DNNFusion: Accelerating Deep Neural Networks Execution with Advanced Operator Fusion

(PLDI 2021)

Wei Niu[1], Jiexiong Guan[1], Yanzhi Wang[2],
Gagan Agrawal[3], Bin Ren[1]

[1]William & Mary, [2]Northeastern University, [3]Augusta University

Presenter: Taehee Jeong

Optimizing Deep Neural Networks

Deep networks need high memory and computation requirements

- **Operator fusion** is a key optimization technique in many frameworks (TF, TVM, MNN)

Existing approaches adapt **too restrictive fusion strategies** (i.e. few hand-coded pattern matching rules)

- Not able to cover **diversity** of operators and layer connections
- *Polyhedral-based* loop fusion techniques: focused on **affine-loop** optimizations, won’t be able to capture some operation combinations

Contributions

- Graph rewriting based on **mathematical property** of operations
- Operator fusion plan generation from high-level operator abstractions (with **mapping types** and mathematical properties)
- Optimized code generation for **fused** blocks

⇒ Up to **8.8x** more loop fusions and **9.3x** speedup compared to existing frameworks

Deeper models have lesser throughput

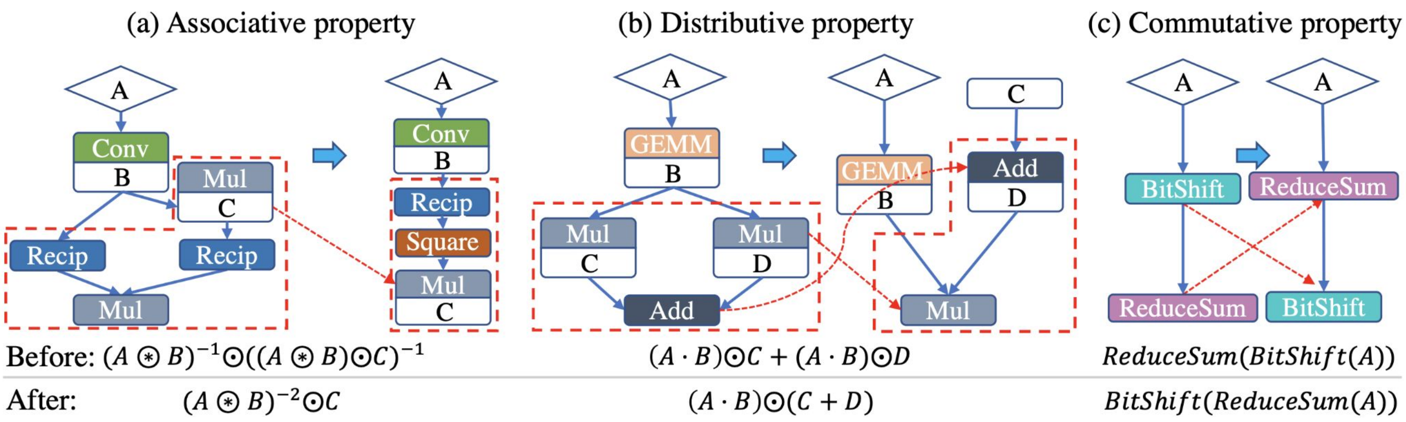
Model	#Total layer	IR size	#FLOPS	Speed (FLOPs/S)
VGG-16 [62]	51	161M	31.0B	320G
YOLO-V4 [7]	398	329M	34.6B	135G
DistilBERT [60]	457	540M	35.3B	78G
MobileBERT [65]	2,387	744M	17.6B	44G
GPT-2 [55]	2,533	1,389M	69.1B	62G

- More layer: more intermediate results
    - Memory/cache pressure ↑
    - ex) **`reshape, squeeze`** in TFLite are just **`memcpy`**!
- Insufficient amount of computations per layer
    - Utilization ↓ (esp. for GPU)

⇒ Operator fusion can pack more computations per (fused) layer

Mathematical property based graph rewriting

- Mathematical properties: associative, distributive, commutative
- Identifies set of rewrite rules with **operator fusion in mind**
  - Results in fewer layers in the final fused graph
- Focus on 1-to-1 mappings and reduction operators (many-to-many)
- Optimizes based on FLOPS (and memory footprint for tie)



Mathematical property based graph rewriting

- Rewrite candidate search: pattern matching
  - NP-complete problem
- Partition graph on the points where ops don't have mathematical properties
  - Each partition would (*hopefully?*) have tractable search space
  - All cases are considered, and one with least FLOPS is chosen.

- Isn't compiler already doing it after codegen?
  - Operation on the tensors might not be easily optimized as scalars

Fusion planning: DNN operator classification

Mapping type	Representative
One-to-One	Add, Relu
One-to-Many	Expand
Many-to-Many & N-to-1 (e.g. reduce)	Conv, GEMM
Reorganize	Reshape
Shuffle	Transpose

Second op / First op	One-to-One	One-to-Many	Many-to-Many	Reorganize	Shuffle
One-to-One	One-to-One	One-to-Many	Many-to-Many	Reorganize	Shuffle
One-to-Many	One-to-Many	One-to-Many	×	One-to-Many	One-to-Many
Many-to-Many	Many-to-Many	Many-to-Many	×	Many-to-Many	Many-to-Many
Reorganize	Reorganize	One-to-Many	Many-to-Many	Reorganize	Reorganize ?
Shuffle	Shuffle	One-to-Many	Many-to-Many	Reorganize	Shuffle

ECG (Extended computational graph) IR will have mapping class annotation for each operator

Fusion opportunity

Green: fusion is profitable
Yellow(orange?): further profiling is required (from empirical results)
Red: Unprofitable

Fusion planning: DNN operator classification

Second op / First op	One-to-One	One-to-Many	Many-to-Many	Reorganize	Shuffle
One-to-One	One-to-One	One-to-Many	Many-to-Many	Reorganize	Shuffle

One-to-one: follows the class of the other fused op

- As the values are directly mapped, fused kernel is not likely to require extra overhead, and only need limited number of registers.

ex) **`GEMM(Add(x, y), W)`** will be fused to **`GEMM(x + y, W)`** in codegen step.

Fusion planning: DNN operator classification

First op \ Second op	One-to-One	One-to-Many	Many-to-Many	Reorganize	Shuffle
Reorganize	Reorganize	One-to-Many	Many-to-Many	Reorganize	Reorganize
Shuffle	Shuffle	One-to-Many	Many-to-Many	Reorganize	Shuffle

Reorder / Shuffle: can be considered as one-to-one with special mapping

one-to-many / many-to-many fusion should be handled with care

- possible data copying, data access order, redundant computations

ex) **`Transpose(Expand(x))`** would break continuous memory access pattern when fused

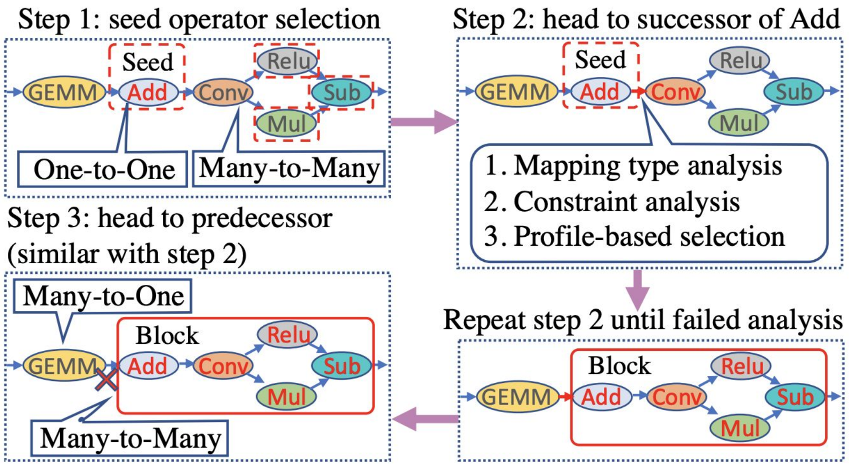
Fusion planning: DNN operator classification

Second op / First op	One-to-Many	Many-to-Many
One-to-Many	One-to-Many	×
Many-to-Many	Many-to-Many	×

- 1-to-N & N-to-M (`Conv(Expand(x))`): conv requires **continuous** memory access *vs* expand would need **distributed** access -> **unprofitable**
- N-to-M & M-to-K (`Conv(Conv(x))`): too complicated. **unprofitable**
- N-to-M & 1-to-K (`Expand(Conv(x))`): won’t affect conv’s memory access pattern when expanding one dimension -> **might be OK**
    - `Reshape(Conv(x))`: interfere with memory access pattern -> **unprofitable**

Fusion plan generation

1. Choose **1-to-1** operator with **minimal result** as seed
2. By traversing predecessors and successors, **greedily** group the operators that can be fused
3. Repeat until there's no more fusion candidates



Other Fusion-related Optimizations

- **Intra**-block optimizations
  - Replace shuffle/reorganize operations with index transform (changed data index)
- **Inter**-block optimizations
  - (Heuristic) Choose one memory layout that would benefit the most **compute-heavy** op in the fusion block

Codegen

- Generates code for each fused block with data-flow tree (DFT)
- 23 codegen rules, for each mapping class combination
- redundant computation checks from DFT

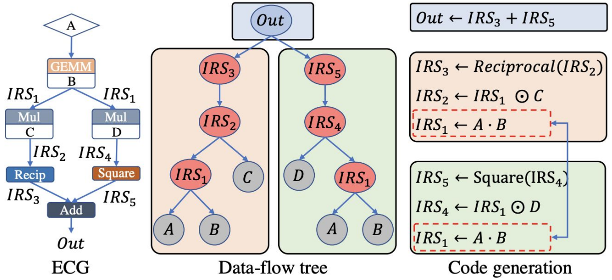


Figure 4. Code generation.

- Paper says other optimizations were introduced in *PatDNN* (former paper), but not much information
- Seems to have combined all techniques from other frameworks

Table 1. DNN acceleration frameworks on mobile devices.

DNNs	Optimization Knobs	TFLite	TVM	MNN	**Ours**
Dense	Parameters auto-tuning	N	Y	N	**Y**
	CPU/GPU support	Y	Y	Y	**Y**
	Half-floating support	Y	Y	Y	**Y**
	Computation graph optimization	$Y^{!}$	Y^{*}	$Y^{!}$	$\mathbf{Y}^{**}$
	Tensor optimization	$Y^{!}$	$Y^{\dagger}$	$Y^{!}$	$\mathbf{Y}^{\dagger\dagger}$

(PatDNN, ASPLOS 2020)

Putting it all together

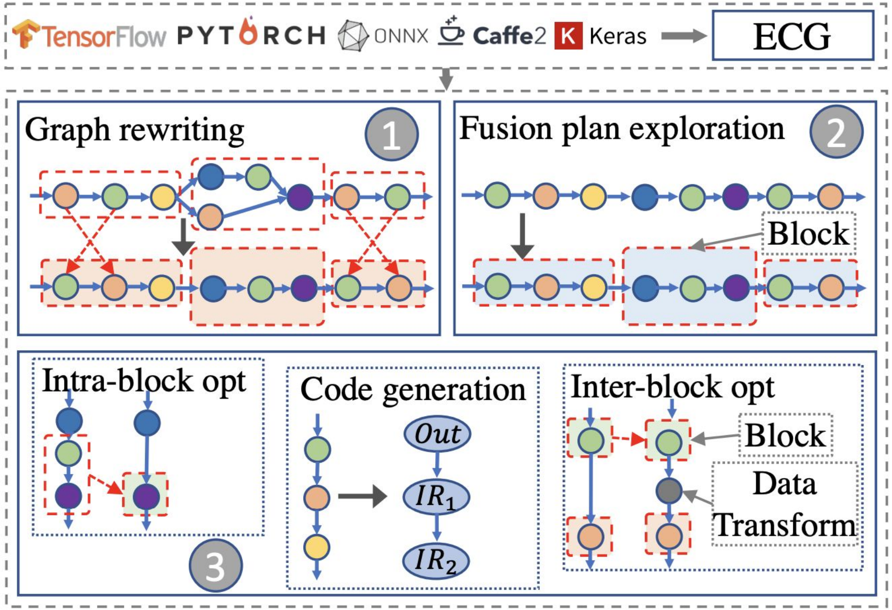


Figure 1. DNNFusion overview.

Results - fusion

Table 5. Fusion rate evaluation: computation layer count and intermediate result size for all evaluated DNNs. CIL (Compute-Intensive Layer): each input is used more than once, e.g. MatMul, CONV. MIL (Memory-Intensive Layer): each input is used only once, e.g. Activation. IRS: intermediate results. '-' means this framework does not support this model.

Model	Type	Task	Layer counts and IRS sizes before opt.				Layer counts and IRS sizes after opt.					
			#CIL	#MIL	#Total layer	IRS size	MNN	TVM	TFLite	Pytorch	**DNNF**	**IRS size**
EfficientNet-B0	2D CNN	Image classification	82	227	309	108MB	199	195	201	210	**97**	**26MB**
VGG-16	2D CNN	Image classification	16	35	51	161MB	22	22	22	22	**17**	**52MB**
MobileNetV1-SSD	2D CNN	Object detection	16	48	202	110MB	138	124	138	148	**71**	**37MB**
YOLO-V4	2D CNN	Object detection	106	292	398	329MB	198	192	198	232	**135**	**205MB**
C3D	3D CNN	Action recognition	11	16	27	195MB	27	27	-	27	**16**	**90MB**
S3D	3D CNN	Action recognition	77	195	272	996MB	-	-	-	272	**98**	**356MB**
U-Net	2D CNN	Image segmentation	44	248	292	312MB	241	232	234	-	**82**	**158MB**
Faster R-CNN	R-CNN	Image segmentation	177	3,463	3,640	914MB	-	-	-	-	**942**	**374MB**
Mask R-CNN	R-CNN	Image segmentation	187	3,812	3,999	1,524MB	-	-	-	-	**981**	**543MB**
TinyBERT	Transformer	NLP	37	329	366	183MB	-	304†	322	-	**74**	**55MB**
DistilBERT	Transformer	NLP	55	402	457	540MB	-	416†	431	-	**109**	**197MB**
ALBERT	Transformer	NLP	98	838	936	1,260MB	-	746†	855	-	**225**	**320MB**
$\text{BERT}_{\text{BASE}}$	Transformer	NLP	109	867	976	915MB	-	760†	873	-	**216**	**196MB**
MobileBERT	Transformer	NLP	434	1,953	2,387	744MB	-	1,678†	2,128	-	**510**	**255MB**
GPT-2	Transformer	NLP	84	2,449	2,533	1,389MB	-	2,047†	2,223	-	**254**	**356MB**

† TVM does not support this model on mobile. This layer count number is collected on a laptop platform for reference.

Results - fusion

Model	Type	Task	Layer counts and IRS sizes before opt.				Layer counts and IRS sizes after opt.					
			#CIL	#MIL	#Total layer	IRS size	MNN	TVM	TFLite	Pytorch	**DNNF**	**IRS size**
EfficientNet-B0	2D CNN	Image classification	82	227	309	108MB	199	195	201	210	**97**	**26MB**
VGG-16	2D CNN	Image classification	16	35	51	161MB	22	22	22	22	**17**	**52MB**
MobileNetV1-SSD	2D CNN	Object detection	16	48	202	110MB	138	124	138	148	**71**	**37MB**

Overall impressive, few notes:

- VGG has 22 layers (w/o activation): 13 conv, 5 pooling, 3 dense, 1 softmax
  - 17 seems *13 conv + 3 dense + 1 softmax*. poolings are seems to be fused, although considered many-to-many?
- MobileNetV1-SSD in TFLite can have 64 layers, with predefined fusion of NMS to custom op `TFLite_Detection_PostProcess`.
  - Also shows faster performance than the evaluation result (87ms vs 29ms)
  - Note: 29ms was on Pixel 4, both S20 and Pixel 4 runs on SD865 SoC
  - Hand-crafted fusion kernels still works better for some cases.

Results - latency improvement

Table 6. Inference latency comparison: DNNFusion, MNN, TVM, TFlite, and PyTorch on mobile CPU and GPU. #FLOPS denotes the number of floating point operations. OurB is our baseline implementation by turning off all fusion optimizations and OurB+ is OurB with a fixed-pattern fusion as TVM. DNNF is short for DNNFusion, i.e., our optimized version. '-' denotes this framework does not support this execution.

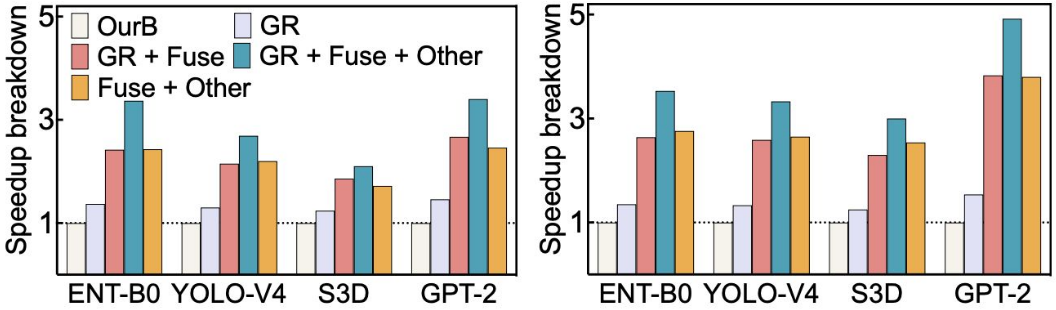
Model	#Params	#FLOPS	MNN (ms)		TVM (ms)		TFLite (ms)		Pytorch (ms)		OurB (ms)		OurB+ (ms)		**DNNF (ms)**	
			CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	**CPU**	**GPU**
EfficientNet-B0	5.3M	0.8B	41	26	56	27	52	30	76	-	54	35	38	24	**16**	**10**
VGG-16	138M	31.0B	242	109	260	127	245	102	273	-	251	121	231	97	**171**	**65**
MobileNetV1-SSD	9.5M	3.0B	67	43	74	52	87	68	92	-	79	56	61	39	**33**	**17**
YOLO-V4	64M	34.6B	501	290	549	350	560	288	872	-	633	390	426	257	**235**	**117**
C3D	78M	77.0B	867	-	1,487	-	-	-	2,541	-	880	551	802	488	**582**	**301**
S3D	8.0M	79.6B	-	-	-	-	-	-	6,612	-	1,409	972	1,279	705	**710**	**324**
U-Net	2.1M	15.0B	181	106	210	120	302	117	271	-	227	142	168	92	**99**	**52**
Faster R-CNN	41M	47.0B	-	-	-	-	-	-	-	-	2,325	3,054	1,462	1,974	**862**	**531**
Mask R-CNN	44M	184B	-	-	-	-	-	-	-	-	5,539	6,483	3,907	4,768	**2,471**	**1,680**
TinyBERT	15M	4.1B	-	-	-	-	97	-	-	-	114	89	92	65	**51**	**30**
DistilBERT	66M	35.5B	-	-	-	-	510	-	-	-	573	504	467	457	**224**	**148**
ALBERT	83M	65.7B	-	-	-	-	974	-	-	-	1,033	1,178	923	973	**386**	**312**
$\text{BERT}_{\text{Base}}$	108M	67.3B	-	-	-	-	985	-	-	-	1,086	1,204	948	1,012	**394**	**293**
MobileBERT	25M	17.6B	-	-	-	-	342	-	-	-	448	563	326	397	**170**	**102**
GPT-2	125M	69.1B	-	-	-	-	1,102	-	-	-	1,350	1,467	990	1,106	**394**	**292**

Results - latency improvement

Table 6. Inference latency comparison: DNNFusion, MNN, TVM, TFlite, and PyTorch on mobile CPU and GPU. #FLOPS denotes the number of floating point operations. OurB is our baseline implementation by turning off all fusion optimizations and OurB+ is OurB with a fixed-pattern fusion as TVM. DNNF is short for DNNFusion, i.e., our optimized version. '-' denotes this framework does not support this execution.

Model	#Params	#FLOPS	MNN (ms)		TVM (ms)		TFLite (ms)		Pytorch (ms)		OurB (ms)		OurB+ (ms)		**DNNF (ms)**	
			CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	**CPU**	**GPU**
EfficientNet-B0	5.3M	0.8B	41	26	56	27	52	30	76	-	54	35	38	24	**16**	**10**
VGG-16	138M	31.0B	242	109	260	127	245	102	273	-	251	121	231	97	**171**	**65**
MobileNetV1-SSD	9.5M	3.0B	67	43	74	52	87	68	92	-	79	56	61	39	**33**	**17**
YOLO-V4	64M	34.6B	501	290	549	350	560	288	872	-	633	390	426	257	**235**	**117**
C3D	78M	77.0B	867	-	1,487	-	-	-	2,541	-	880	551	802	488	**582**	**301**
S3D	8.0M	79.6B	-	-	-	-	-	-	6,612	-	1,409	972	1,279	705	**710**	**324**
U-Net	2.1M	15.0B	181	106	210	120	302	117	271	-	227	142	168	92	**99**	**52**
Faster R-CNN	41M	47.0B	-	-	-	-	-	-	-	-	2,325	3,054	1,462	1,974	**862**	**531**
Mask R-CNN	44M	184B	-	-	-	-	-	-	-	-	5,539	6,483	3,907	4,768	**2,471**	**1,680**
TinyBERT	15M	4.1B	-	-	-	-	97	-	-	-	114	89	92	65	**51**	**30**
DistilBERT	66M	35.5B	-	-	-	-	510	-	-	-	573	504	467	457	**224**	**148**
ALBERT	83M	65.7B	-	-	-	-	974	-	-	-	1,033	1,178	923	973	**386**	**312**
$BERT_{Base}$	108M	67.3B	-	-	-	-	985	-	-	-	1,086	1,204	948	1,012	**394**	**293**
MobileBERT	25M	17.6B	-	-	-	-	342	-	-	-	448	563	326	397	**170**	**102**
GPT-2	125M	69.1B	-	-	-	-	1,102	-	-	-	1,350	1,467	990	1,106	**394**	**292**

Results - latency improvement

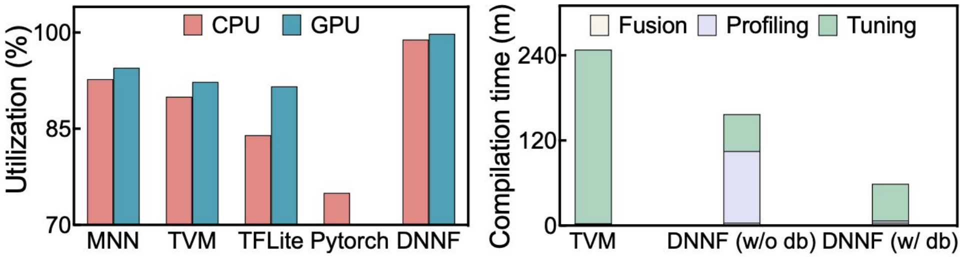


(a) CPU. **(b)** GPU.

Figure 7. Optimization breakdown on y-axis: speedup over OurB, i.e. a version w/o fusion opt. GR, Fuse, and Other denote graph rewriting, fusion, and other fusion-related optimizations, respectively.

Results - others

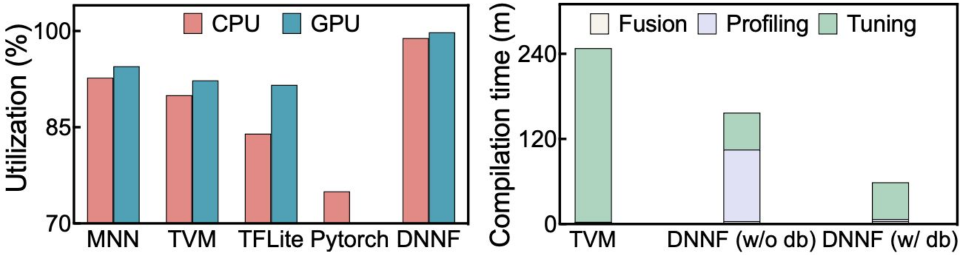
- vs TASO w/ TFLite: 1.4x~2.6x speedup
  - Possibly due to fusion and codegen
- Memory access and cache misses: improved from all frameworks
- G/GPU Utilization: better, > 90%
- Compilation time (vs TVM): best when profiling result can be looked up



(a) CPU and GPU utilization. **(b)** Compilation time.

Results - others

- vs TASO w/ TFLite: 1.4x~2.6x speedup
    - Possibly due to fusion and codegen
- Memory access and cache misses: improved from all frameworks
- G/GPU Utilization: better, > 90%
- Compilation time (vs TVM): best when profiling result can be looked up
- Can be applied to other targets



(a) CPU and GPU utilization. **(b)** Compilation time.

Takeaways

- Results and related work sections also serve as great **reference points**
- **Simple greedy algorithm** for finding fusion candidates works surprisingly well
- **Unlocked** mobile inference for various models that were not possible for prior frameworks

- Accuracy is not reported - might be needed to mention correctness/exactness analysis result after aggressive fusion
- Some details are questionable - fused pooling in VGG, details for codegen