Bolt

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

MLSys'22

Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, Yibo Zhu, ByteDance, Rice University, NVIDIA

Introduction

About model compilation

- Frameworks
  - PyTorch, TensorFlow…
- Complier frontend
  - High-level IR / Graph IR
  - Devices independent
- **Complier backend**
  - Low-level IR / Operator IR
  - **Devices specific**
  - **Use kernel libraries**

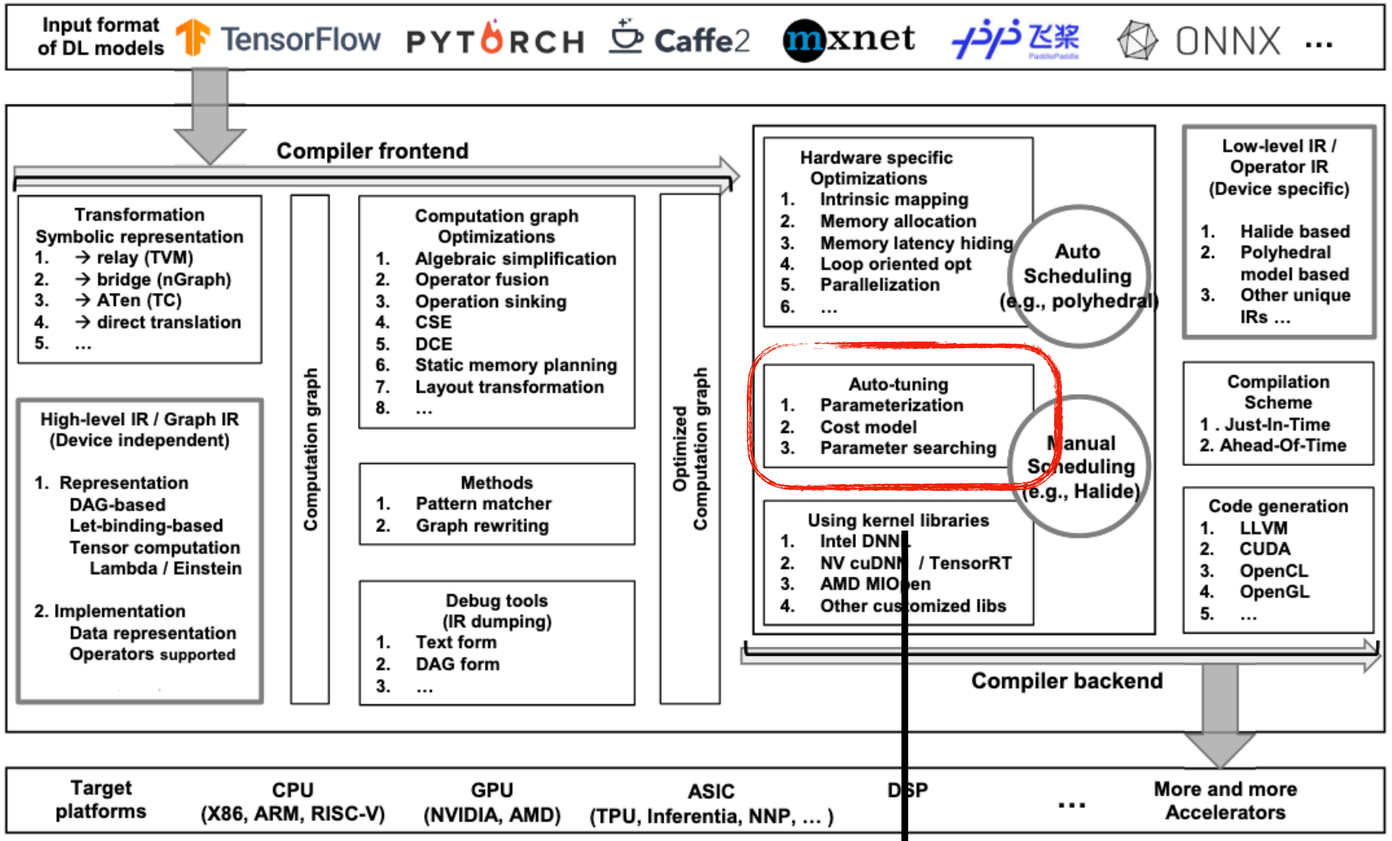


Fig. 2. The overview of commonly adopted design architecture of DL compilers.

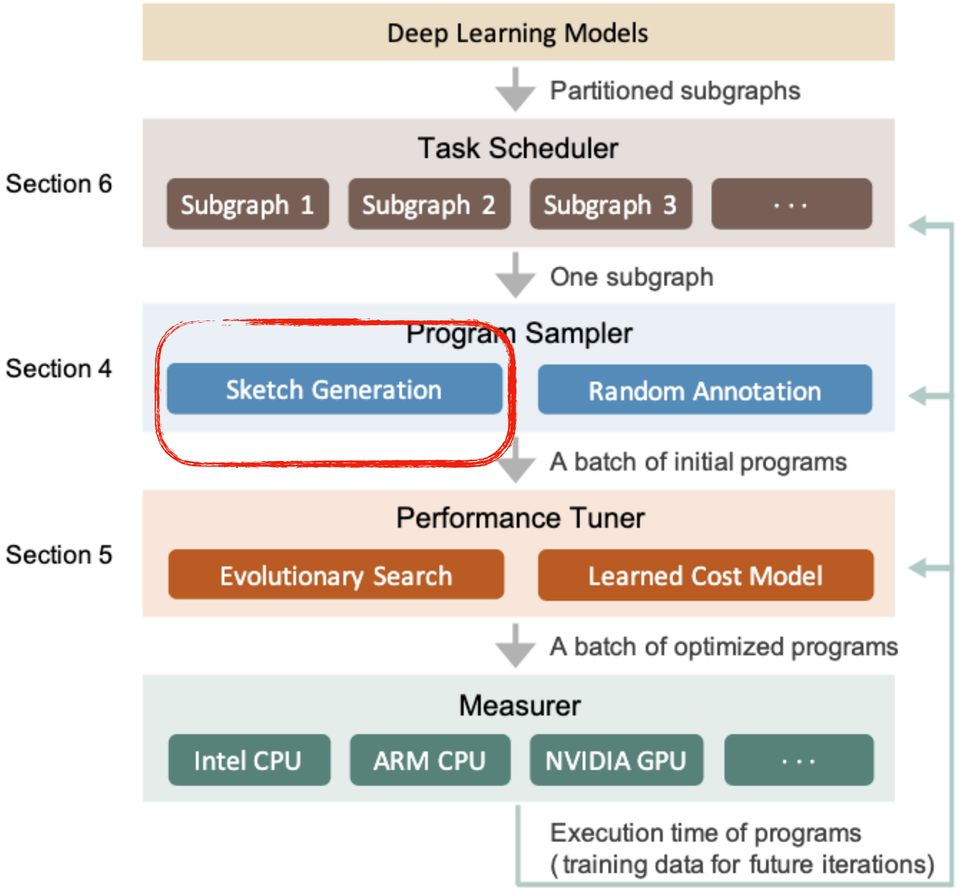
Auto-tuning: Time-consuming

The Deep Learning Compiler: A Comprehensive Survey

Introduction

Auto-tuner — example Ansor

- Ansor — Auto-tuner developed base on TVM
  - Program sampler (**billions of combinations**)
  - Performance tuner (evolutionary search / **learned cost model**)



Ansor: Generating High-Performance Tensor Programs for Deep Learning OSDI'22

Introduction

Auto-tuners have a performance gap

#1 Traditional vendor library

- A **fixed** set of primitives
  - **hand-optimized** for the underlying **hardware**
  - **black box**, can't observe the logic from outside

#2 Auto-tuners

- Generate 'efficient' tensor programs by navigating large search space
  - **Use cost models based on opaque device to ensure generality of auto-tuner**

Can't achieve native hardware performance ←

Background & Motivation

Auto-tuners have a performance gap

- **Lack of hardware-native performance**
  - auto-tuners — use opaque device models
- **Inefficient program search**
  - SAMPL — only for static models

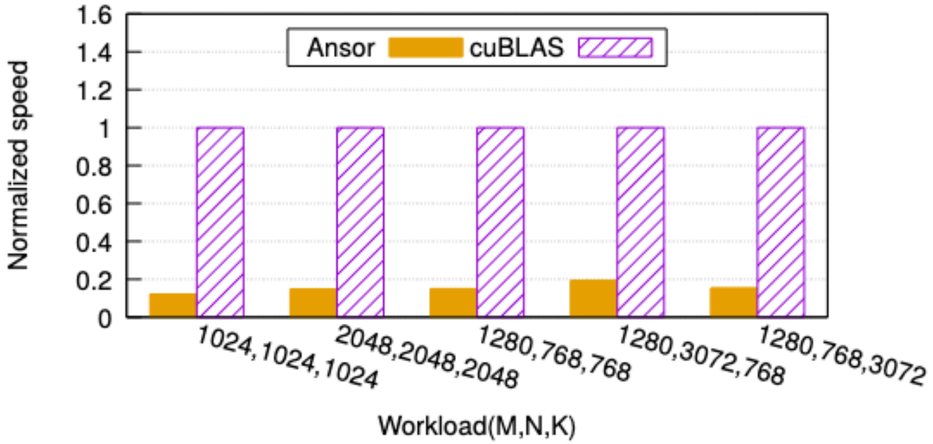


Figure 1. The speed of Ansor, implemented in TVM auto-scheduler, under-performs significantly compared to the device speeds achievable in cuBLAS. Workloads: two large square GEMMs and three GEMMs in BERT (Devlin et al., 2018), a widely adopted NLP model, when the batch size is 32 and the sequence length is 40.

less than 20% of the library performance

Background & Motivation

Emerging trend: Templated libraries

- **CUTLASS** (CUDA Templates for Linear Algebra Subroutines) / Nvidia
- OneDNN / Intel
- ROCm / AMD

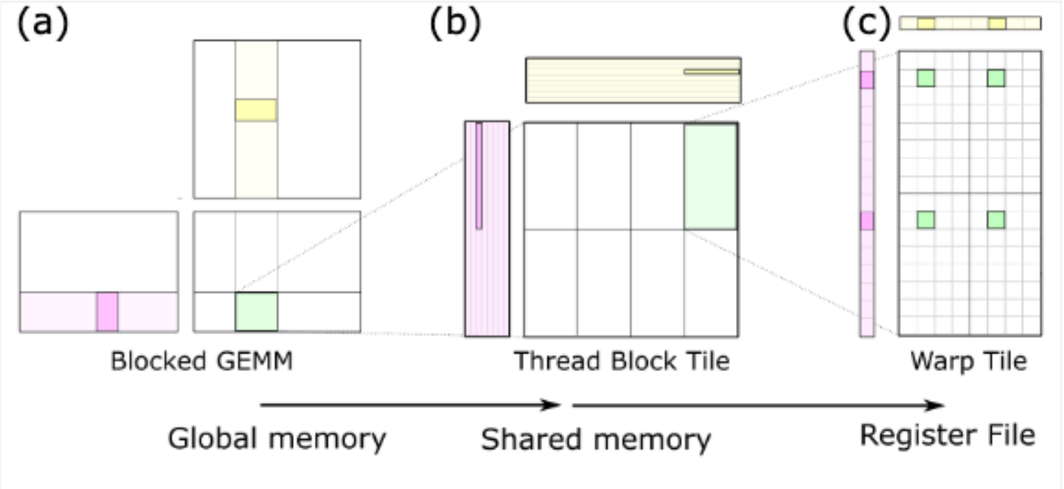


Figure 2. The GEMM hierarchy in CUTLASS and the data movement in threadblock and warp tiles.

Templated libraries

Efficient design patterns that take into account device details

Background & Motivation

Bolt: The best of both worlds

- **Three levels of optimizations**
  - **Group level: Enabling deeper operator fusion**
    - persistent kernel fusion (multi-Conv/GEMM)
  - **Operator level: Automating templated code generation**
    - Light-weight performance profiler
    - Templated code generation
  - **Model level: System-friendly models**
    - summarized three system-model codesign principles

Overview of Bolt's design

Overview of Bolt

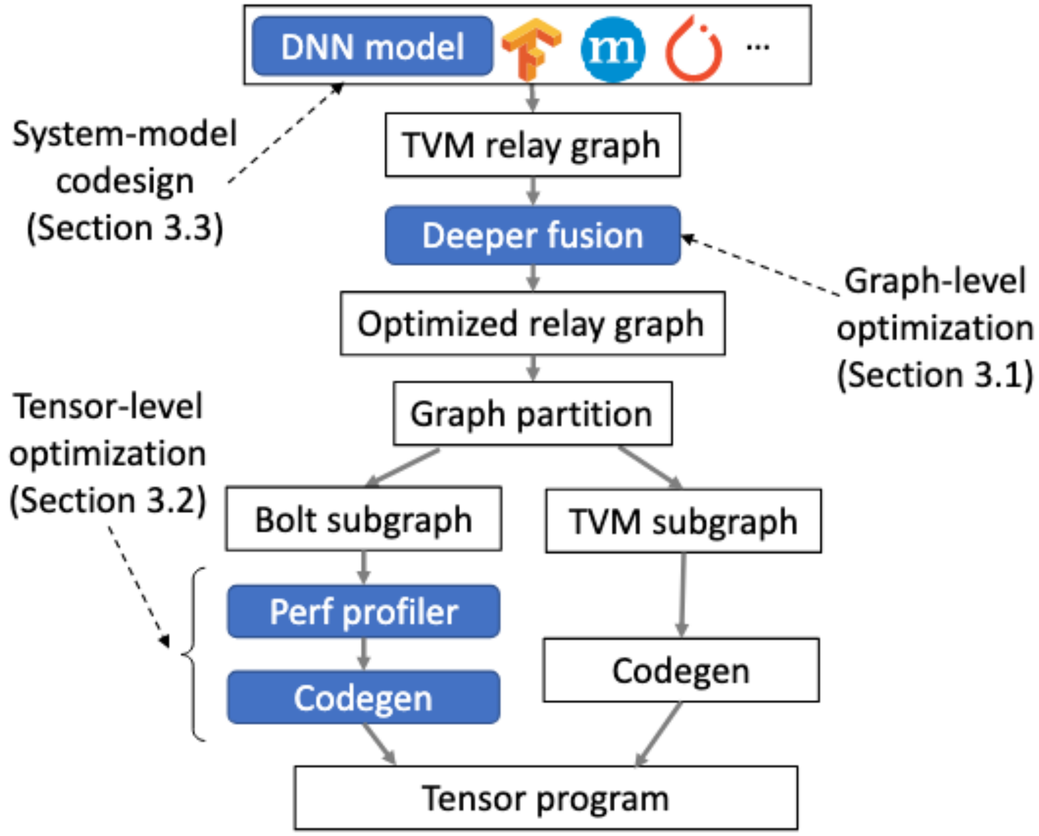


Figure 3. The workflow of Bolt. Blue boxes are our contributions.

Optimization I — Enabling deeper operator fusion

Bolt

#1 Prerequisite: Epilogue fusion (CUTLASS)

- **Fuse two sequential GEMMs/Convs into a single operator**
  - **eliminating memory traffic** for storing and loading **inter-layer activations**
  - **eliminating launch latency** which is especially beneficial for **short kernels with small batch sizes**
  - **enlarging optimization scope for the compiler** to explore more instruction scheduling options

Optimization I — Enabling deeper operator fusion

Bolt

#1 Prerequisite: Epilogue fusion (CUTLASS)

- **Bolt takes epilogue fusion as a starting point, and develops deeper fusions using persistent kernels.**

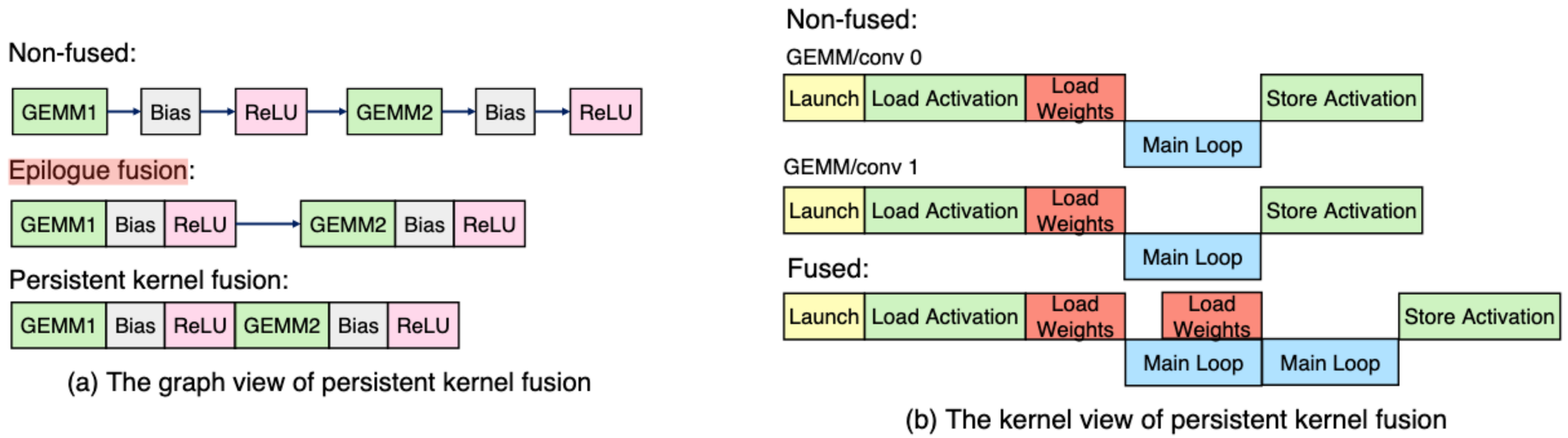


Figure 4. The graph view and kernel view of persistent kernel fusion for back-to-back GEMMs/Convs.

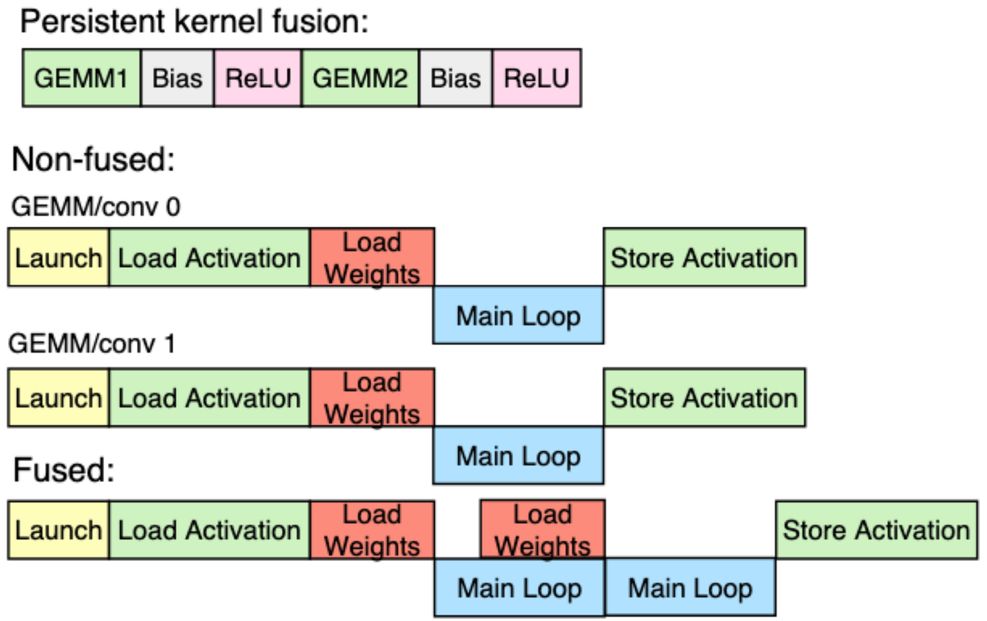
Optimization I — Enabling deeper operator fusion

Bolt

#2 Persistent kernel (GEMM/Conv) fusion

- **Persistent kernels** allow fusing **multiple GEMMs or Convs into one operator** to improve performance
- Example: back-to-back GEMM

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



(b) The kernel view of persistent kernel fusion

Figure 4. The graph view and kernel view of persistent kernel fusion for back-to-back GEMMs/Convs.

Optimization I — Enabling deeper operator fusion

Bolt

#2 Persistent kernel (GEMM/Conv) fusion

- In order to fuse backto-back GEMMs, **output activation D0 of the first GEMM** layer **must be used as input activation of the second GEMM layer.**

Back-to-back GEMM: $M_1 = M_2 = \ldots = M_n$

- **Back-to-back Conv:** all **subsequent Convs (from the 2nd)** must use **1 × 1 filter** with **no padding** and a **stride of one**.

$$D0 = \alpha_0 A0 \cdot W0 + \beta_0 C0, \quad (1)$$

$$D1 = \alpha_1 D0 \cdot W1 + \beta_1 C1, \quad (2)$$

Optimization I — Enabling deeper operator fusion

Bolt

#3 Key property: Threadblock residence

- **The key challenge of persistent kernels** is to **compute the 2nd GEMM/Conv without loading its input activation from the global memory**

- **Each output threadblock of the 1st GEMM/Conv to remain within the same threadblock memory (either in the shared memory or register files) as its respective input threadblock**

Back-to-back GEMM: ThreadBlock_N = GEMM_N

Back-to-back Conv: ThreadBlock_N = Conv_output_channel

Optimization I — Enabling deeper operator fusion

Bolt

#3 Key property: Threadblock residence

- **Each output threadblock of the 1st GEMM/Conv to remain within the same threadblock memory (either in the shared memory or register files) as its respective input threadblock**

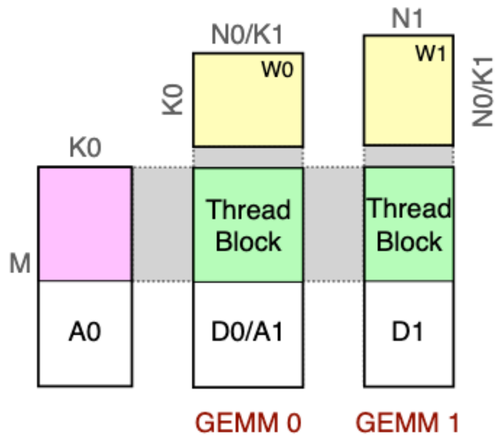


Figure 5. Illustration of threadblock-residence of GEMM fusion. Colored boxes represent one single threadblock. This requires ThreadBlock0_N = N0, ThreadBlock1_N = N1.

Back-to-back GEMM: ThreadBlock_N = GEMM_N

Optimization I — Enabling deeper operator fusion

Bolt

#4 Fusion: RF-resident fusion

- When **the weight matrix W1 can be completely streamed into a warp tile in its 'N ' dimension**, **threadblock-residence can be satisfied** by **storing the output activation for each threadblock entirely in the register file (RF)**

CUTLASS warp fragment iterator

Warp_N = ThreadBlock_N = GEMM_N

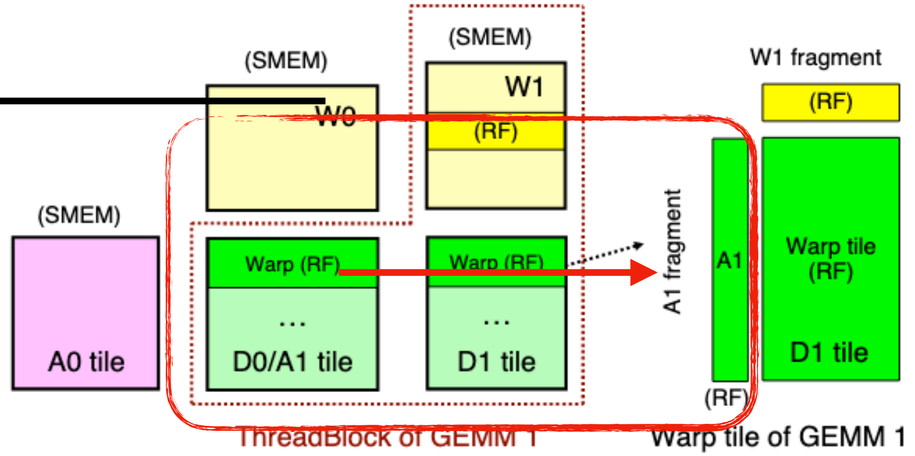


Figure 6. RF-resident fusion in a threadblock of back-to-back GEMMs. The threadblock and warp size requirements are: Warp0_N=ThreadBlock0_N=N0, Warp1_N=ThreadBlock1_N=N1.

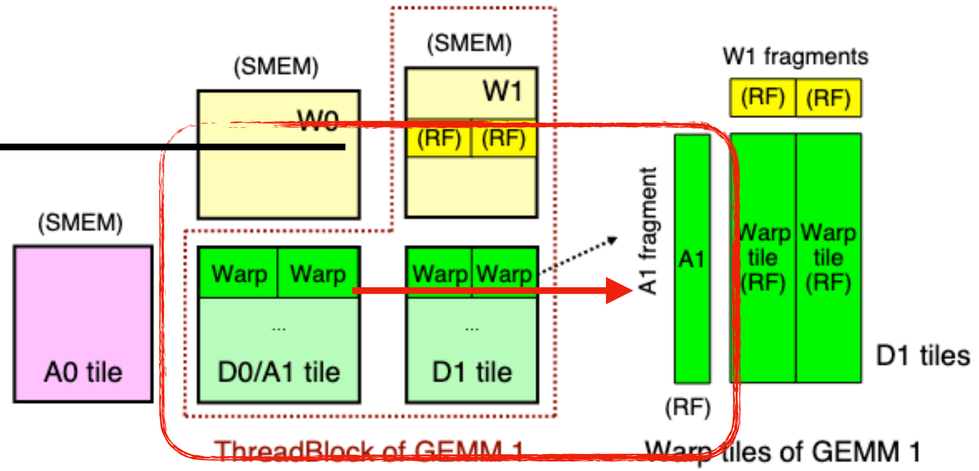
Optimization I — Enabling deeper operator fusion

Bolt

#4 Fusion: Shared memory-resident fusion

- **RF-resident GEMM fusion** **creates higher RF pressure** especially **when GEMM N is large**, which will potentially **harm the kernel performance** and limit the applicable scenarios

CUTLASS smem fragment iterator



Shared memory-resident fusion

relax the warp size restriction

Figure 7. Shared memory-resident fusion in a threadblock of back-to-back GEMMs. The threadblock size requirements are: ThreadBlock0_N = N0 ≠ Warp0_N, ThreadBlock1_N = N1 ≠ Warp1_N.

Optimization 2 —Automating templated code generation

Bolt

#1 Challenges in code generation

- **Templated library usually do not provide complete functionality for end-to-end models, but only support a subset of operators**
  - Develop a full compiler stack from scratch for each hardware—not scale
  - **BYOC**
- **Templated device libraries by themselves are not directly runnable.**
  - **a light-weight hardware-native performance profiler**
- **Conventional BYOC regards device libraries as agnostic external functions and generates hardware code with hooks to invoke them at runtime**

Optimization 2 —Automating templated code generation

Bolt

#2 Light-weight performance profiler

- **Traditional auto-tuners' cost model without hardware info**
  - Large search space & long tuning time
  - **Bolt: Sampling(User hardware details to reduce space) & profiling**
- **Bolt determines params according to the GPU architecture as well as tuning guidelines that are specific to each hardware, thanks to the whitebox approach.**
  - Large warp size within capacity of RF
  - 4/8 warps per threadblock for NVIDIA GPUs
  - Small problem need small threadblock size to launch enough block

Optimization 2 —Automating templated code generation

Bolt

#3 Templated code generation

- Bolt produces low-level tensor implementations in the CUTLASS convention **by instantiating the templates with the best parameters identified by the profiler**

- **Advantages**
  - the generated code **delivers superior performance**
  - it provides full **flexibility** to add novel optimizations in the generated code.

Optimization 2 —Automating templated code generation

Bolt

#3 Templated code generation-Layout transformation

- Bolt **implements the transformation in the generated CUDA code** of the model's first and last layer directly to **save extra kernel launch overhead**

NCHW (PyTorch) ->
NHWC (CUTLASS)

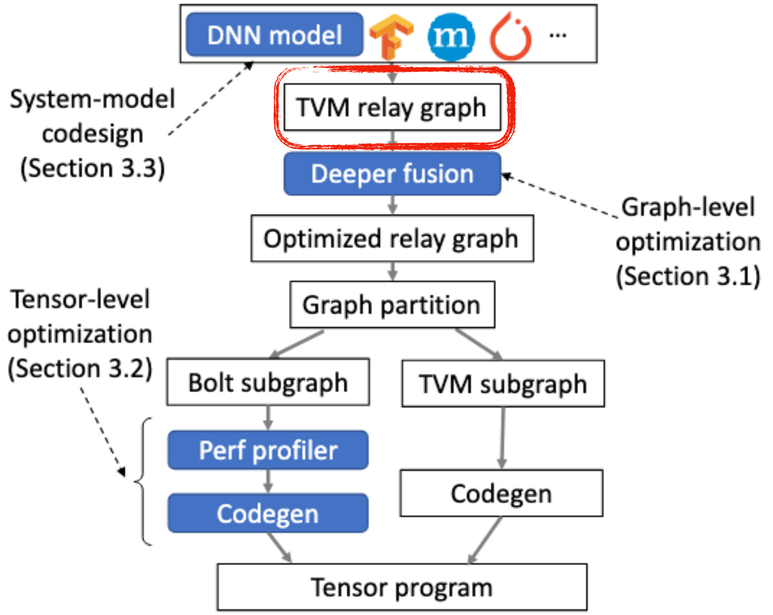


Figure 3. The workflow of Bolt. Blue boxes are our contributions.

Optimization 2 —Automating templated code generation

Bolt

#3 Templated code generation-Kernel padding

- CUTLASS supports **alignments 8, 4, 2, 1** to cover all different workloads, the performance varies significantly across different alignments.
  - The largest vectorized load and store supported by NVIDIA GPUs is **128 bits**, so the most efficient way to use it for **FP16** data type is alignment **8 (128/16)**.
  - Tensor shapes with a dimension that cannot be divided by **8** will have to use **smaller alignments**

Fully utilize tensor core acceleration
reduce memory loading time

Optimization 3 — Designing system-friendly models

Bolt

System-model codesign

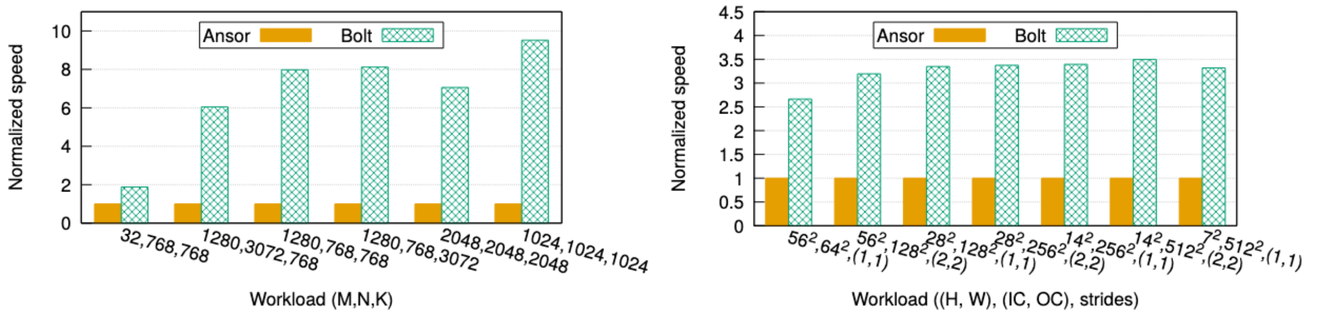
- **Three principles**
  - **Exploring different activation functions with epilogue fusion**
  - **Deepening models with 1×1 Convs (persistent kernel fusion)**
  - **Aligning tensor shapes to use GPUs more efficiently (kernel padding)**

Evaluation

Setup & Microbenchmarks 1,2

- **Ansor** — the **stateof-the-art auto-tuner** in TVM as our **baseline**

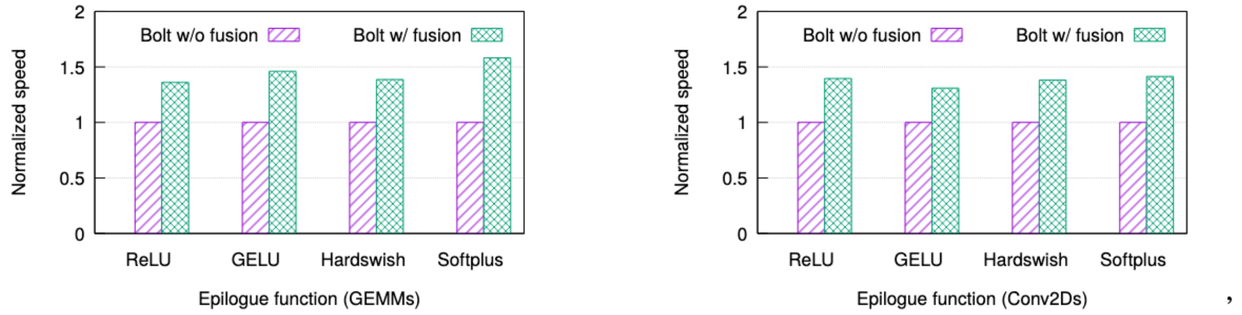
GEMM/Conv2D performance



(a) GEMMs performance. (b) Conv2D performance.

Figure 8. The performance of Bolt on GEMMs and Conv2Ds. Figure 8a shows the speed of GEMMs in BERT with batch size=32 and sequence length=40 and two square GEMMs. Figure 8b shows the speed of 3 × 3 Conv2Ds in ResNet-50. The batch size=32 and all Conv2Ds use (1, 1) zero padding.

Epilogue fusion performance



(a) GEMM epilogue fusion. (b) Conv2D epilogue fusion.

Figure 9. The performance of epilogue fusion on pattern GEMM/Conv2D+BiasAdd+Activation. The workload of the GEMM is M=1280, N=3072, and N=768. The workload of the Conv2d is H=W=56, IC=OC=64, kernel=(3, 3), stride=(1,1), and padding=(1,1).

Evaluation

Microbenchmarks 3 Persistent kernel fusion performance

Table 1. The performance of fusing two back-to-back GEMMs using persistent kernels. Each GEMMs is followed by a ReLU epilogue and all of them will be fused into one kernel.

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

GEMM performance

Table 2. The performance of fusing two back-to-back Conv2Ds using persistent kernels. Each Conv2D is followed by a BiasAdd and a ReLU epilogue. The 3×3 Conv2D uses (1, 1) padding and the 1×1 Conv2D uses (1, 1) strides and does not have padding.

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

Conv performance

Evaluation

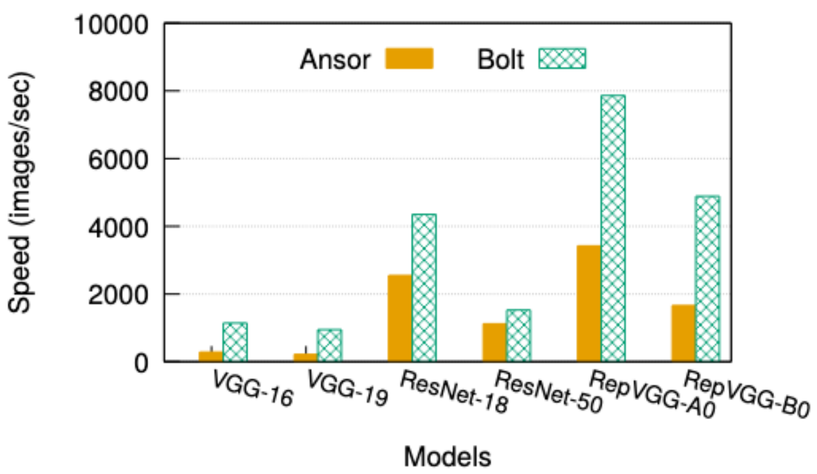
Microbenchmarks 4 Padding performance and overhead

Table 3. The performance and overhead of Bolt's automated padding. Unpadded Conv2Ds are computed with alignment=2; after being padded, alignement=8 can be used. The cost of padding is the time spent on the padding over the total computation time (padding+Conv2D).

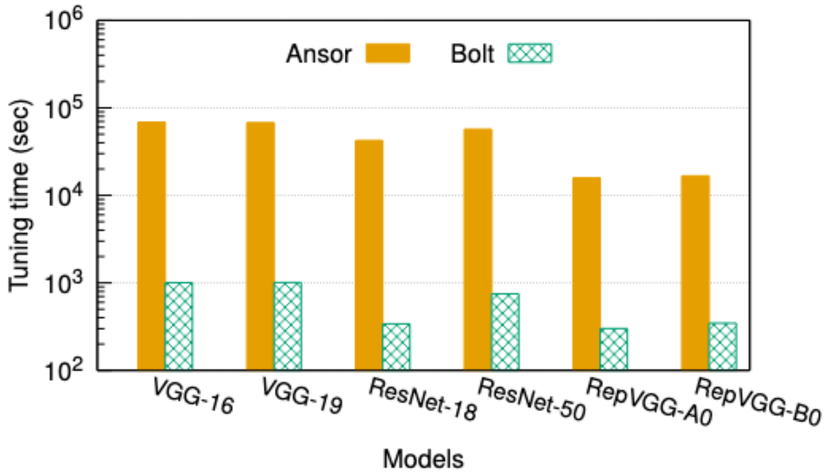
N	H, W	IC, OC	kernel	padding	Norm. speed		Cost
					unpad	pad	
32	20, 26	46, 32	(3, 3)	(1, 1)	1.00	1.62	18%
32	20, 26	46, 32	(5, 5)	(2, 2)	1.00	1.95	9%
128	14, 19	46, 32	(5, 7)	(0, 0)	1.00	1.77	15%
288	11, 15	46, 32	(5, 7)	(0, 0)	1.00	1.71	18%
32	20, 26	174, 64	(3, 3)	(1, 1)	1.00	1.60	24%
32	20, 26	174, 64	(5, 5)	(2, 2)	1.00	1.99	12%

Evaluation

End-to-end optimization



(a) Inference speed.



(b) Tunning time.

Figure 10. The normalized inference speed and tuning time for widely used convolutional neural networks.

Evaluation

System-friendly models: RepVGG case study

Table 4. The top-1 accuracy and speed of RepVGG-A0 using different activation functions (120 epochs + simple data augmentation).

Activation	Top-1 accuracy	Speed (images/sec)
ReLU	72.31	5909
GELU	72.38	5645
Hardswish	72.98	5713
Softplus	72.57	5453

Changing activation functions

Table 5. The top-1 accuracy and speed of original RepVGG models and their augmentation with 1×1 Conv2Ds (200 epochs + simple data augmentation).

Model	Top-1 accuracy	Speed	Params
RepVGG-A0	73.05	7861	8.31
RepVGG-A1	74.75	6253	12.79
RepVGG-B0	75.28	4888	14.34
RepVGGAug-A0	73.87	6716	13.35
RepVGGAug-A1	75.52	5241	21.7
RepVGGAug-B0	76.02	4145	24.85

Deepening the model with 1x1 Conv2Ds

Table 6. The top-1 accuracy and speed of original RepVGG models and their augmentation with 1×1 Conv2Ds+Hardswish (300 epochs + advanced augmentation, label smoothing, and mixup).

Model	Top-1 accuracy	Speed (images/sec)
RepVGG-A0	73.41	7861
RepVGG-A1	74.89	6253
RepVGG-B0	75.89	4888
RepVGGAug-A0	74.54	6338
RepVGGAug-A1	76.72	4868
RepVGGAug-B0	77.22	3842

Conclusion

- **Bridge the gap** between auto-tuners and hardware-native performance by **combining templated vendor library & auto-tuner**
- Purpose persistent kernel fusion
- **Three system-model codesign principles**

https://github.com/apache/tvm/blob/26aeaee04699ec3c67721abe1378aa15815e780b/docs/reference/publications.rst#L78

Thanks

2023-11-27

Presented by Guangtong Li