ArcSysU Weekly Paper Sharing

Hidet: Task-Mapping Programming Paradigm for Deep Learning Tensor Programs

Yaoyao Ding*†
University of Toronto
Toronto, Canada
yaoyao@cs.toronto.edu

Cody Hao Yu
Amazon Web Services
Santa Clara, USA
hyuz@amazon.com

Bojian Zheng†
University of Toronto
Toronto, Canada
bojian@cs.toronto.edu

Yizhi Liu
Amazon Web Services
Santa Clara, USA
yizhiliu@amazon.com

Yida Wang
Amazon Web Services
Santa Clara, USA
wangyida@amazon.com

Gennady Pekhimenko†
University of Toronto
Toronto, Canada
pekhimenko@cs.toronto.edu

ASPLOS'23

中山大學 SUN YAT-SEN UNIVERSITY

Contents

中山大學
SUN YAT-SEN UNIVERSITY

About the Authors

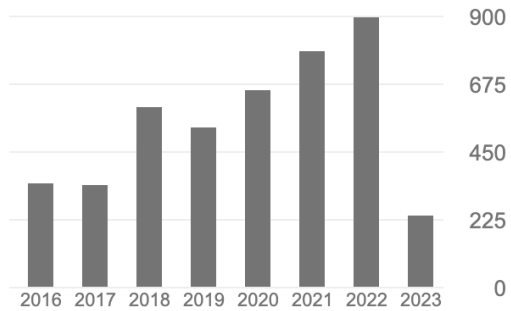
Yaoyao Ding

- Homepage: https://www.cs.toronto.edu/~yaoyao/
- 1st Year PhD at University of Toronto
- Bachelor from Shanghai Jiao Tong University (at ACM Honors Class)
- MLSys'21: IOS: Inter-Operator Scheduler for CNN Acceleration

Gennady Pekhimenko

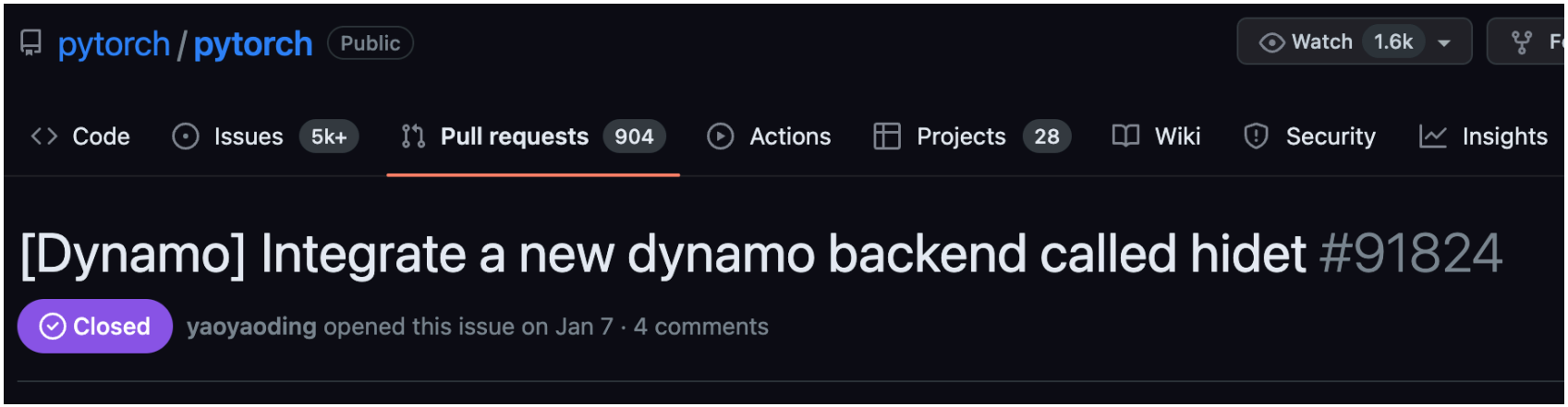
- Homepage: http://www.cs.toronto.edu/~pekhimenko/

	All	Since 2018
Citations	4712	3716
h-index	32	32
i10-index	50	48



About the Project

- Repository: https://github.com/hidet-org/hidet.
- Plug-and-play with PyTorch[1] (20k LOC in Python and C++): `torch.compile(model, backend='hidet')`.
- Proposed a PR to PyTorch, but rejected.



[1] PyTorch 2.0 experimental preview released on 2022/12/02, the Hidet version on artifact evaluation used PyTorch 1.11. By now it has integrated with PyTorch 2.0's `compile` feature

TVM's Loop-Oriented Scheduling

Computation

```
C = compute([M, N], lambda i, j: reduce([K], A[i, k] * B[k, j]))
```

❶ Generate Default Tensor Program

Default Program

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

❷ Apply **Declarative** Scheduling Primitives

Schedule Primitives	Original Program	Scheduled Program

fuse(i, j)	for i in range(128):
	  for j in range(4):
	  body(i, j)
		for ij in range(512):
		  body(ij / 4, ij % 4)

split(i, 128)	for i in range(512):
	  body(i)
		for oi in range(4):
		  for ii in range(128):
		  body(oi * 128 + ii)

reorder(i, j)	for i in range(128):
	  for j in range(4):
	  body(i, j)
		for j in range(4):
		  for i in range(128):
		  body(i, j)

bind(i, threadIdx.x)	for i in range(128): body(i)	body(threadIdx.x)

TVM's Loop-Oriented Scheduling (cont.)

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

Default Program

1

```
oi, ii = split(i, 64)
oj, ij = split(j, 64)
```

```
for io in range(M/64):
  for ii in range(64):
    for oj in range(N/64):
      for ij in range(64):
        for k in range(K):
          i, j = io * 64 + ii, jo * 64 + ij
          C[i, j] += A[i, k] * B[k, j]
```

2

```
reorder(oi, oj, ii, ij)
```

```
for io in range(M/64):
  for oj in range(N/64):
    for ii in range(64):
      for ij in range(64):
        for k in range(K):
          i, j = io * 64 + ii, jo * 64 + ij
          C[i, j] += A[i, k] * B[k, j]
```

3

```
bind(oi, blockIdx.x)
bind(oj, blockIdx.y)
```

```
for ii in range(64):
  for ij in range(64):
    for k in range(K):
      i = blockIdx.x * 64 + ii
      j = blockIdx.y * 64 + ij
      C[i, j] += A[i, k] * B[k, j]
```

Scheduled Program

TVM's Loop-Oriented Scheduling (cont.)

Computation

```
C = compute([M, N], lambda i, j: reduce([K], A[i, k] * B[k, j]))
```

❶ Generate Default Tensor Program

Default Program

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

❷ Apply **Declarative** Scheduling Primitives (called Scheduling)

```
1. oi, ii = split(i, 64)      4. bind(oi, blockIdx.x)
2. oj, ij = split(j, 64)      5. bind(oj, blockIdx.y)
3. reorder(oi, oj, ii, ij)    6. ...
```

Two Scheduling Methods:

- Template-based: AutoTVM [1]
- Rule-based: Ansor [2]

Kernel Tuning:
Search best schedule hyper-parameter

Scheduled Program

```
for ii in range(64):
  for ij in range(64):
    for k in range(K):
      C[i, j] += A[ii + blockIdx.x * 64, k] * B[k, ij + blockIdx.y * 64]
```

Declarative Loop-Oriented Scheduling

Contents

Non Loop-Oriented Optimization

An example of tiled matrix multiply:

```
1  def matmul(A: fp32[M, K], B: fp32[K, N], C: fp32[M, N]):
2      SmemA, SmemB = shared fp32[64, 8], fp32[8, 64]
3      RegsC = local fp32[...]
4      ... # Step ❶: Calculate sub-task offset
5      for k0 in range(128):   # Iterate each K tile
6          # Step ❷: Load A and B frag. to SmemA and SmemB
7          SmemA, SmemB = cooperative_load(A, B, k0)
8          sync_threads()
9          # Step ❸: Block MMA (RegsC = SmemA * SmemB + RegsC)
10         RegsC = block_mma(SmemA, SmemB, RegsC)
11         sync_threads()
12     ... # Step ❹: Write back C (RegsC  => C)
```

Non Loop-Oriented Optimization (cont.)

Use double buffer to overlap computation and data transfer:

```
1   RegsA, RegsB = register fp32[...], fp32[...]
2   SmemA, SmemB = shared fp32[2, 64, 8], fp32[2, 8, 64]
3   ...
4   SmemA[0], SmemB[0] = cooperative_load(A, B, 0)
5   sync_threads()
6   for k0 in range(127):
7       p, q = k0 % 2, (k0 + 1) % 2
8       RegsA, RegsB = cooperative_load(A, B, k0 + 1)
9       RegsC = block_mma(SmemA[p], SmemB[p], RegsC)
10      SmemA[q], SmemB[q] = RegsA, RegsB
11      sync_threads()
12  RegsC = block_mma(SmemA[0], SmemB[0], RegsC)
13  ...
```

Two Buffers for A & B

Preloading **Next** Tile of A/B into **Regs**

Computation of **Current** Tile

Store Next Tile of A/B into **Shared** Memory

Non Loop-Oriented Optimization (cont.)

More on non *loop-oriented* optimization:

- Thread block swizzle[2].
- Tensor Core MMA primitive.
- Multi-stage asynchronous prefetching[3].

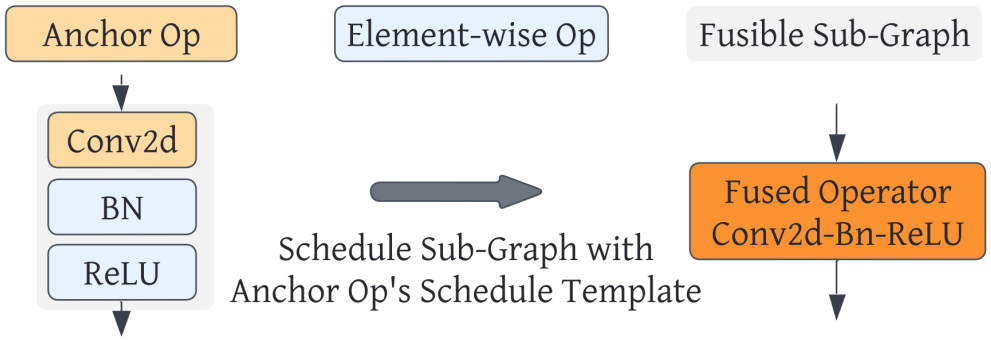
Challenge 1

TVM's existing loop-oriented scheduling primitives cannot manipulate the loop body at **fine granularity**.

[2] Louis Bavoil. 2020. Optimizing Compute Shaders for L2 Locality using Thread Group ID Swizzling. https://developer.nvidia.com/blog/optimizing-compute-shaders-for-l2-locality-using-thread-group-id-swizzling

[3] Andrew Kerr, Duane Merrill, Julien Demouth, et. al. 2018. CUTLASS: CUDA Template Library for Dense Linear Algebra at All Levels and Scales.

Dedicated Schedule Template for Fusion



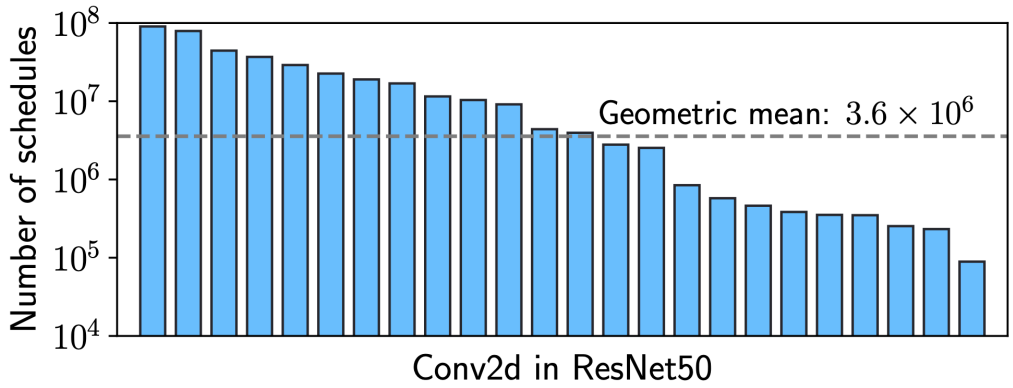
Workflow of TVM's subgraph fusion:

- Identify anchor operator, which is usually the most computate-intensive one.
- Use anchor operator's schedule template to schedule the whole subgraph.

Challenge 2

Fusion rules needs to be general enough to support all existing operators (and even new operators).

Long Tuning Time



Tiling factor is the majority of search space:

- Constrained to perfect tile size, e.g., $N = 2039$ will fail.
- Input size grows $\rightarrow$ Schedule space grows exponentially.
- Long tuning time $\rightarrow$ Fail to co-optimize with graph-level opt and neural architecture search.

Challenge 3

Large search space, long tuning time, input size sensitive.

Contribution

- **C1: Non loop-oriented optimization**
  Introduce fine-grained scheduling primitive[4] for tensor programs.
- **C2: Dedicated schedule template for fusion**
  Based on that primitive, post-scheduling fusion is straight-forward yet general.
- **C3: Long tuning time**
  Based on that primitive, search space could be narrowed down to hardware-centric.
- **Programming Effort**
  Implemented with PyTorch 2.0, conduct solid experiments.

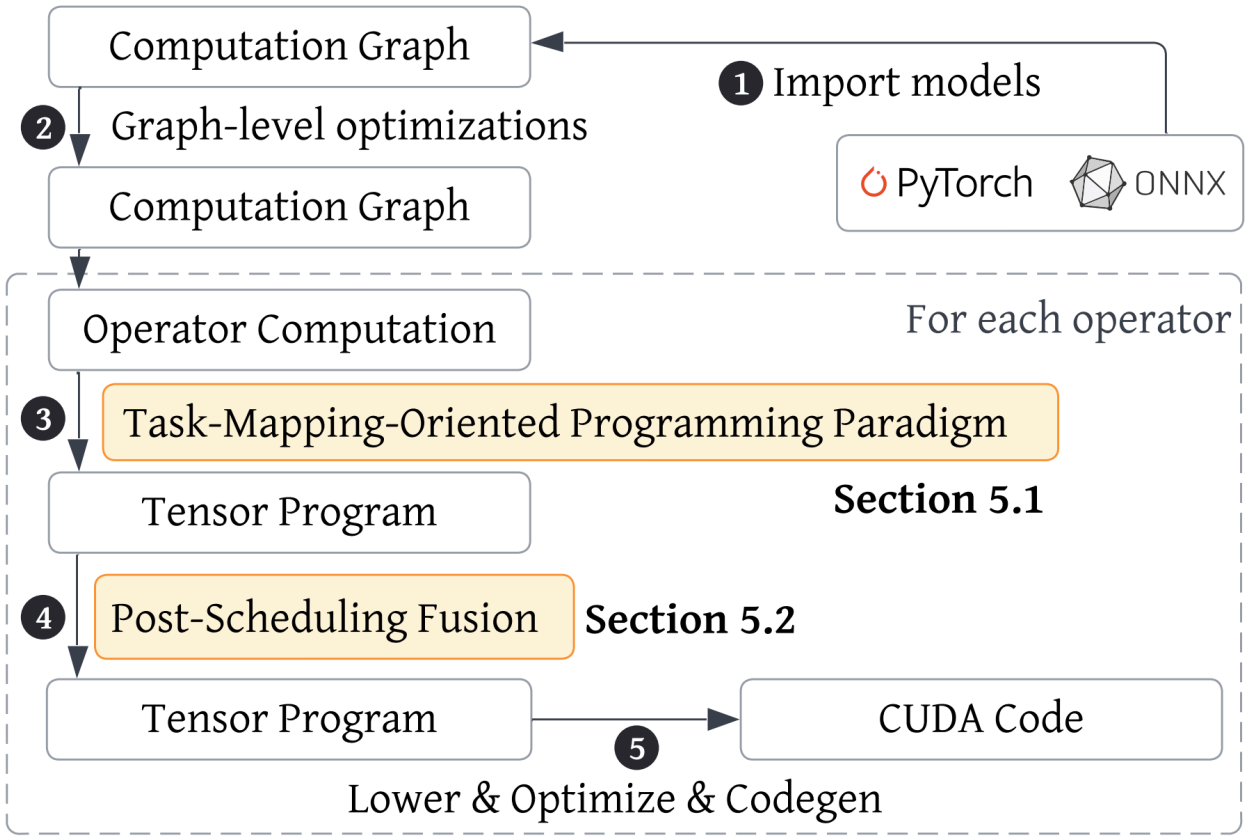
[4] The authors named it as *"task-mapping oriented programing paradigm"*.

Contents

中山大學
SUN YAT-SEN UNIVERSITY

Design Overview

- Task mapping: CUDA-like `thread block` abstraction.
- Tuning: hardware-centric search space.
- Fusion: pre-defined rules.
- Model → Graph-level IR → Tensor-level IR → CUDA C++.

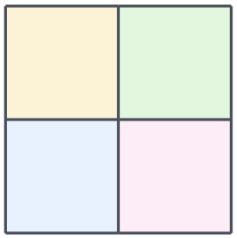


Key Idea: Task Mapping

- Similar to CUDA `thread-block` abstraction.
- Task map: worker ID $\rightarrow$ loop/array index.

0	1
2	3

Task execution order



Task Assignment

Legend

0	1
2	3

repeat(2, 2) f(w) = [(0, 0), (0, 1), (1, 0), (1, 1)]

Assign all tasks to a single worker (a)

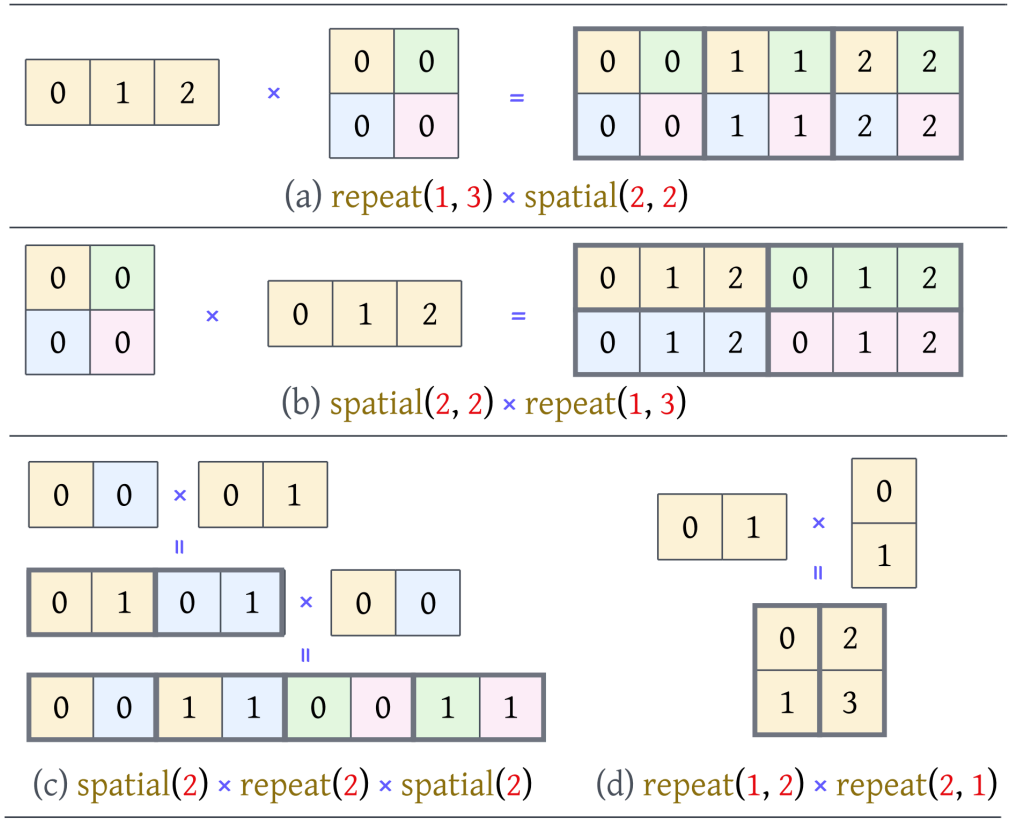
0	0
0	0

spatial(2, 2) f(w) = [(w / 2, w % 2)]

Assign each worker a task (4 workers in this case) (b)

Key Idea: Task Mapping (cont.)

- Can be combined and nested.
- Represents CUDA's `blockDim`, `threadDim` or grid-strided loop.



Key Idea: Task Mapping (cont.)

- Can be combined and nested.
- Represent CUDA's `blockDim`, `threadDim` or grid-strided loop.

```
# spatial(4, 2) * repeat(2, 2) * spatial(4, 8) * repeat(4, 4)

__global__ void kernel(){
    for bi in range(2):                          # repeat(2, 2)
        for bj in range(2):                      # repeat(2, 2)
            for ti in range(4):                  # repeat(4, 4)
                for tj in range(4):              # repeat(4, 4)
                    body(bi, bj, ti, tj);        # task body
}

int main(){
    dim3 blockDim(4, 2);
    dim3 threadDim(4, 8);
    kernel<<<blockDim, threadDim>>>();
}
```

From Loop-Oriented Scheduling to Task Mapping

TVM's loop scheduling primitives

Schedule Primitives	Original Program	Scheduled Program

fuse(i, j)	for i in range(128):
	  for j in range(4):
	  body(i, j)
		for ij in range(512):
		  body(ij / 4, ij % 4)

split(i, 128)	for i in range(512):
	  body(i)
		for oi in range(4):
		  for ii in range(128):
		  body(oi * 128 + ii)

reorder(i, j)	for i in range(128):
	  for j in range(4):
	  body(i, j)
		for j in range(4):
		  for i in range(128):
		  body(i, j)

bind(i, threadIdx.x)	for i in range(128): body(i)	body(threadIdx.x)

From Loop-Oriented Schd. to Task Mapping (cont.)

Task map: worker ID $\rightarrow$ loop/array index

```
Smem = compute([64, 8], lambda i, j => Gmem[i, j])
```

1 Default Program

```
for i in range(64):
  for j in range(8):
    Smem[i, j] = Gmem[i, j]
```

2 Schedule

```
io, ii = split(i, 16)
iik = fuse(ii, k)
bind(iik, threadIdx.x)
```

3 Lower Primitives

Declarative Loop-Oriented Scheduling

1 Write Program with Task Mapping

```
mapping = repeat(4, 1).spatial(16, 8)
for i, j in mapping(threadIdx.x):
  Smem[i, j] = Gmem[i, j]
```

2 Lower Task Mappings

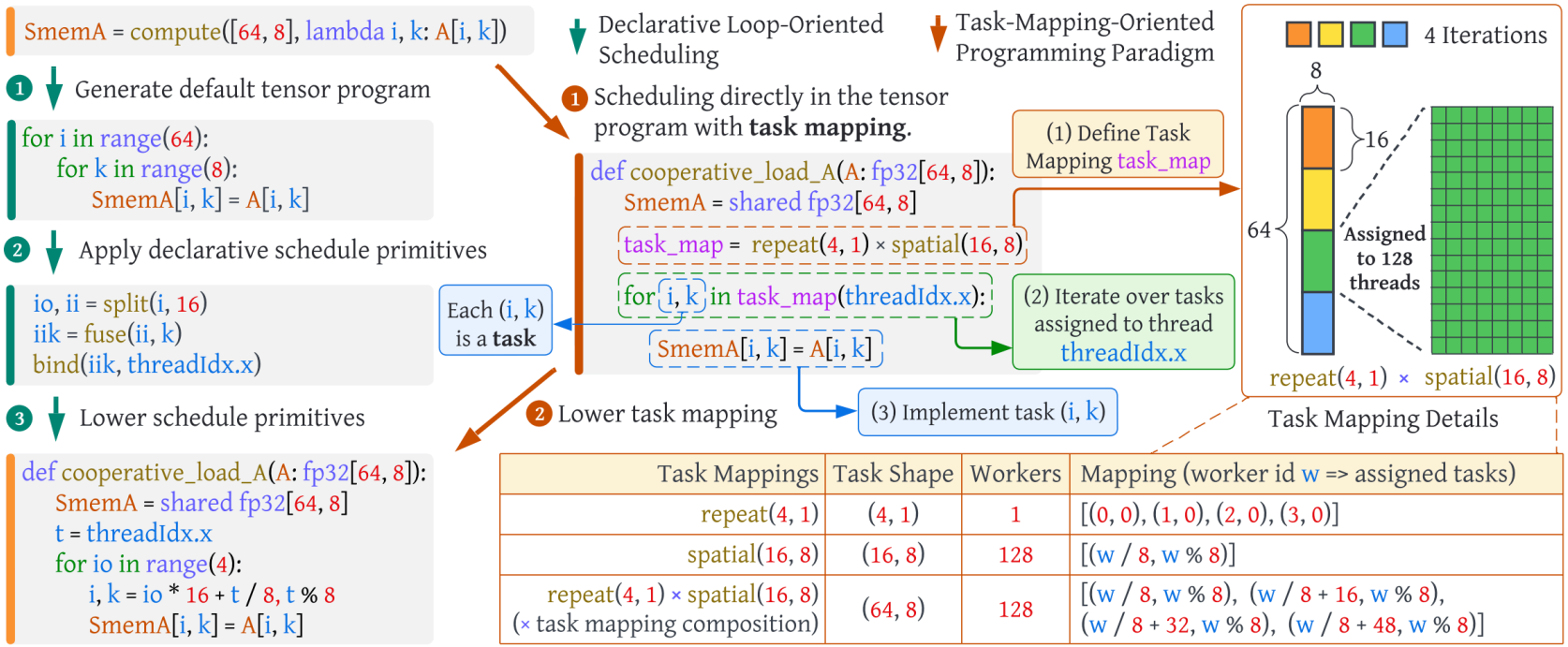
```
for io in range(4):
  t = threadIdx.x
  i, j = io * 16 + t / 8, t % 8
  Smem[i, j] = Gmem[i, j]
```

Task Mapping replaces **Schedule Primitives**

Task-Mapping Programming Paradigm

From Loop-Oriented Schd. to Task Mapping (cont.)

Putting it together



Task Mappings	Task Shape	Workers	Mapping (worker id w => assigned tasks)
repeat(4, 1)	(4, 1)	1	[(0, 0), (1, 0), (2, 0), (3, 0)]
spatial(16, 8)	(16, 8)	128	[(w / 8, w % 8)]
repeat(4, 1) × spatial(16, 8) (× task mapping composition)	(64, 8)	128	[(w / 8, w % 8), (w / 8 + 16, w % 8), (w / 8 + 32, w % 8), (w / 8 + 48, w % 8)]

Complex Example: MMA (Matrix Multiply Accumulate)

- Task: 16×16 grid
- Warp: worker
- Each warp compute 4 WMMA

```
1  def block_mma(SmemA: fp32[64, 8], SmemB: fp32[8, 64],
2                RegsC: fp32[4, 4, 4]):
3      RegsA, RegsB = register fp32[4], fp32[4]
4      task_map = spatial(2, 2) * repeat(2, 2)
5      worker_id = threadIdx.x / 32      # warp index
6      for i, j in task_map(worker_id):
7          wmma_load_a(&SmemA[i * 16, 0], RegsA)
8          wmma_load_b(&SmemB[0, j * 16], RegsB)
9          wmma_mma(RegsA, RegsB, RegsC[i, j])
```

Scheduling Machanism

Hidet use similar scheduling mechanism with AutoTVM(template-based) and Ansor(rule-based)[5]

	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`

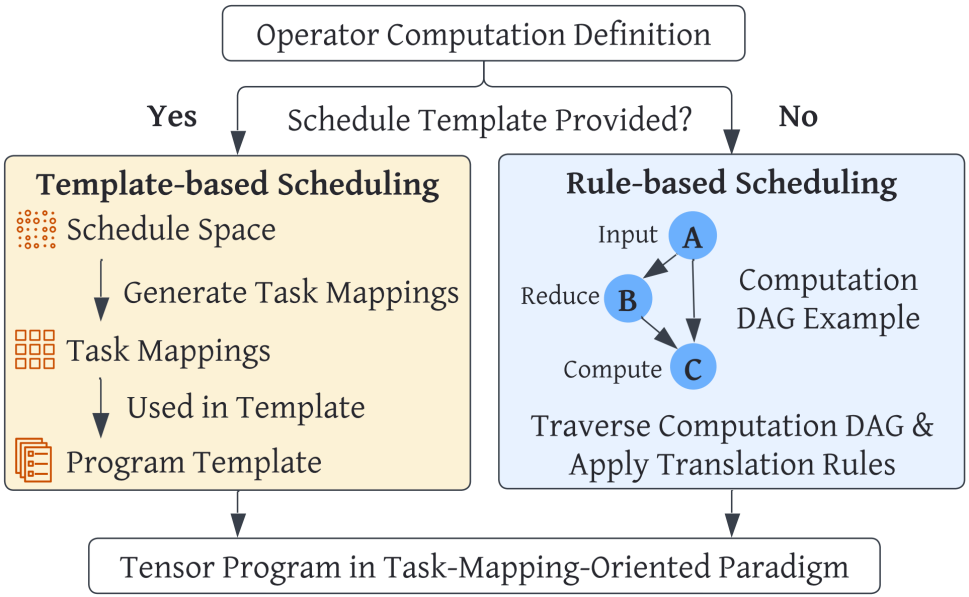
Step 2: Write a schedule template
 	
(difficult part)
	`# 20-100 lines of tricky DSL code`
	
	`# Define search space`
	`cfg.define_split("tile_x", batch, num_outputs=4)`
	`cfg.define_split("tile_y", out_dim, num_outputs=4)`
	`…`
	
	`# Apply config into the template`
	`bx, txz, tx, xi = cfg["tile_x"].apply(s, 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(…)`

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

Scheduling Machanism (cont.)

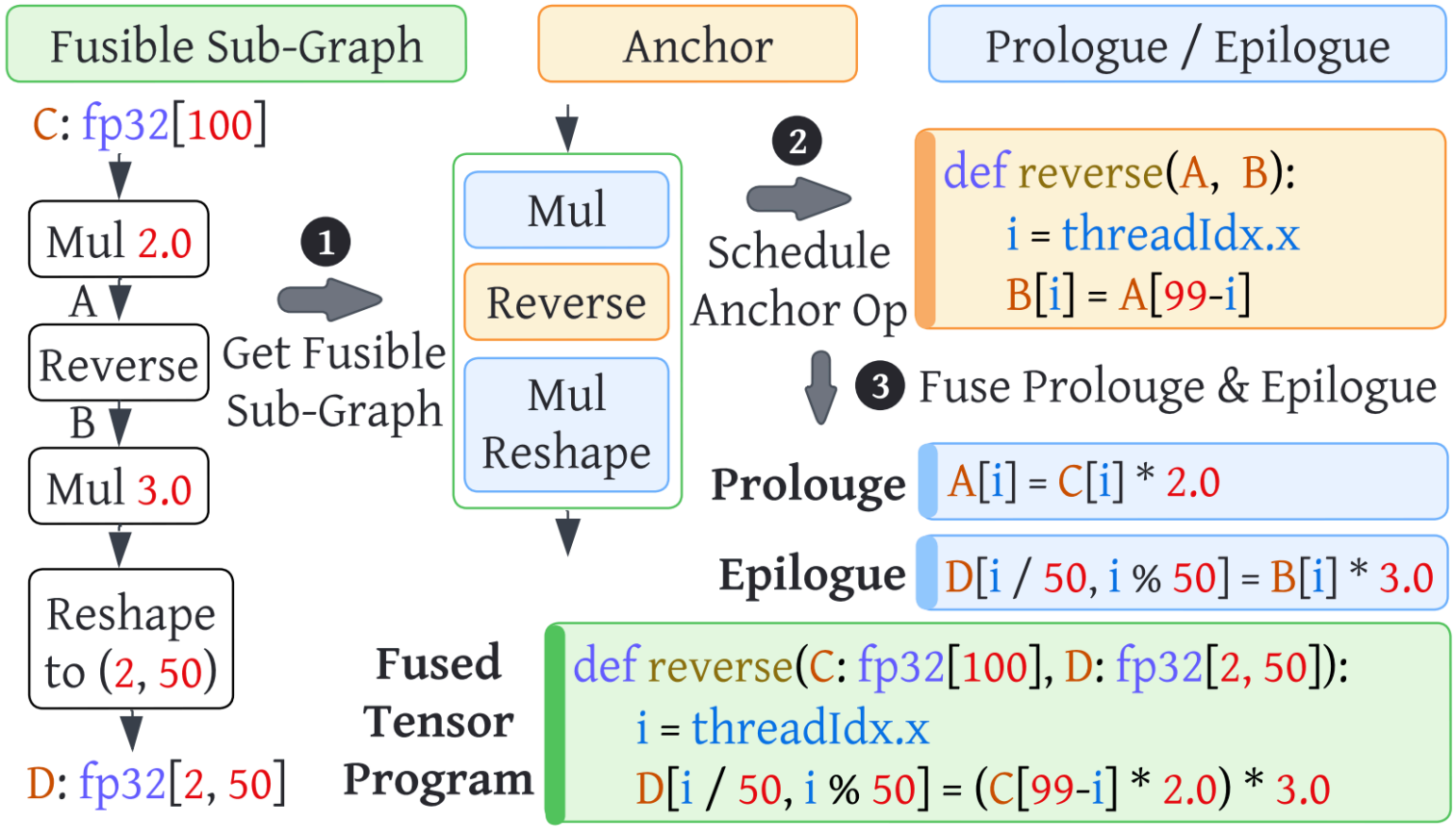
- **Rule-based**: Ansor-like, automatically translate DAG node by pre-defined rules, suitable for most of the operators (`element-wise`).
- **Template-based**: AutoTVM-like, requires manual effort, suitable for extreme-performance operators (`GEMM`, `conv`).
- Hidet only implemented GEMM and Reduce template, and they are able to cover all the operators in the evaluation.



Post-Scheduling Fusion

Rules to decide if an operator is fusible:

- Prologue: no reduction.
- Eplilogue: no reduction and is element-wise.



Contents

中山大學
SUN YAT-SEN UNIVERSITY

Setup

Platform

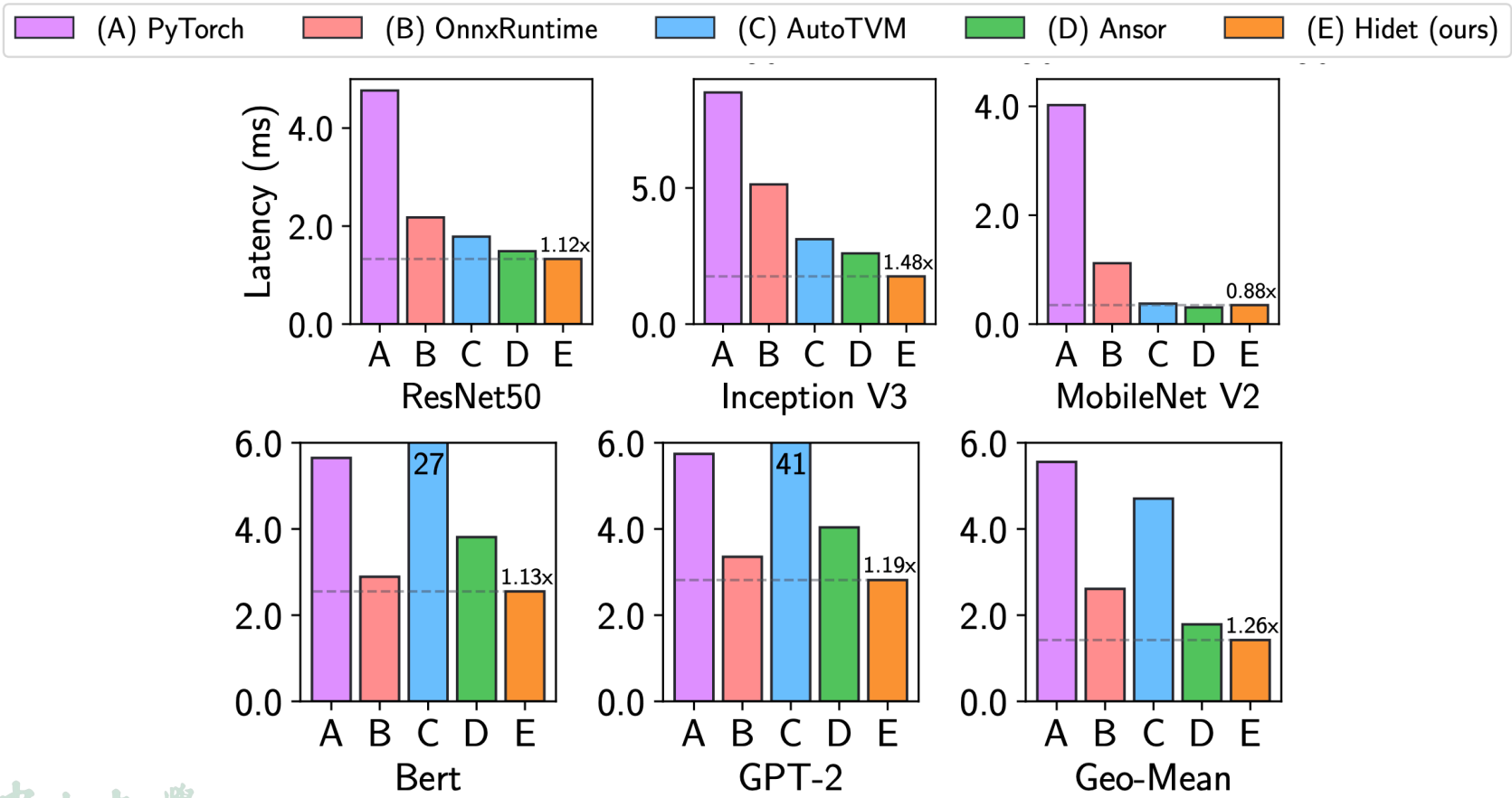
- Intel i9-12900K CPU (16-core, 24-thread, 64GB RAM)
- 1 $\times$ NVIDIA RTX 3090 GPU
- Ubuntu LTS 20.04, NVIDIA driver 510.73.08, CUDA 11.6

Workload

- Models from TorchVision, exported to ONNX
- AutoTVM/Ansor tuning trial: 1000/800 (official's suggestion)
- Baseline: Ansor
- Metric: Inference latency, tuning cost

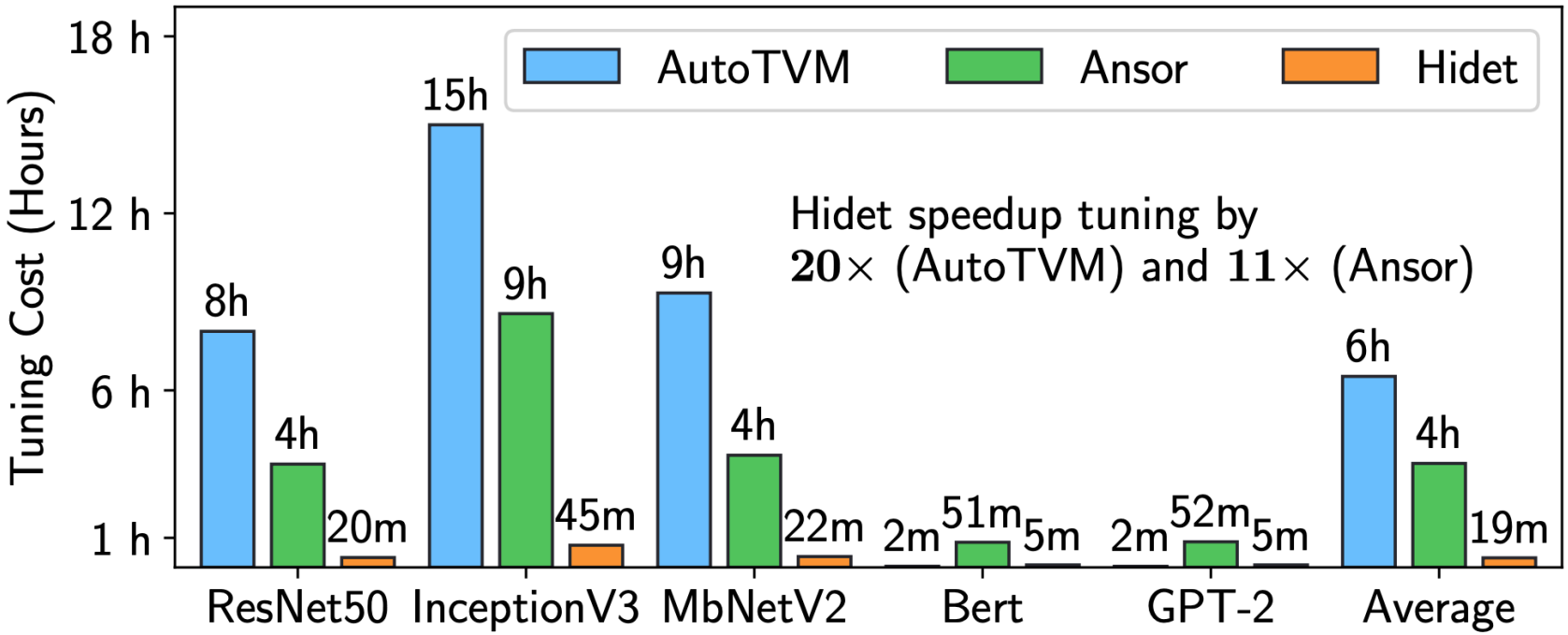
End to End Performance

- MobileNet: Ansor has better schedule on *deepwise convolution*.
- Bert and GPT: AutoTVM is lack of schedule template (< 20).



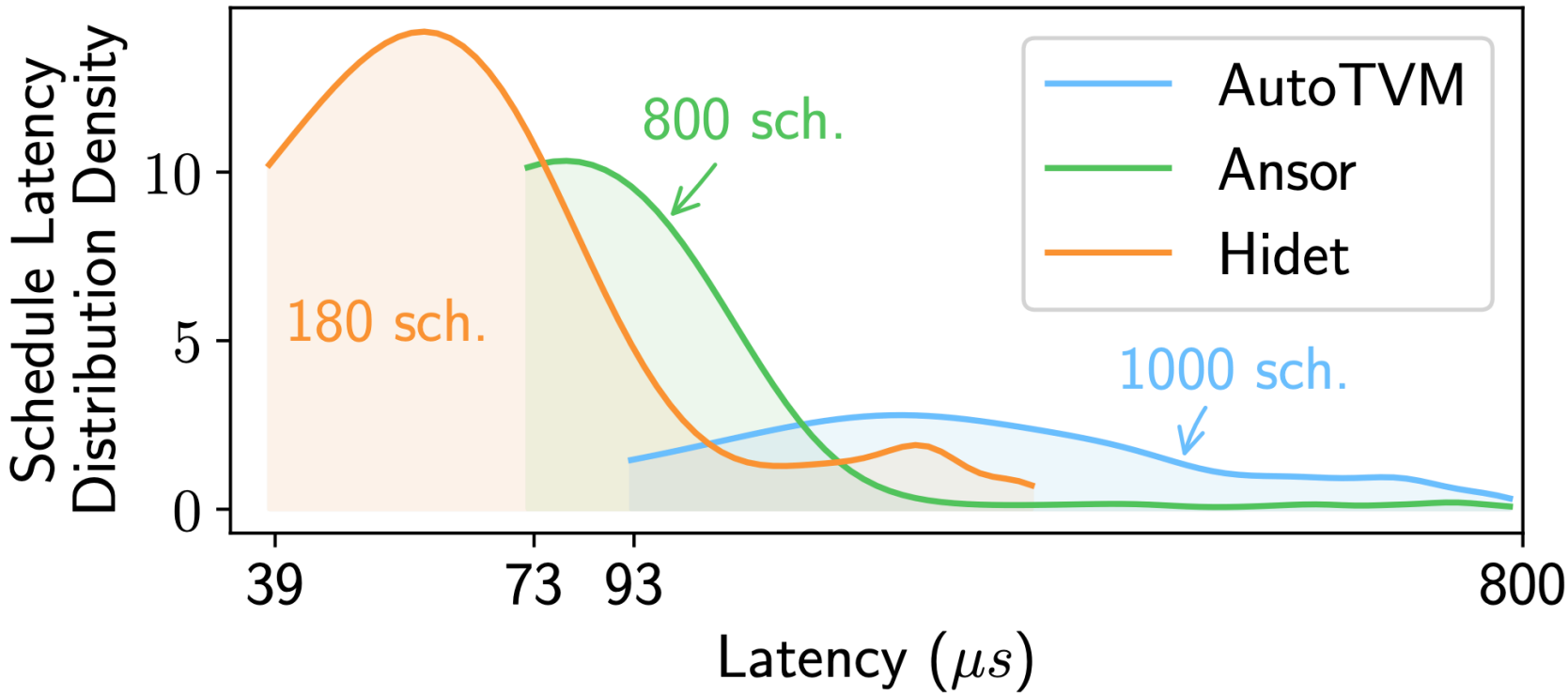
Tuning Cost

- Schedule space: AutoTVM 10^8, Ansor 10^5
- Hidet has small schedule space (180 for matrix multiplication), so can exhaust the schedule space while guarantee the performance.



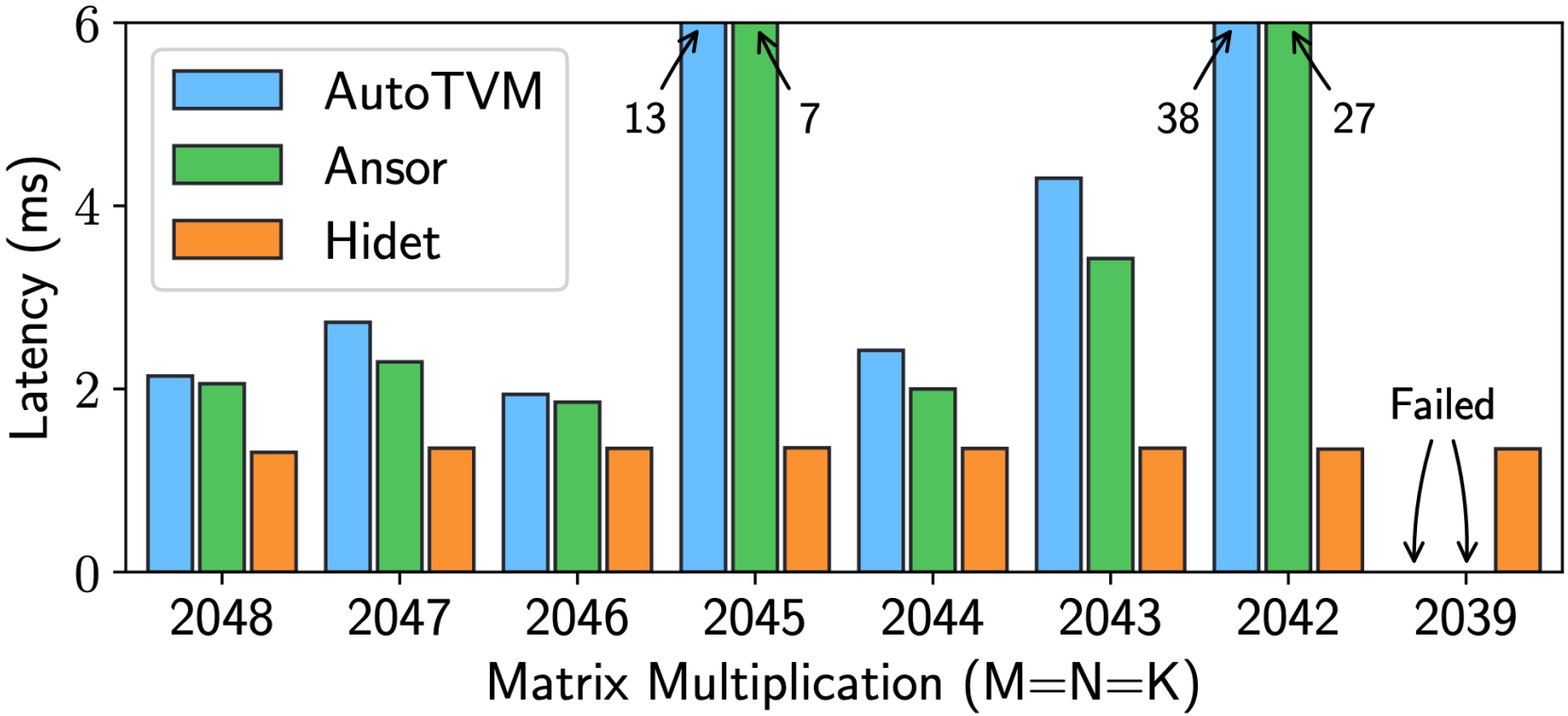
Quality of Schedule Space

- Workload: a conv in ResNet50, batch 1, input 28×28, channels 256, kernel 3×3, padding 1, stride 2.
- Figure below: distribution of the performance in the schedule space.



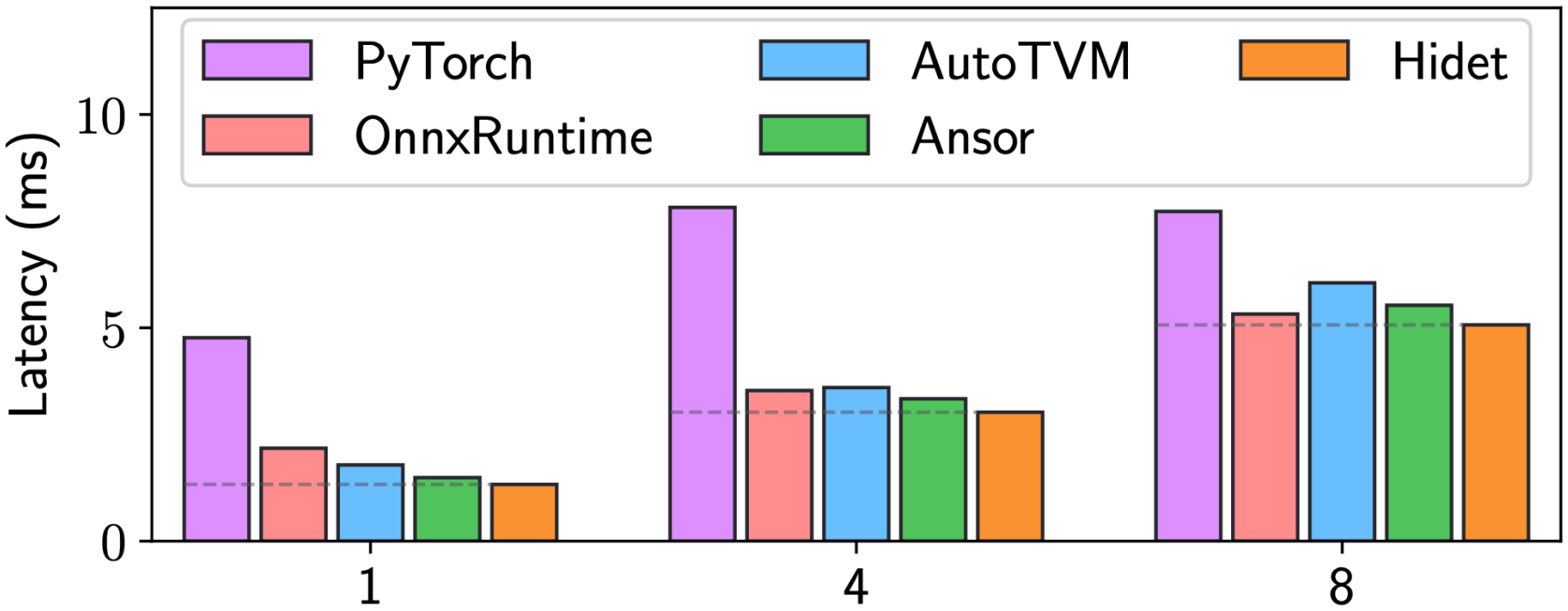
Performance Sensitivity Over Input Size

- AutoTVM and Ansor's scheduling is *input-centric*, while Hidet is *hardware-centric*.
- AutoTVM and Ansor fails for prime number 2039.



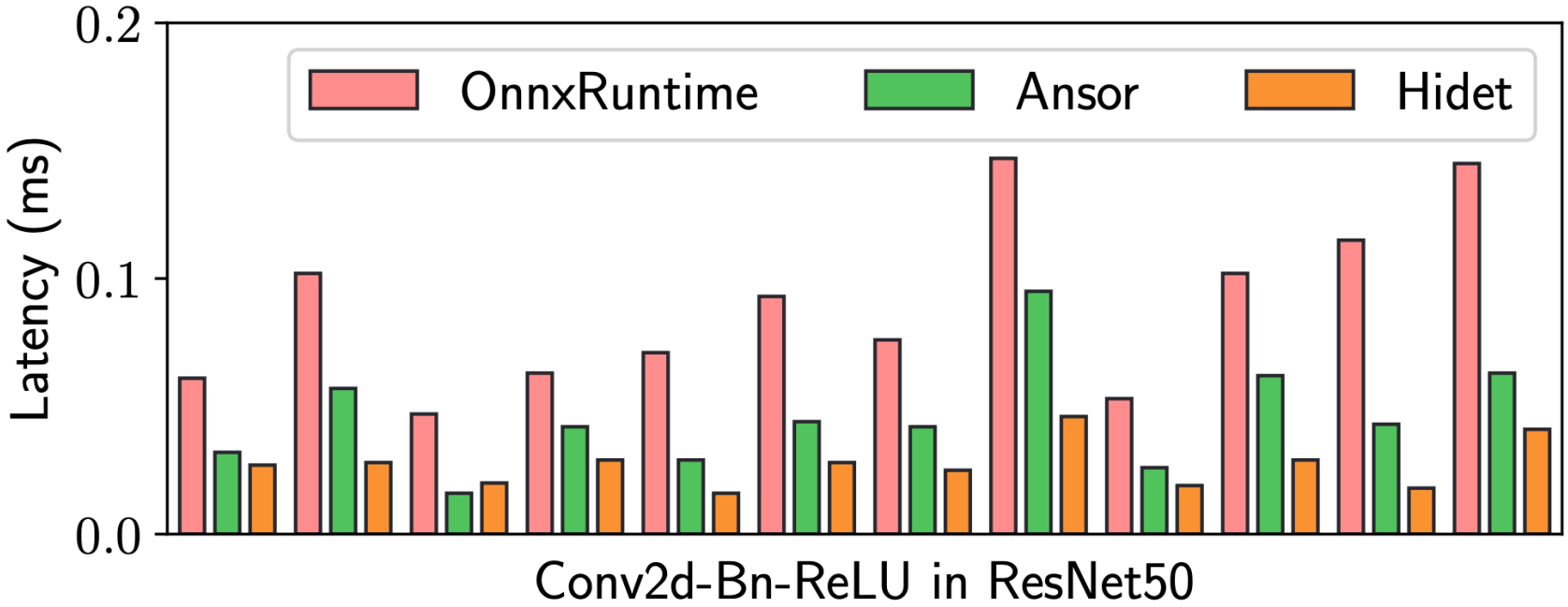
Performance Sensitivity Over Batch Size

- AutoTVM and Ansor: lack of non loop-oriented optimization.
- Why Hidet outperform Onnx?



Operator Fusion Performance

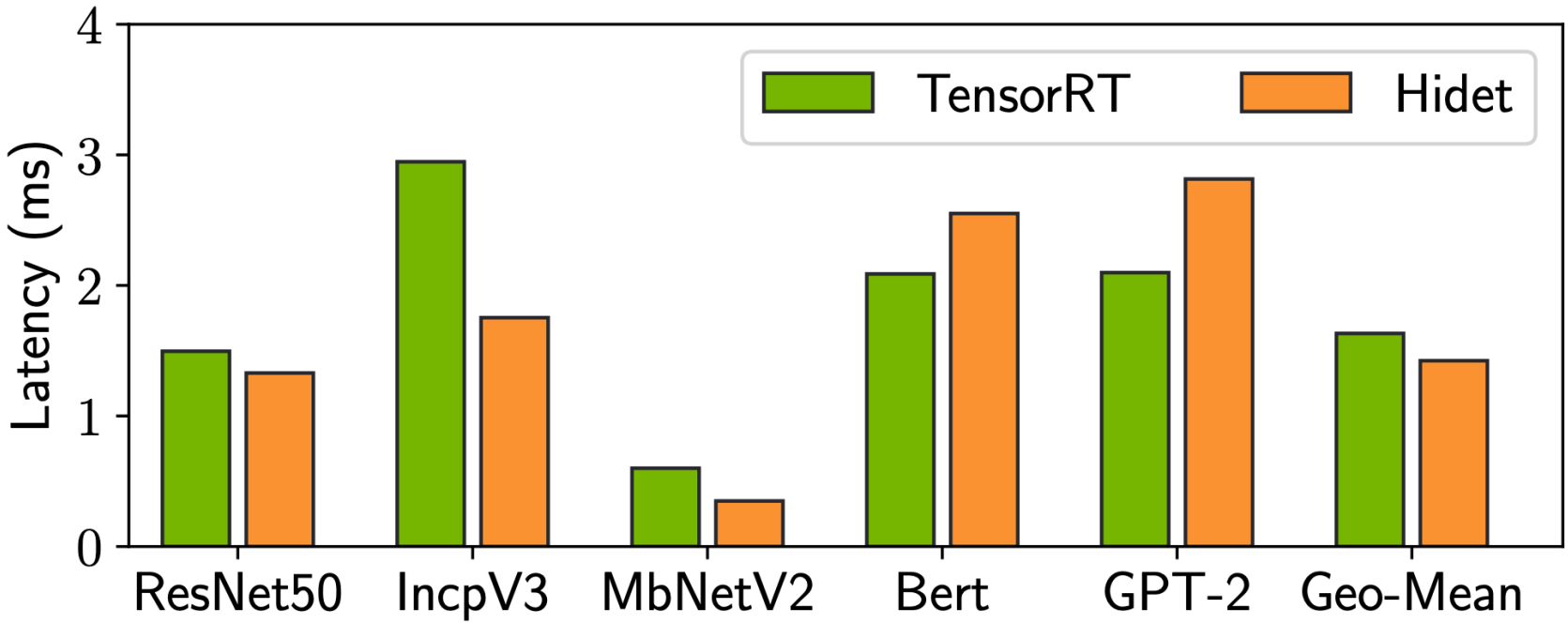
- Hidet implement `conv` by `img2col`[6] algorithm, using GEMM and reduce.
- Such implicit implementation could be co-optimized higher level optimization, and saturate hardware resource.



[6] Kumar Chellapilla, Sidd Puri, and Patrice Simard. 2006. High performance convolutional neural networks for document processing. In Tenth international workshop on frontiers in handwriting recognition. Suvisoft.

Performance vs. TensorRT

- Hidet is able to fuse operator and tune for given input size.
- TensorRT may have dedicated optimization for Transformer?



Contents

中山大學
SUN YAT-SEN UNIVERSITY

Summary

- **Observation**: limited expressiveness of existing schedule method.
- **Proposed method**: task-mapping programming paradigm.
  - Non loop-oriented optimizations.
  - Reduce tuning time.
- **Implementation**: From scratch and integrate with PyTorch 2.0.
- **Results**: 1.4x speedup and 20x tuning time reduction.

Reference

- Paper: https://dl.acm.org/doi/10.1145/3575693.3575702
- Repository: https://github.com/hidet-org/hidet
- Talk on TVMCon'23: https://www.youtube.com/watch?v=-vMSGZS9-DA
- Part of official slide: https://centml.github.io/asplos23-tutorial/hidet_tutorial.pdf
- AutoTVM vs Ansor: https://tvm.apache.org/2021/03/03/intro-auto-scheduler