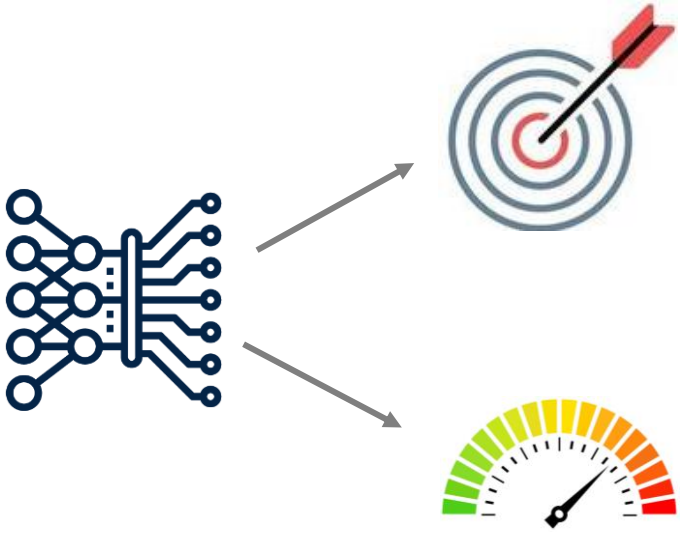
Bolt: Bridging the Gap between Auto-tuners and Hardware-native Performance

Jiarong Xing[12], Leyuan Wang[1], Shang Zhang[3], Jack Chen[3], Ang Chen[2], Yibo Zhu[1]

ByteDance[1], Rice University[2], NVIDIA[3]

Auto-tuners accelerate inference



Accuracy

Speed

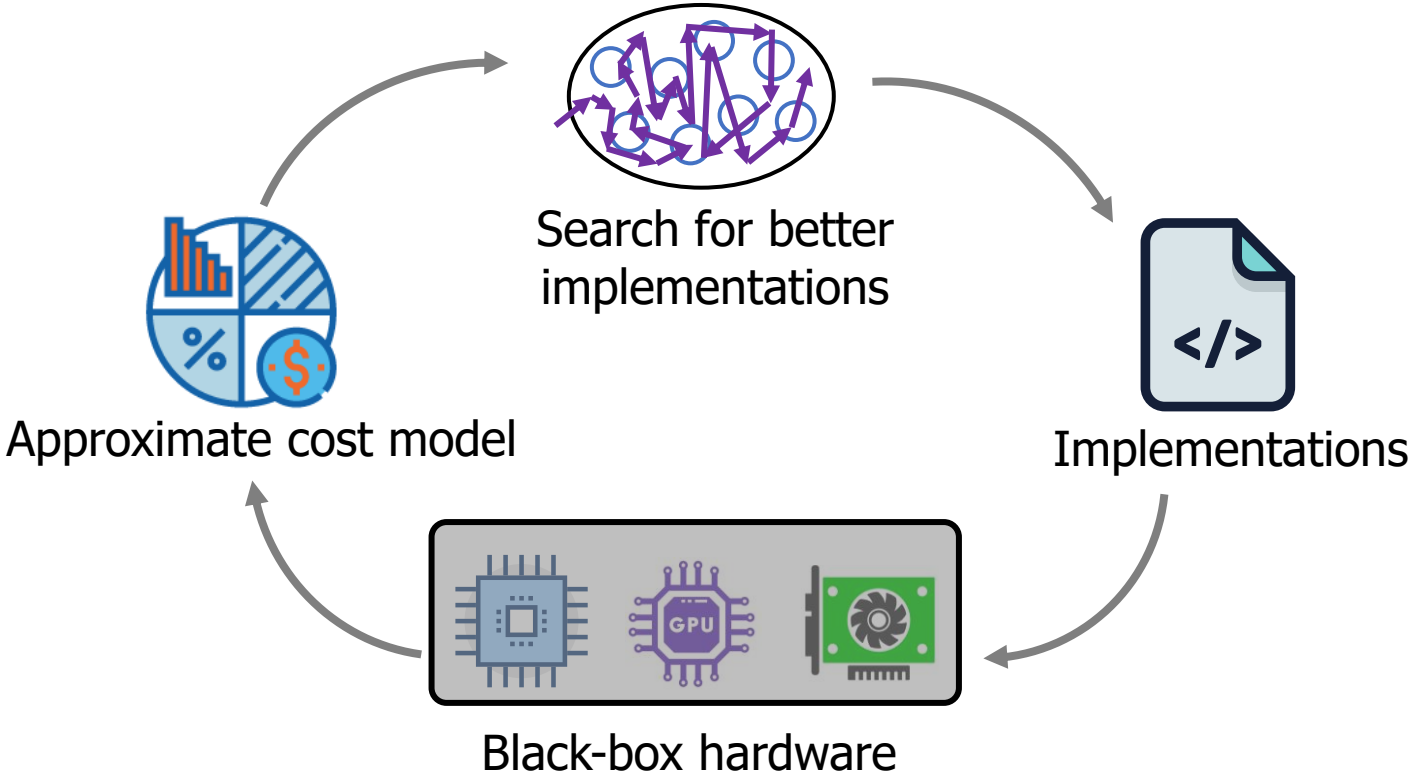
Ansor

tvm

Auto-tuners

- Neural networks are becoming more complex
  - Prediction accuracy has been improved
  - Makes it challenging for inference speed
- Auto-tuners are used to improve inference efficiency
  - E.g., AutoTVM, Ansor
  - End-to-end optimizations automatically for faster inference

Auto-tuner rationale

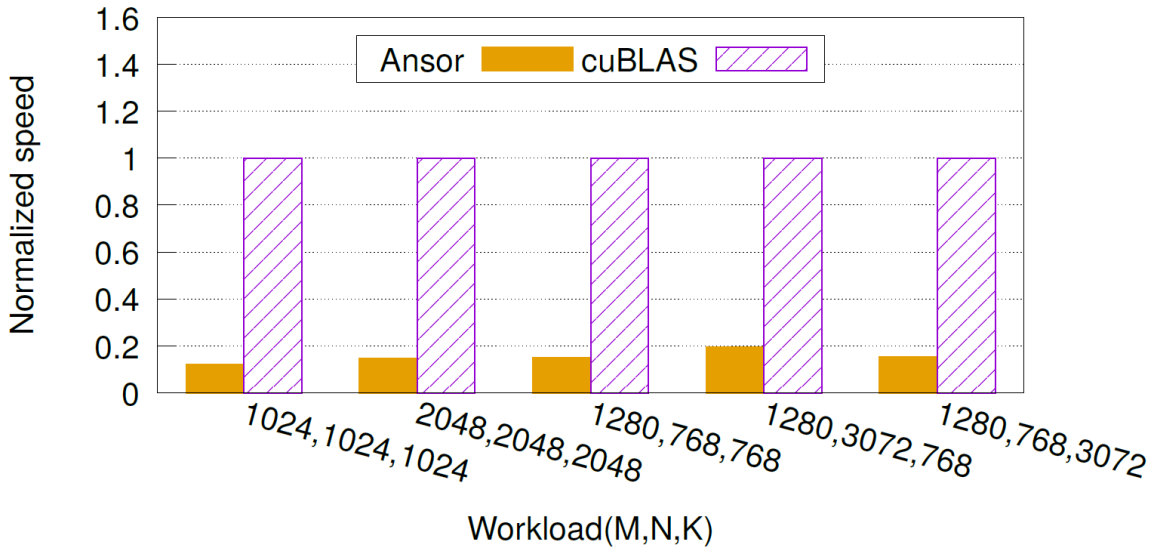


- Operate with opaque hardware models
- Workflow
  - The approximate cost model which will be improved gradually
  - Search for better implementations using the current model
  - Generate implementations and measure the actual performance
  - Improve the cost model and iterate again

Inefficiency of existing auto-tuners

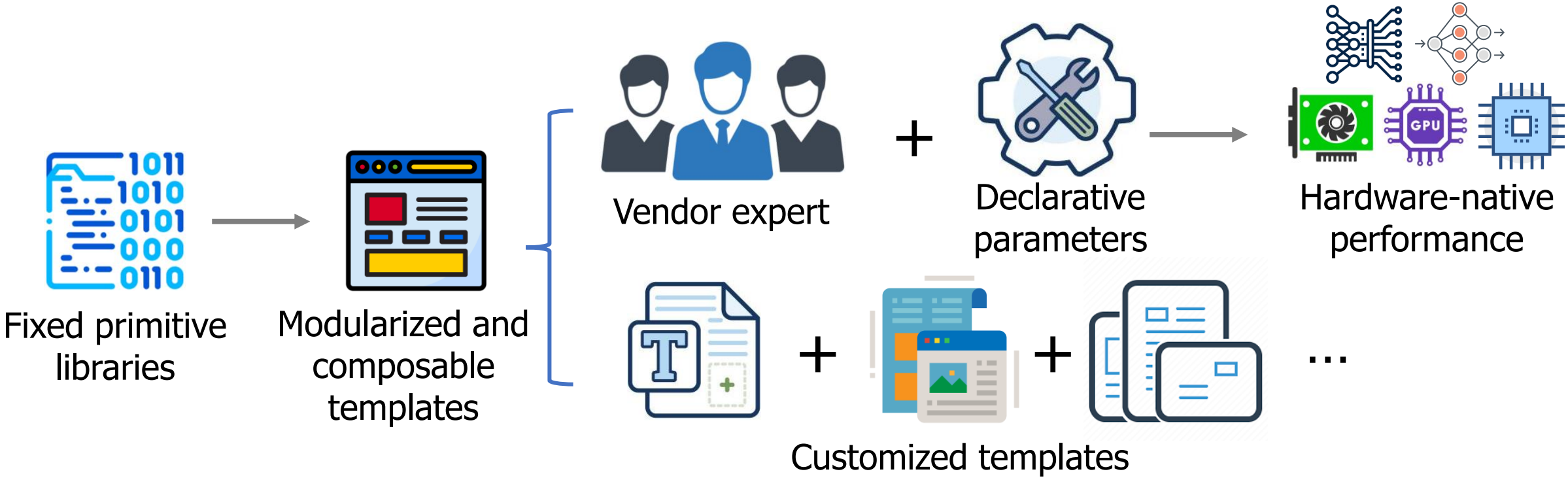
Neural networks	Tuning time (hours)
ResNet-18	11.8
ResNet-50	15.8
RepVGG-A0	4.4
RepVGG-B0	4.6
VGG-16	19.1
VGG-19	18.8

Tuning time on a single NVIDIA T4 GPU



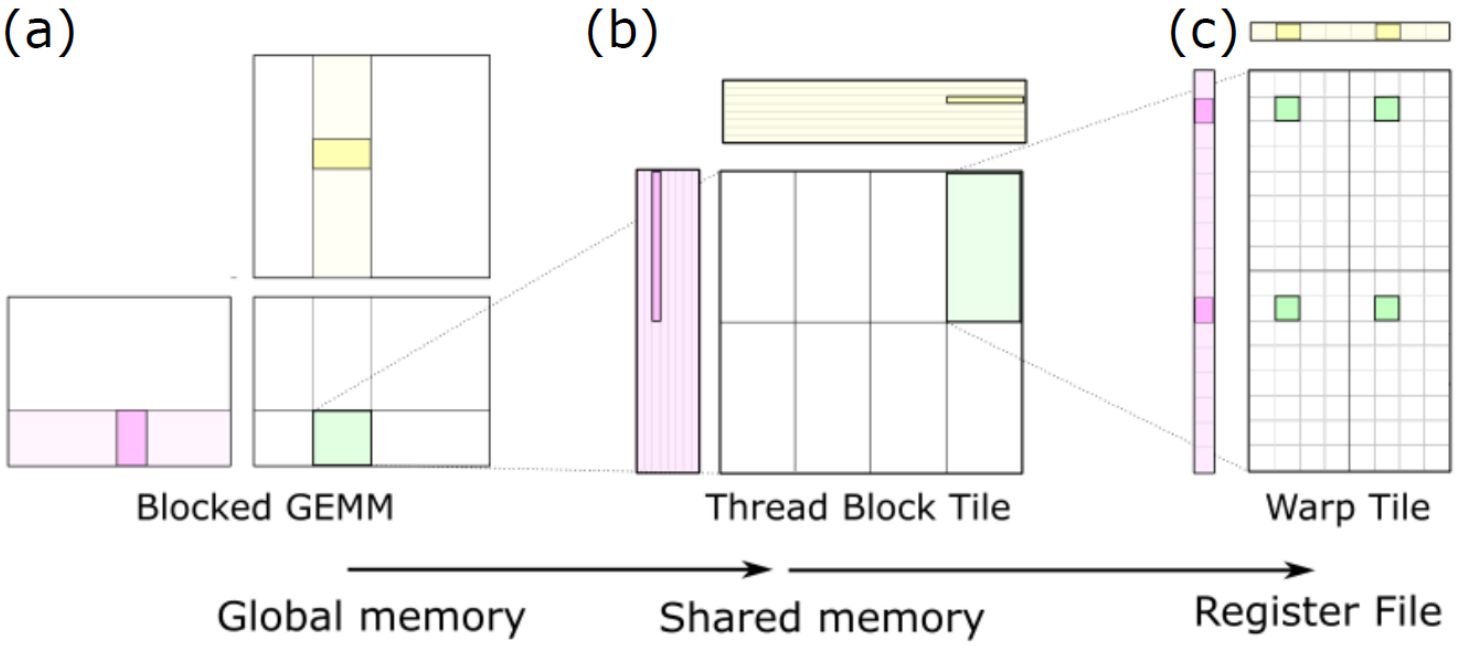
- Hardware-agnostic approaches have inefficiencies:
  - Long tuning time
    - Take hours to days to tune a model
    - Challenging for rapid model iterations
  - Ineffective usage of hardware resources (e.g., tensor core)
    - Low performance for compute-intensive workloads, e.g., FP16 GEMMs, Conv2Ds.
    - Hard to achieve hardware-native performance, e.g., cuBLAS, cuDNN.

An emerging trend: Templated libraries



- Vendor libraries are becoming modularized and composable
- Make it possible for flexible high-performance auto tuning
  - Easily parameterized for different tensor shapes
  - Reusable primitives with hardware-native performance
  - Amenable to a tighter integration with auto-tuning
  - Extendible to new primitives

Example: CUTLASS on NVIDIA GPUs



- CUDA C++ templates for high-performance GEMM and related computations
- Key techniques: Hierarchical decomposition + careful data movement
- Support various GPU architectures and data types.
  - Volta, Turing, and Ampere, etc.
  - B1, INT4, INT8, FP16, BF16, FP32, TF32, FP64, complex, and quaternion

CUTLASS: https://github.com/NVIDIA/cutlass

A Conv2D kernel template in CUTLASS

```
/// Define an Implicit GEMM convolution forward propagation (fprop) kernel
using Conv2dFpropKernel = typename cutlass::conv::kernel::DefaultConv2dFprop<
  ElementInputA,                                  // data type of element a (mapped to activation for fprop)
  LayoutInputA,                                   // layout of element a (mapped to activation for fprop)
  ElementInputB,                                  // data type of element b (mapped to filters for fprop)
  LayoutInputB,                                   // layout of element b (mapped to filters for fprop)
  ElementC,                                       // data type of element c (mapped to output for fprop)
  LayoutC,                                        // layout of element c (mapped to output for fprop)
  ElementAccumulator,                             // data type of internal accumulation
  MMAOp,                                          // opcode class tag
  SmArch,                                         // target SM architecture
  ThreadblockShape,                               // shape of threadblock tile
  WarpShape,                                      // shape of warp-level GEMM tile
  InstructionShape,                               // shape of target math instruction
  EpilogueOp,                                     // epilogue operator
  SwizzleThreadBlock,                             // optional function to reorder threadblocks for locality
  NumStages,                                      // number of pipeline stages in threadblock-scoped GEMM
  cutlass::arch::OpMultiplyAddSaturate,           // math operation on data of element a and b
  cutlass::conv::IteratorAlgorithm::kAnalytic // globabl memory iterator algorithm
>::Kernel
```

- Composable template with manifold tunable parameters
- No end-to-end auto-tuning support
  - Templates must be tuned and invoked manually by developers
  - Repeat for each operator, workload, and hardware

A gap between auto-tuners and templated libraries

Auto-tuners

- + e2e auto optimization
- - inefficient for certain workloads

Templated libraries

- + good performance and extensibility
- - no e2e auto support

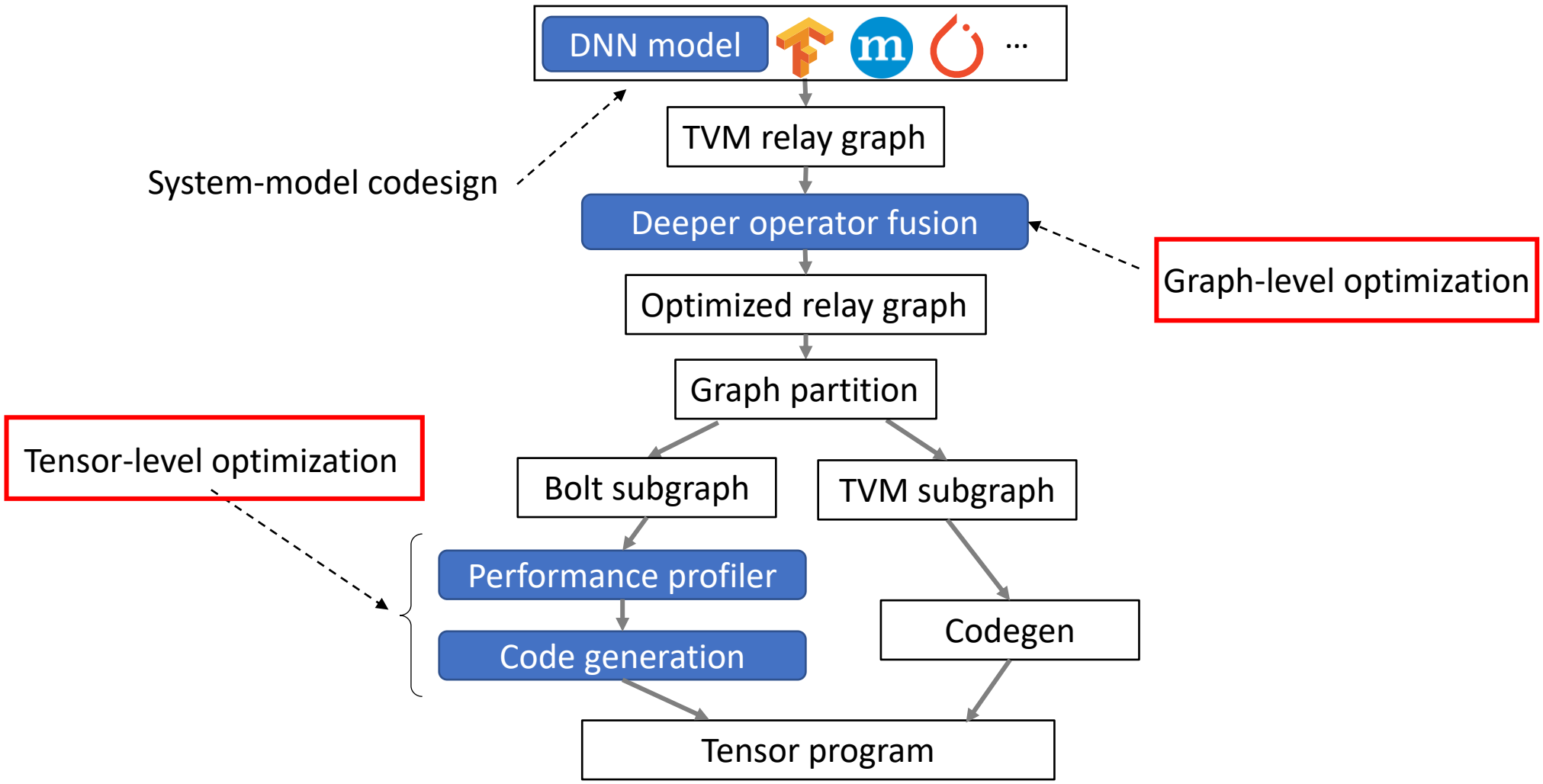
Bridging the gap with Bolt

Auto-tuners

Templated libraries

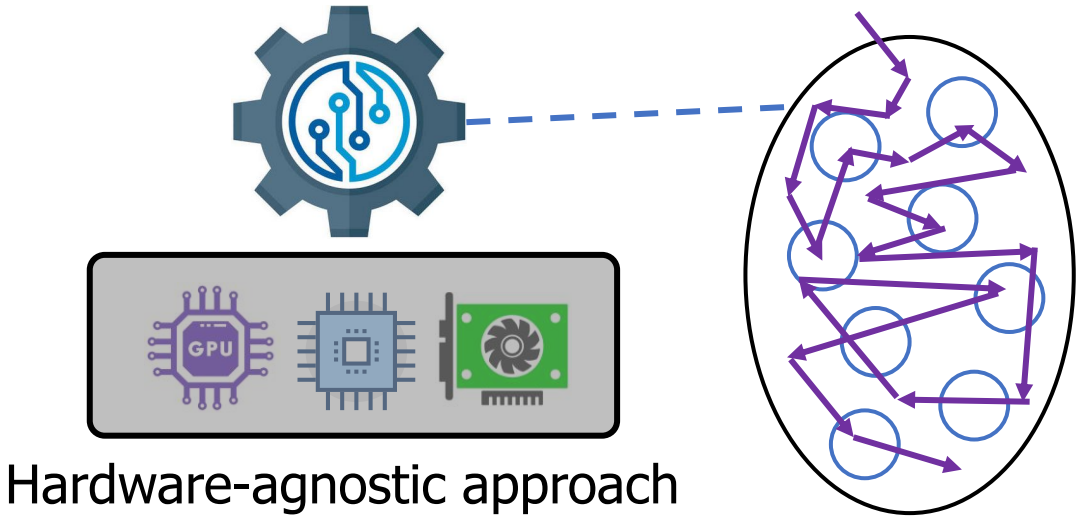
- The best of both worlds
  - Automatic end-to-end model optimization
  - Hardware-native performance
- Key features
  - Faster auto-tunning + tensor code generation
  - Deeper operator fusion
  - System-model co-design insights

The workflow of Bolt

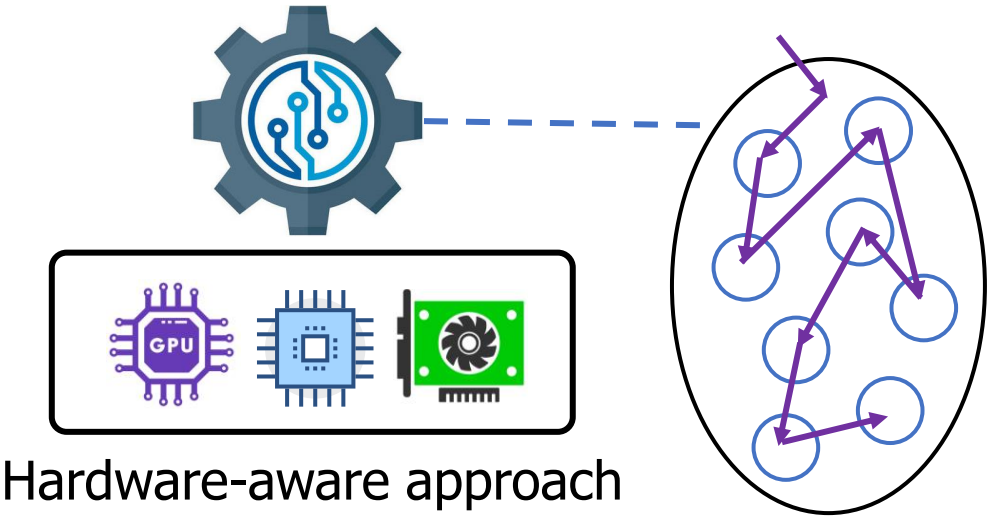


- Build an auto-tuner from scratch takes a lot of efforts
- Adopt the TVM BYOC approach (Bring Your Own Codegen)

Efficient auto-tuning



Hardware-agnostic approach



Hardware-aware approach

- Problem: Current auto-tuners take long to tune a model
  - Hardware-agnostic approach, large search space
- Solution: Hardware-aware approach (reduce tuning time to minutes)
  - Search over template parameters
  - Utilize hardware knowledge to reduce search space
    - Within the capacity of RF, large warp sizes can achieve higher compute-memory ratio
    - Small threadblock sizes for small problem sizes to keep more SMs busy
    - …

Templated code generation

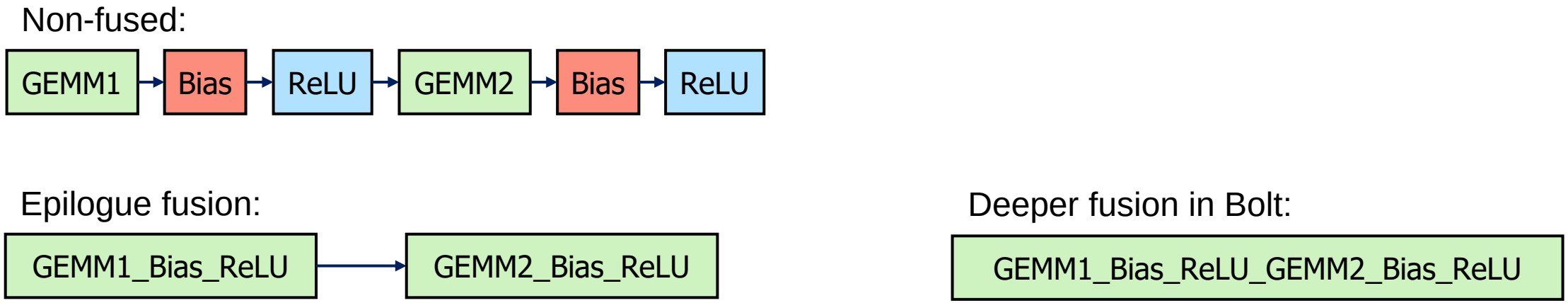
Generate CUDA code

Layout transformation

Padding

- Bolt generates CUDA code in the CUTLASS convention directly
  - Can reach 300 TFLOPS throughput for FP16 GEMM on A100
  - More than 95% of the hardware theoretic limit
- Optimizations:
  - Layout transformation (NCHW to NHWC)
  - Kernel padding: 8, 4, 2, 1 alignment

Deeper operator fusion

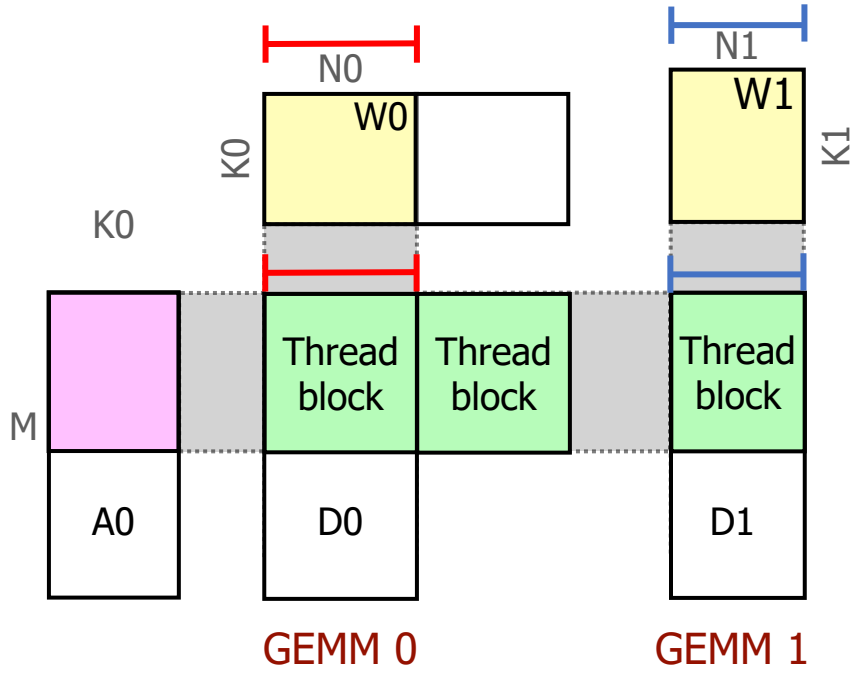


- Operator fusion: Merging back-to-back operators to improve performance
- Existing methods cannot fuse back-to-back GEMMs/Convs
- Deeper operator fusion in Bolt:
  - Fusing two or more back-to-back GEMMs/Convs
  - Achieved by extending hardware-native templates

Key ideas of deeper fusion

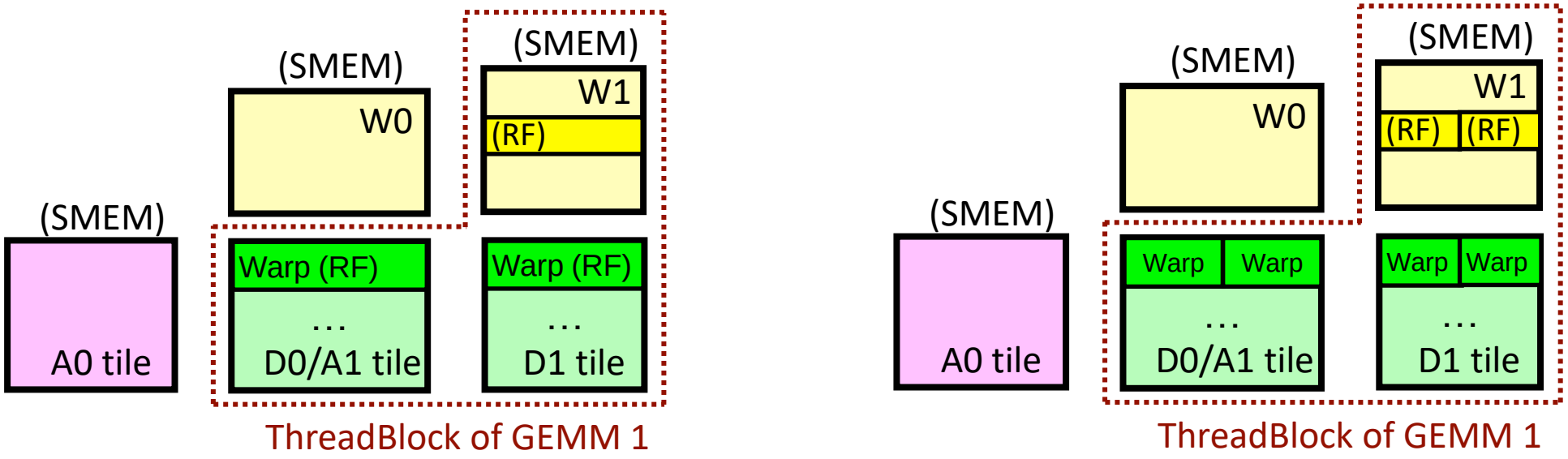
$$D0 = \alpha_0 A0 \cdot W0 + \beta_0 C0, \quad (1)$$
$$D1 = \alpha_1 D0 \cdot W1 + \beta_1 C1, \quad (2)$$

Definition of two GEMM fusion



- Key idea:
  - Compute the second GEMM without loading its input from the global memory
  - Output threadblock of the 1st GEMM in the same threadblock as input for the 2nd GEMM
- Threadblock residence
  - GEMM: ThreadBlock_N = GEMM_N
  - Conv2D: ThreadBlock_N = # output channels

Two threadblock residence implementations



Register file-based fusion

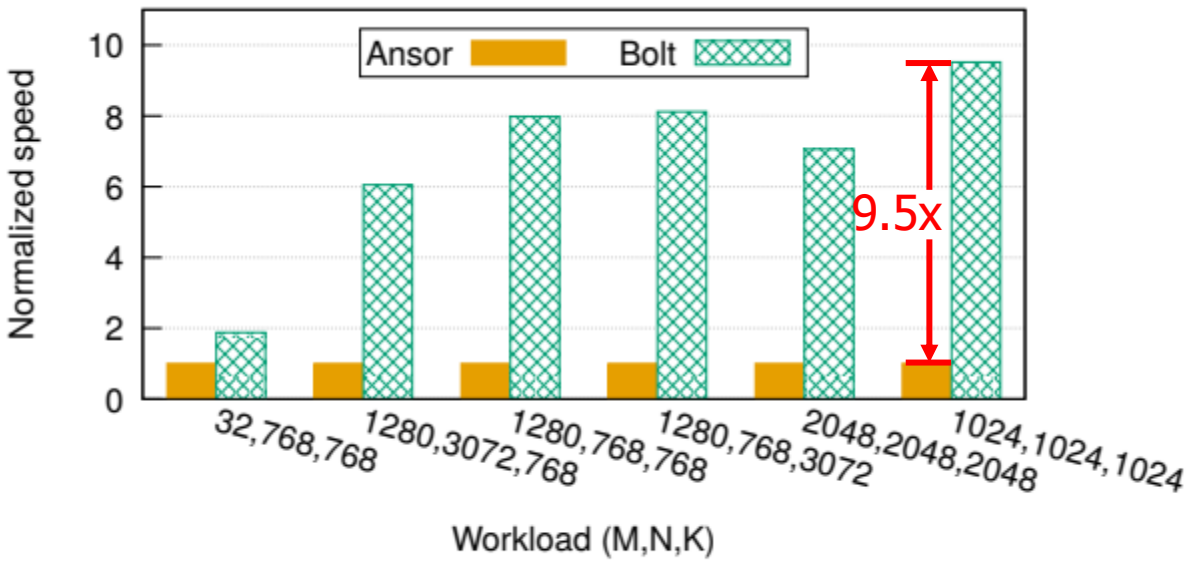
Shared memory-based fusion

- RF-based fusion:
  - ThreadBlock_N = GEMM_N = Warp_N
  - High RF pressure and constraints on kernel choices
- Shared memory-based fusion:
  - Relax the constraint of ThreadBlock_N = Warp_N
- Bolt hides the low-level implementations from users

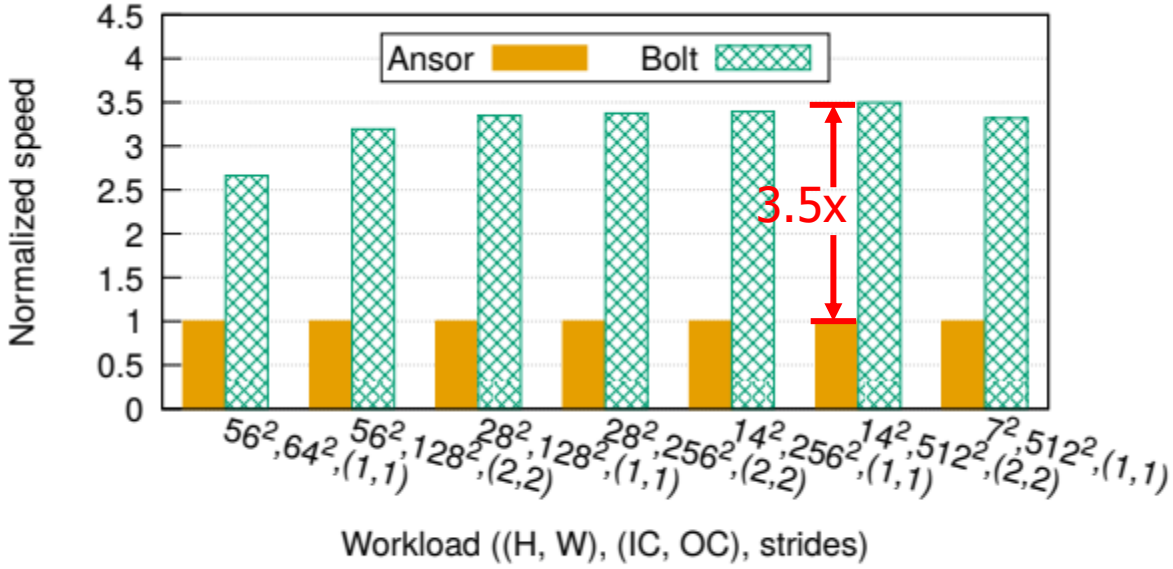
Evaluation

- Setup:
  - NVIDIA T4 GPU + TVM-BYOC+CUTLASS
  - FP16 workloads
  - Baseline: Ansor-OSDI’20
- Experiments:
  - GEMM and Conv2D speed
  - Deeper operator fusion performance
  - End-to-end model performance

GEMM and Conv2D speed



(a) GEMMs performance.



(b) Conv2D performance.

- GEMM workload: Square GEMMs + Bert.
- Conv2D workload: ResNet-50. Kernel size=3*3, bs=32, (1,1) zero padding.
- Speedup: 6.1-9.5x(1.9x) for GEMMs, 2.7-3.5x for Conv2Ds

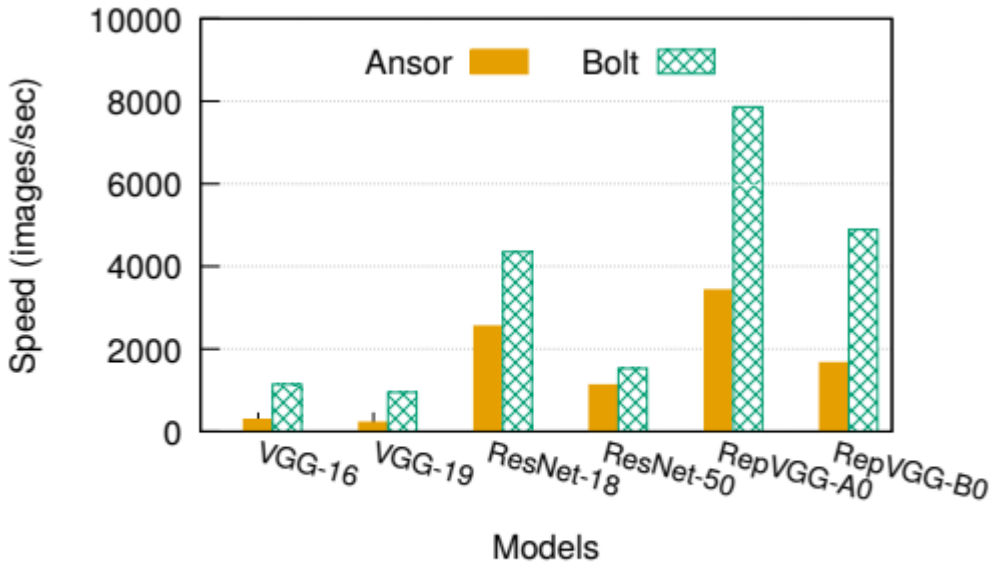
Deeper operator fusion performance

1st GEMM			2nd GEMM			Normalized speed	
M	N	K	M	N	K	w/o fuse.	w/ fuse.
2464	1	4	2464	4	1	1.00	1.24
16384	64	256	16384	16	64	1.00	1.34
32768	128	576	32768	64	128	1.00	1.28
128320	32	96	128320	96	32	1.00	1.46

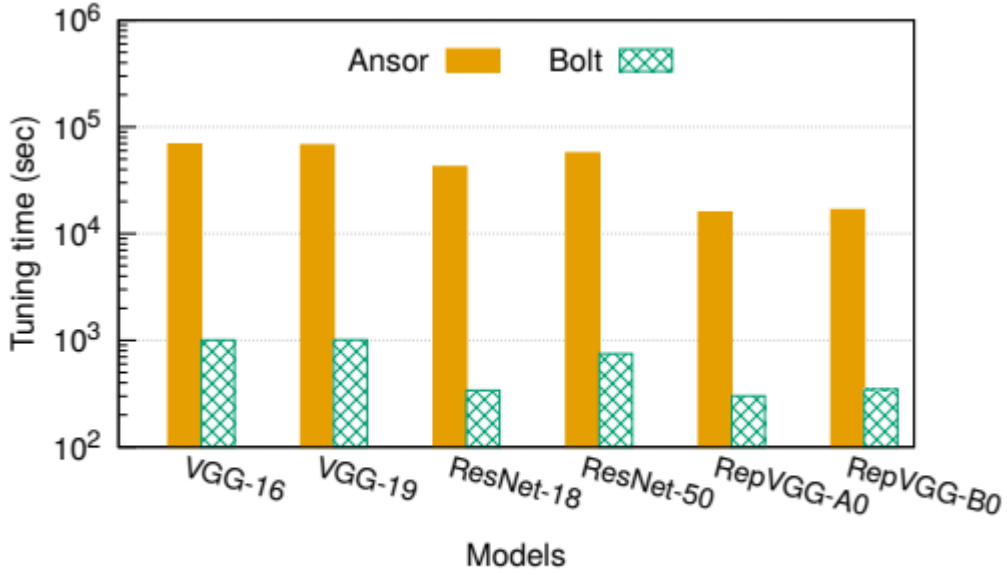
3×3 Conv2D			1×1 Conv2D		Normalized speed	
H, W	IC, OC	strides	H, W	IC, OC	w/o fuse.	w/ fuse.
224^2	3, 48	(2, 2)	112^2	48, 48	1.00	1.10
112^2	48, 48	(2, 2)	56^2	48, 48	1.00	1.41
56^2	48, 48	(1, 1)	56^2	48, 48	1.00	1.87
224^2	3, 64	(2, 2)	112^2	64, 64	1.00	1.24
112^2	64, 64	(2, 2)	56^2	64, 64	1.00	1.12
56^2	64, 64	(1, 1)	56^2	64, 64	1.00	2.02

- Baseline: Bolt without deeper operator fusion.
- GEMM1 + ReLU +GEMM2 + ReLU
  - Recommendation models: DCNv2, DLRM
- Conv2D1+BiasAdd+ReLU+Conv2D2+BiasAdd+ReLU
  - RepVGG-A0 and A1
- Speedup: 1.2x-1.5x for GEMM, 1.1x-2.0x for Conv2D

End-to-end model optimization



(a) Inference speed.



(b) Tunning time.

- Bolt is 4.2x faster on VGG models, 1.5x faster on ResNet models, and 2.6x faster on RepVGG models.
- Bolt can finish the tuning within 20 minutes for all models while Ansor takes 12 hours on average.

Summary

- A gap between auto-tuners and templated libraries
  - Automatic end-to-end optimization vs. hardware-native performance
- Bolt: Bridging the gap between them
  - Efficient auto-tunning and code generation
  - Deeper operator fusion
  - System model co-design insights
- Evaluation:
  - Bolt can improve the inference speed and reduce tuning time significantly.
- Code: https://github.com/apache/tvm/pull/9261

Thanks!