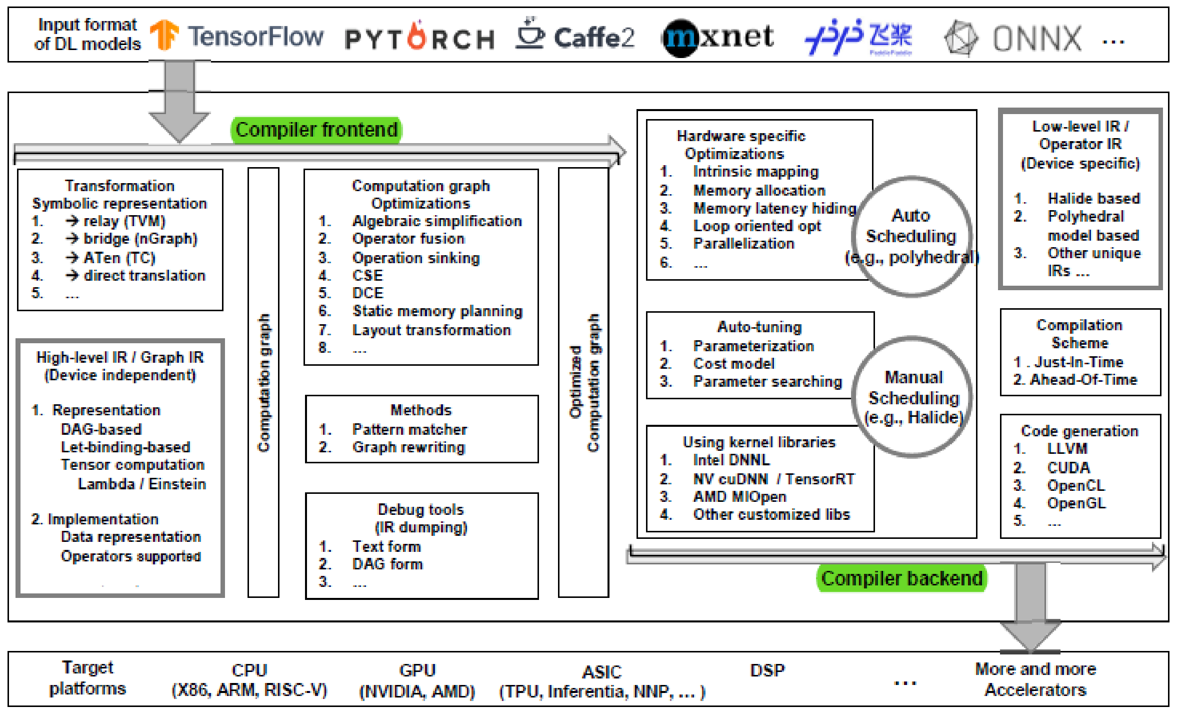
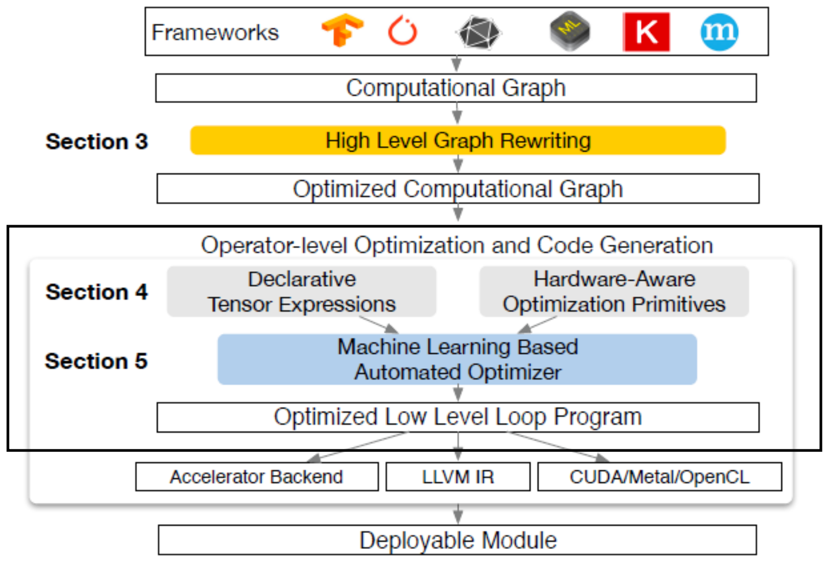


Fig. 2. The overview of commonly adopted design architecture of DL compilers.

The Deep Learning Compiler: A Comprehensive Survey
https://arxiv.org/abs/2002.03794



Frameworks
Computational Graph
Section 3
High Level Graph Rewriting
Optimized Computational Graph
Operator-level Optimization and Code Generation
Section 4
Declarative Tensor Expressions
Hardware-Aware Optimization Primitives
Section 5
Machine Learning Based Automated Optimizer
Optimized Low Level Loop Program
Accelerator Backend
LLVM IR
CUDA/Metal/OpenCL
Deployable Module

Figure 2: System overview of TVM. The current stack supports descriptions from many deep learning frameworks and exchange formats, such as CoreML and ONNX, to target major CPU, GPU and specialized accelerators.

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

Discussion Questions

1. While TVM achieves best-in-class speed-up for inference time compared to baselines, is model accuracy preserved through the compilation?

2. The compiler platforms search for optimal kernels using the ML based cost technique to approximate the behaviour of a kernel on hardware. The true architecture of the accelerator is opaque.

    a. Is this approach specific to an architecture and hence gated by code generators like LLVM/CUDA?

    b. Can this technique be generalized across a family of chips that share an ISA, although microarchitecture could be different?

    c. Can this technique be extended to explore optimal device assignment for distributed compute within a MPSoC. Eg: Xilinx Ultrascale with ARM Cortex-A, Cortex-R and Mali GPU core?

3. AutoTVM approach requires profiling on physical hardware, and feedback to tune cost function. To what extent can this system be augmented with virtual hardware to perform hardware-in-the-loop Auto-Tune?

4. Can TVM be included into training phase to provide feedback and develop an unconditionally stable model with guaranteed performance metrics at inference time?

    - Metrics: DC power cost, memory footprint, inference latency, accuracy preservation within tolerance