Tensor Compilers for LLM

Yue Guan | Sep. 2025

Introduction

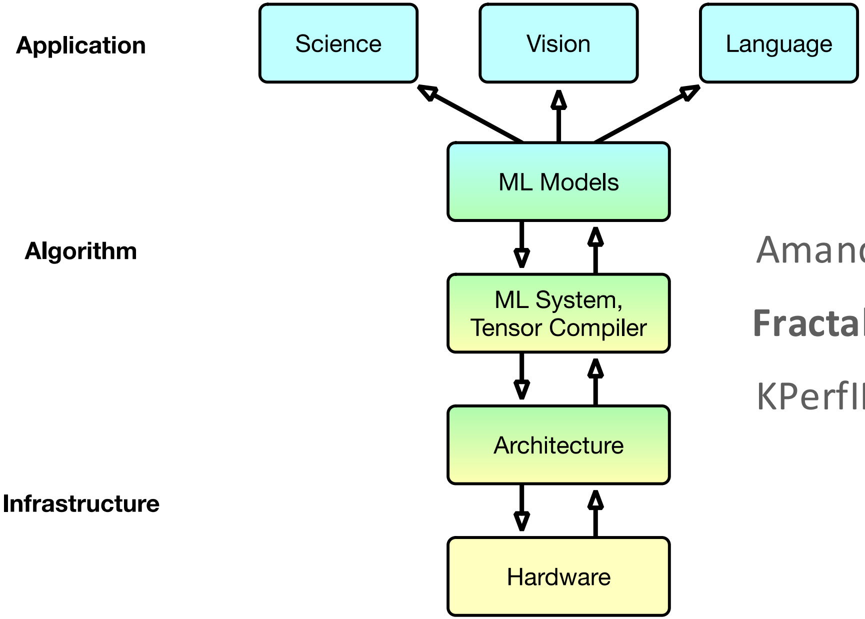
Yue Guan

- Postdoc at UCSD, Picasso Lab, supervised by Prof. Yufei Ding
- Research Interests: ML System, Tensor Compiler

- Publications on ML Compilers
- [ASPLOS24] Fractal: Joint Multi-Level Sparse Pattern Tuning of Accuracy and Performance for DNN Pruning
- [ASPLOS24] Amanda: Unified Instrumentation Framework for Deep Neural Networks
- [OSDI25] KPerfIR: Towards an Open and Compiler-centric Ecosystem for GPU Kernel Performance Tooling on Modern AI Workloads
- [SOSP25] Mercury: Unlocking Multi-GPU Operator Optimization for LLMs via Remote Memory Scheduling

Introduction

ML-Centric System



Amanda

Fractal, Mercury

KPerfIR

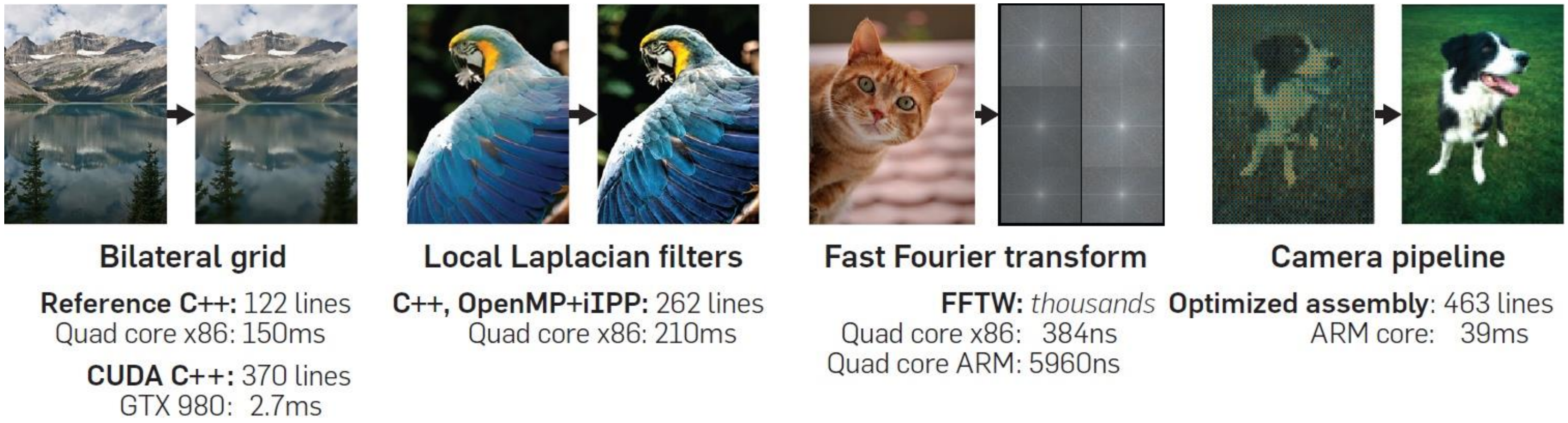
Full stack optimization

Contents

Contents

Halide

A language and compiler for image processing.

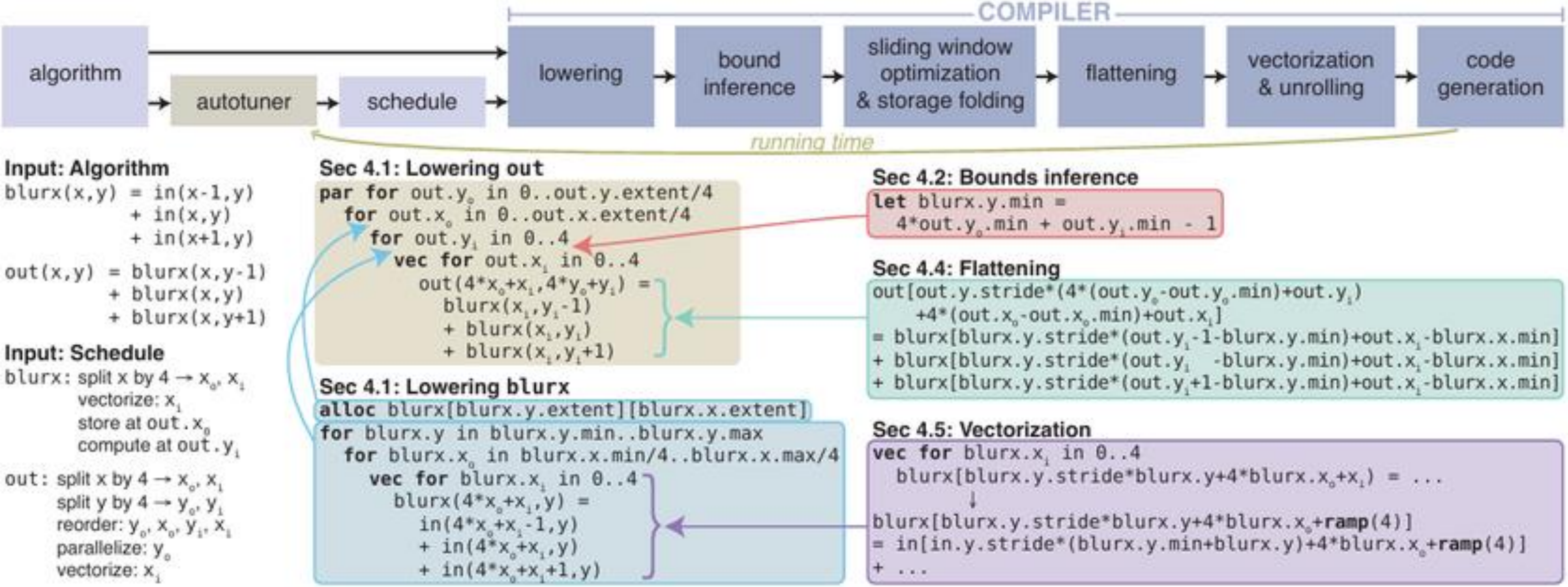


We have so many processing algorithm pipelines.

We have to write many (efficient) implementations for them.

Halide

Separating the algorithm from the schedule
- Algorithm: computation of the program
- Schedule: semantic equivalent transformations (primitives)



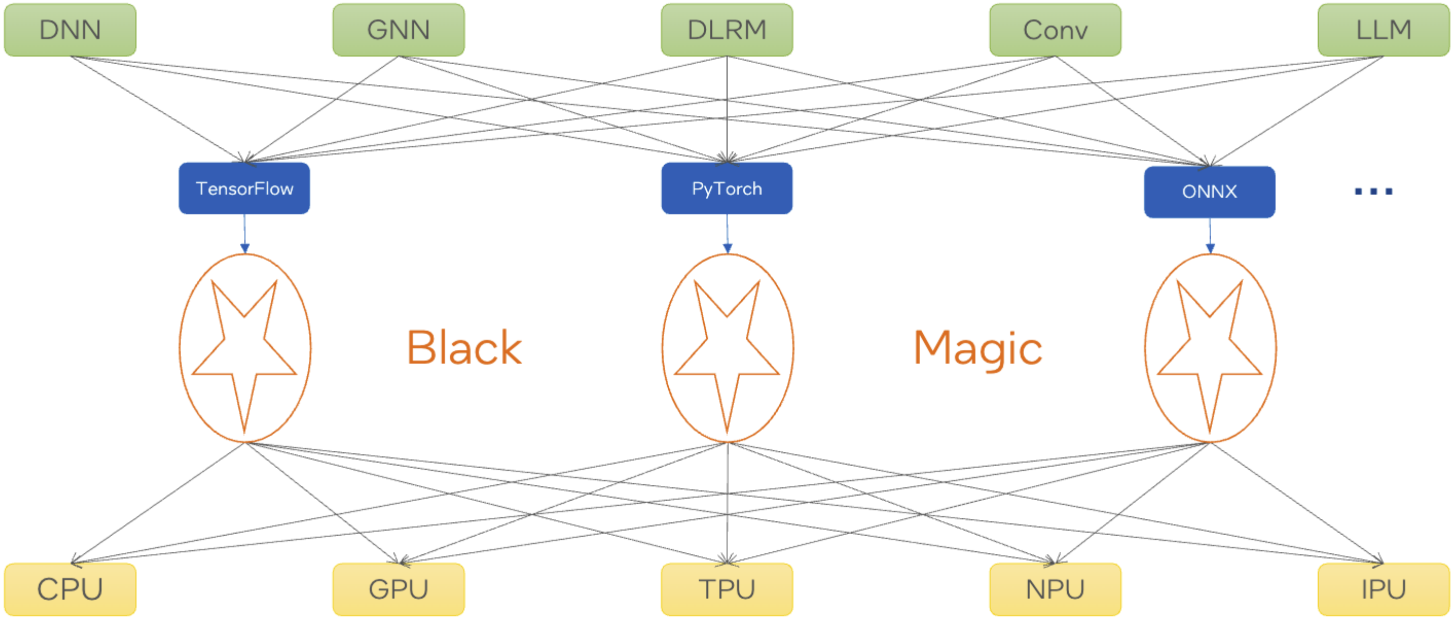
Halide

A language and compiler for image processing.

Bilateral grid	Local Laplacian filters	Fast Fourier transform	Camera pipeline
Reference C++: 122 lines Quad core x86: 150ms **CUDA C++:** 370 lines GTX 980: 2.7ms	**C++, OpenMP+iIPP:** 262 lines Quad core x86: 210ms	**FFTW:** *thousands* Quad core x86: 384ns Quad core ARM: 5960ns	**Optimized assembly**: 463 lines ARM core: 39ms
Halide algorithm: 34 lines **schedule:** 6 lines Quad core x86: 14ms **GPU schedule:** 6 lines GTX 980: 2.3ms	**Halide algorithm:** 62 lines **schedule:** 11 lines Quad core x86: 92ms **GPU schedule:** 9 lines GTX 980: 23ms	**Halide algorithm:** 350 lines **schedule:** 30 lines Quad core x86: 250ns Quad core ARM: 1250ns	**Halide algorithm:** 170 lines **schedule:** 50 lines ARM core: 41ms **DSP schedule:** 70 lines Hexagon 680: 15ms

Tensor Comprehension

The same problem happens in machine learning (ML).



Tensor Comprehension

From image processing to tensor programs.

$$
\begin{array}{l}
\text{Domain} \left[\begin{array}{ll} \{\mathtt{S}(i,j) & \mid 0 \le i < N \wedge 0 \le j < K\} \\ \{\mathtt{T}(i,j,k) & \mid 0 \le i < N \\ & \quad \wedge 0 \le j < K \wedge 0 \le k < M\} \end{array} \right. \\
\quad \text{Sequence} \\
\quad\quad \text{Filter}\{\mathtt{S}(i,j)\} \\
\quad\quad\quad \text{Band}\{\mathtt{S}(i,j) \to (i,j)\} \\
\quad\quad \text{Filter}\{\mathtt{T}(i,j,k)\} \\
\quad\quad\quad \text{Band}\{\mathtt{T}(i,j,k) \to (i,j,k)\}
\end{array}
$$

(a) canonical `sgemm`

$$
\begin{array}{l}
\text{Domain} \left[\begin{array}{ll} \{\mathtt{S}(i,j) & \mid 0 \le i < N \wedge 0 \le j < K\} \\ \{\mathtt{T}(i,j,k) & \mid 0 \le i < N \wedge 0 \le j < K \wedge 0 \le k < M\} \end{array} \right. \\
\quad \text{Band} \left[\begin{array}{ll} \{\mathtt{S}(i,j) & \to (i,j)\} \\ \{\mathtt{T}(i,j,k) & \to (i,j)\} \end{array} \right. \\
\quad\quad \text{Sequence} \\
\quad\quad\quad \text{Filter}\{\mathtt{S}(i,j)\} \\
\quad\quad\quad \text{Filter}\{\mathtt{T}(i,j,k)\} \\
\quad\quad\quad\quad \text{Band}\{\mathtt{T}(i,j,k) \to (k)\}
\end{array}
$$

(b) fused

$$
\begin{array}{l}
\text{Domain} \left[\begin{array}{ll} \{\mathtt{S}(i,j) & \mid 0 \le i < N \wedge 0 \le j < K\} \\ \{\mathtt{T}(i,j,k) & \mid 0 \le i < N \\ & \quad \wedge 0 \le j < K \wedge 0 \le k < M\} \end{array} \right. \\
\quad \text{Band} \left[\begin{array}{ll} \{\mathtt{S}(i,j) & \to (32\lfloor i/32 \rfloor, 32\lfloor j/32 \rfloor)\} \\ \{\mathtt{T}(i,j,k) & \to (32\lfloor i/32 \rfloor, 32\lfloor j/32 \rfloor)\} \end{array} \right. \\
\quad\quad \text{Band} \left[\begin{array}{ll} \{\mathtt{S}(i,j) & \to (i \bmod 32, j \bmod 32)\} \\ \{\mathtt{T}(i,j,k) & \to (i \bmod 32, j \bmod 32)\} \end{array} \right. \\
\quad\quad\quad \text{Sequence} \\
\quad\quad\quad\quad \text{Filter}\{\mathtt{S}(i,j)\} \\
\quad\quad\quad\quad \text{Filter}\{\mathtt{T}(i,j,k)\} \\
\quad\quad\quad\quad\quad \text{Band}\{\mathtt{T}(i,j,k) \to (k)\}
\end{array}
$$

(c) fused and tiled

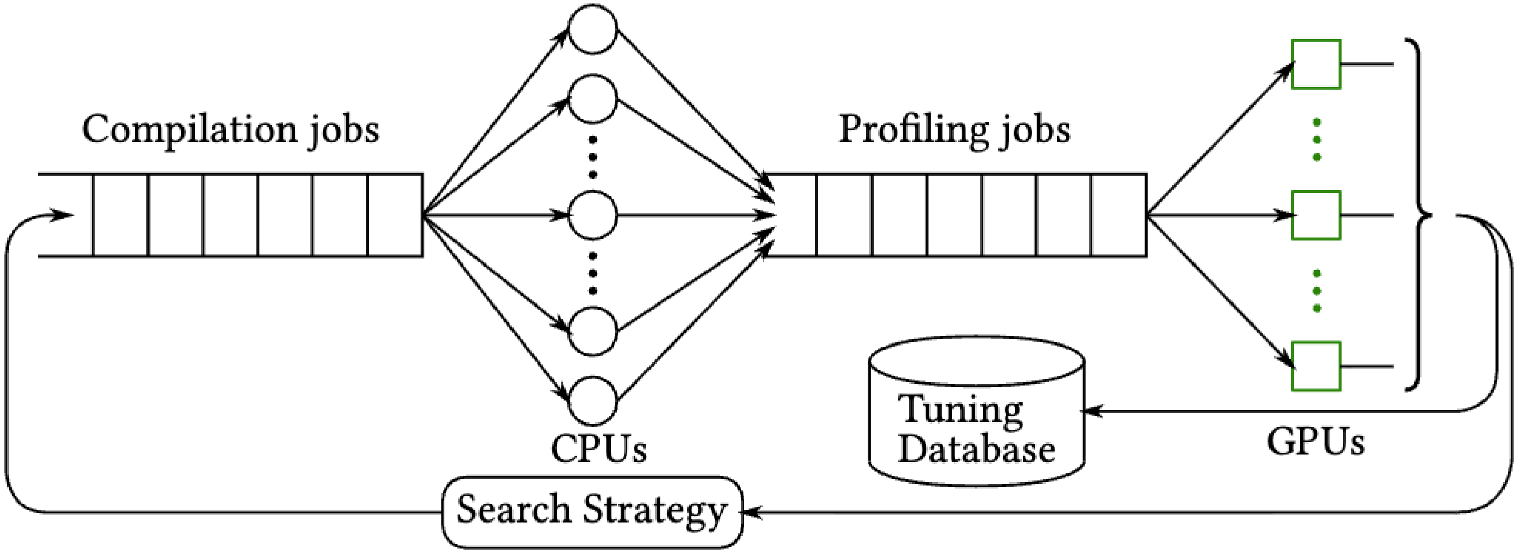
$$
\begin{array}{l}
\text{Domain} \left[\begin{array}{ll} \{\mathtt{S}(i,j) & \mid 0 \le i < N \wedge 0 \le j < K\} \\ \{\mathtt{T}(i,j,k) & \mid 0 \le i < N \wedge 0 \le j < K \wedge 0 \le k < M\} \end{array} \right. \\
\quad \text{Band} \left[\begin{array}{ll} \{\mathtt{S}(i,j) & \to (32\lfloor i/32 \rfloor, 32\lfloor j/32 \rfloor)\} \\ \{\mathtt{T}(i,j,k) & \to (32\lfloor i/32 \rfloor, 32\lfloor j/32 \rfloor)\} \end{array} \right. \\
\quad\quad \text{Sequence} \\
\quad\quad\quad \text{Filter}\{\mathtt{S}(i,j)\} \\
\quad\quad\quad\quad \text{Band}\{\mathtt{S}(i,j) \to (i \bmod 32, j \bmod 32)\} \\
\quad\quad\quad \text{Filter}\{\mathtt{T}(i,j,k)\} \\
\quad\quad\quad\quad \text{Band}\{\mathtt{T}(i,j,k) \to (32\lfloor k/32 \rfloor)\} \\
\quad\quad\quad\quad\quad \text{Band}\{\mathtt{T}(i,j,k) \to (k \bmod 32)\} \\
\quad\quad\quad\quad\quad\quad \text{Band}\{\mathtt{T}(i,j,k0 \to (i \bmod 32, j \bmod 32)\}
\end{array}
$$

(d) fused, tiled and sunk

Tensor Comprehension

Another magic: auto-tuning of transformation schedules.

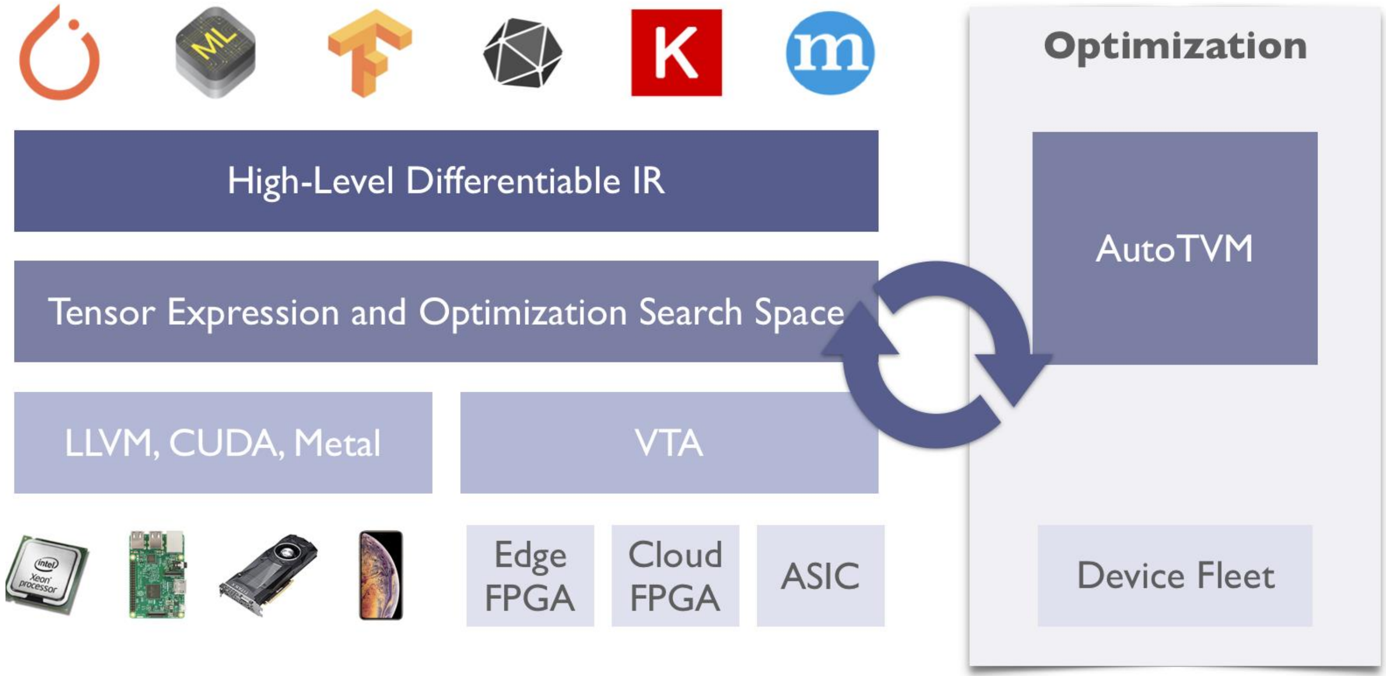
Instead of writing the schedule, we can somehow search the schedule.



TVM

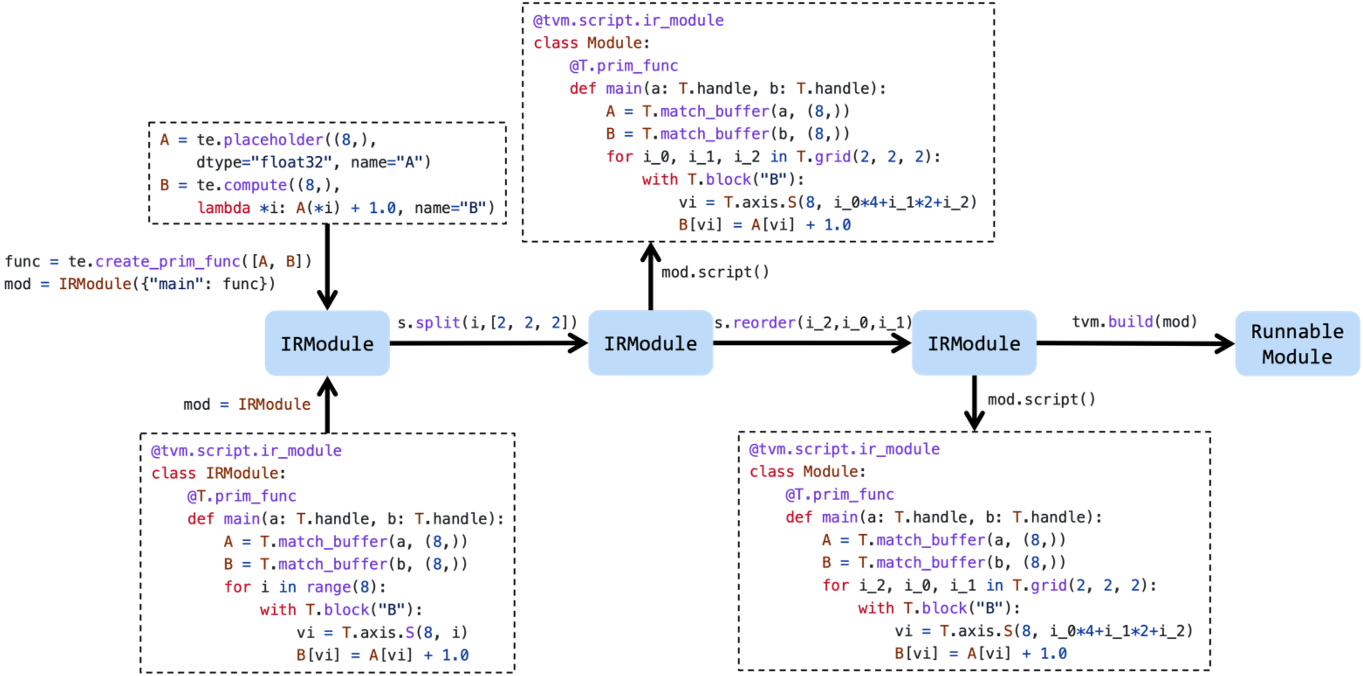
UC San Diego

Generic representation for ML algorithms and accelerators



TVM

Generic representation for ML algorithms and accelerators



TensorIR interactive optimization flow

Summary

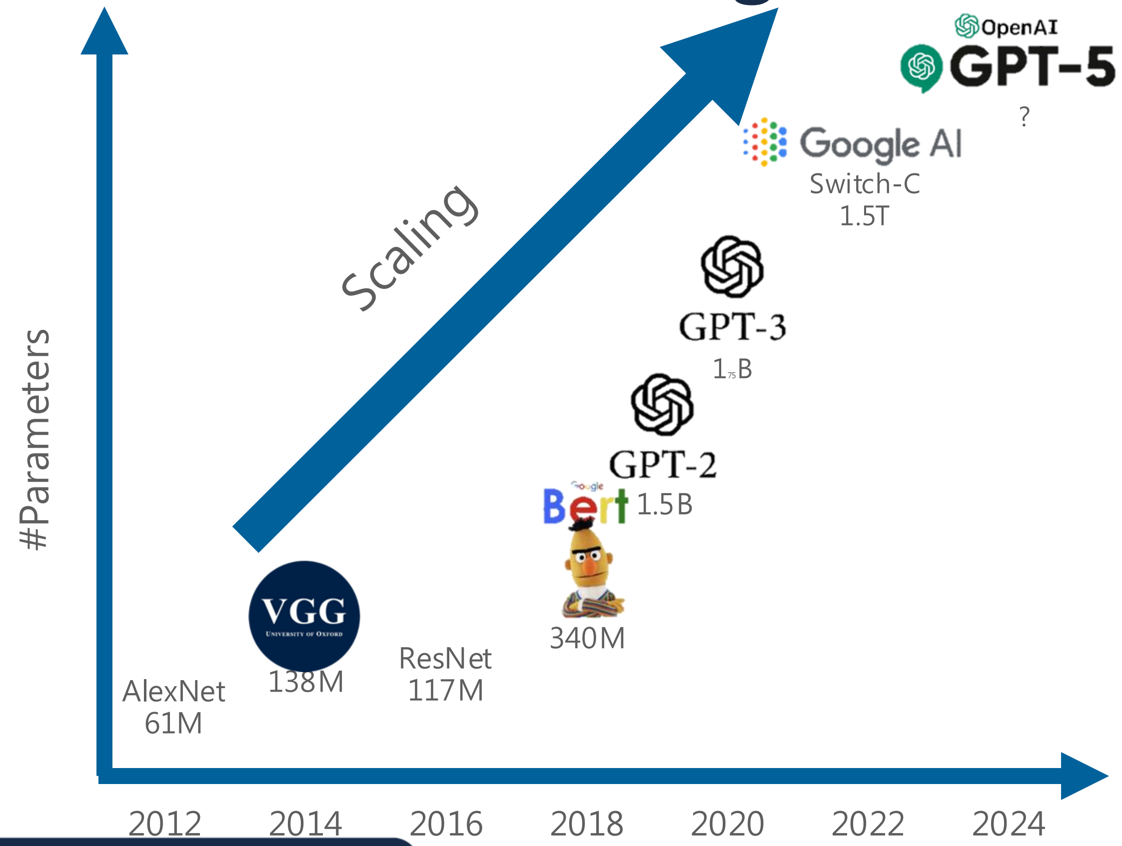
UC San Diego

Compilers	Halide	Tensor Comprehension	TVM	Fractal	Mercury
Domain	Image processing	ML	ML	Sparse ML	ML
Hardware	CPU/GPU	GPU	GPU/Accelerator	GPU/Accelerator	Multi-GPU
Autotuning	Cost-model guided beam-search	Genetic search	Template-free	Enumeration-based	Enumeration-based
Comments	First scheduling language	Early ML support	Extending to accelerators	Extending to sparse ML	Expanding to multi-GPU

Contents

Contents

LLM Model Scaling

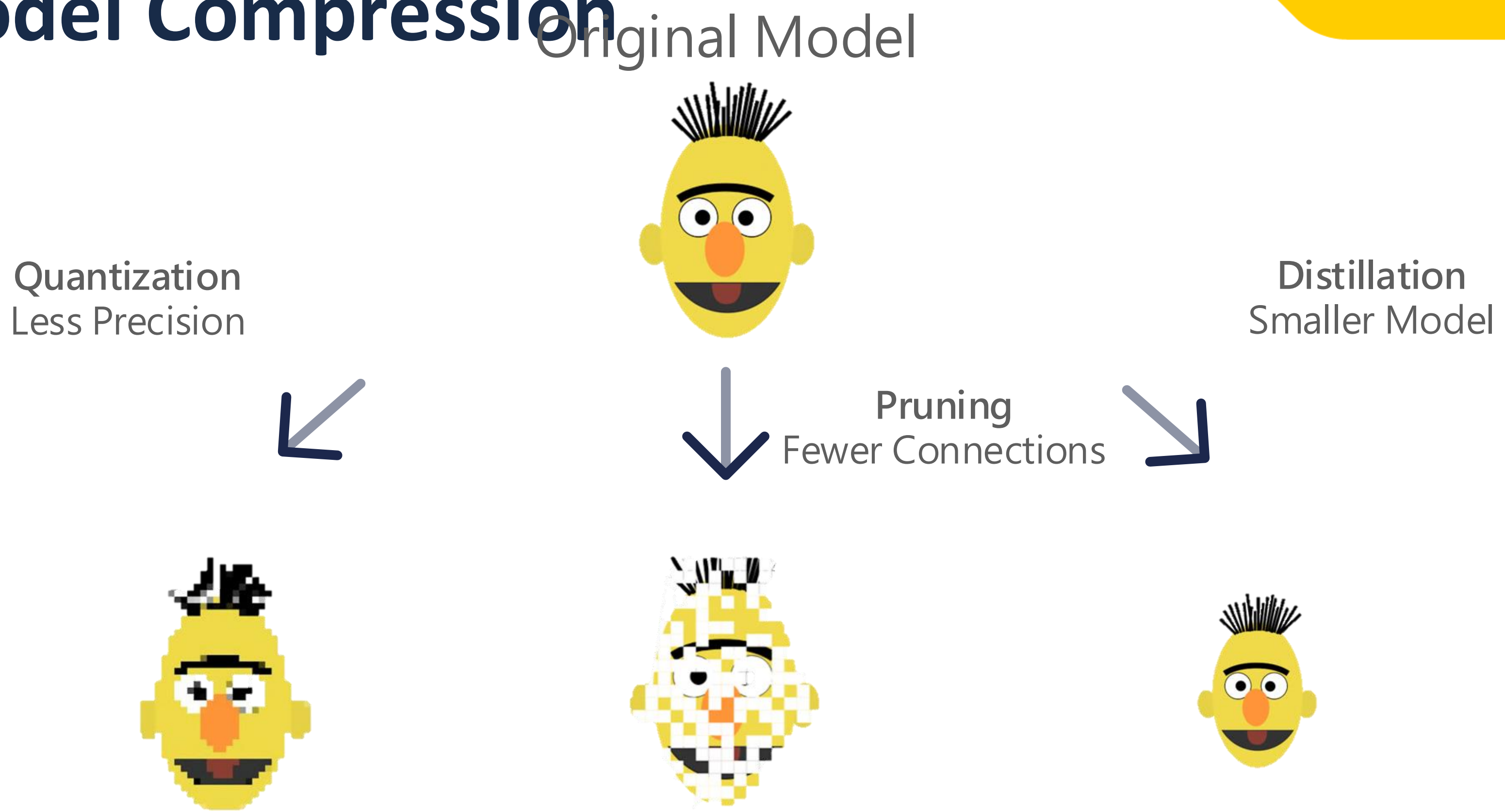


Model size grows rapidly

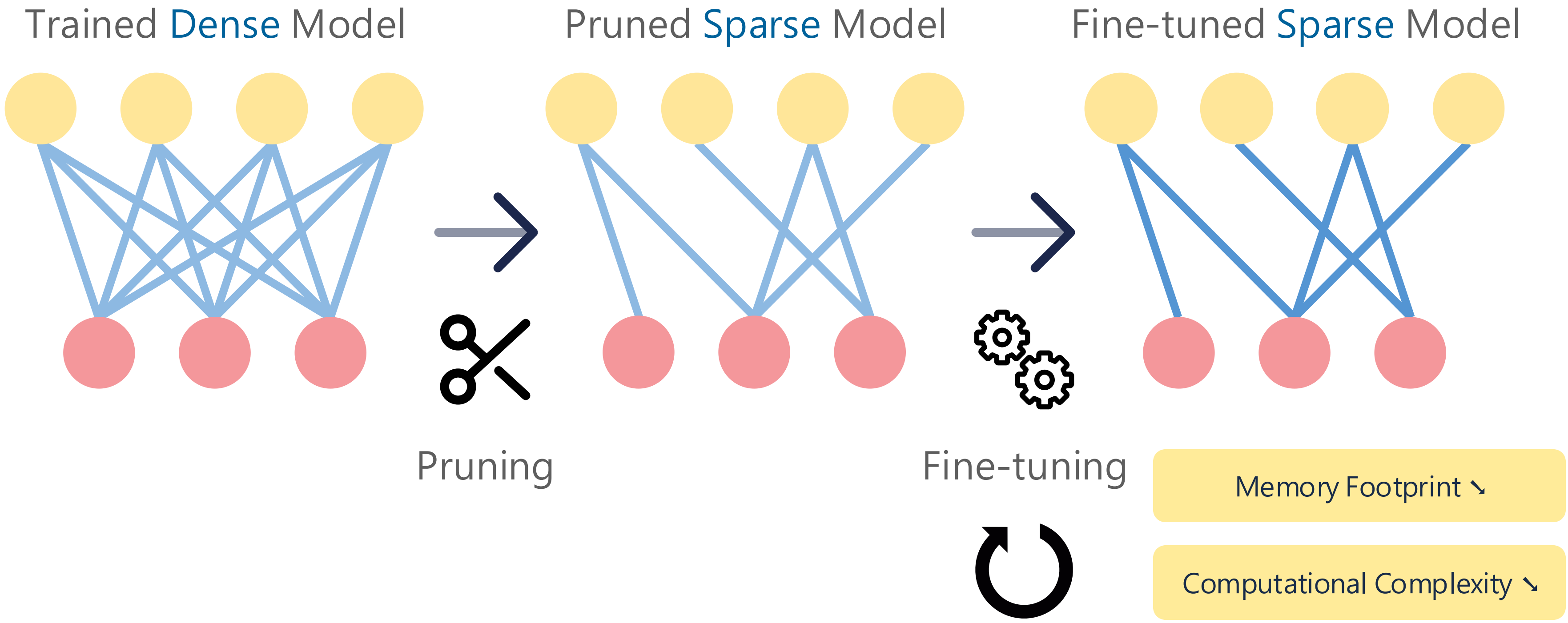
Cost is expensive

Model Compression

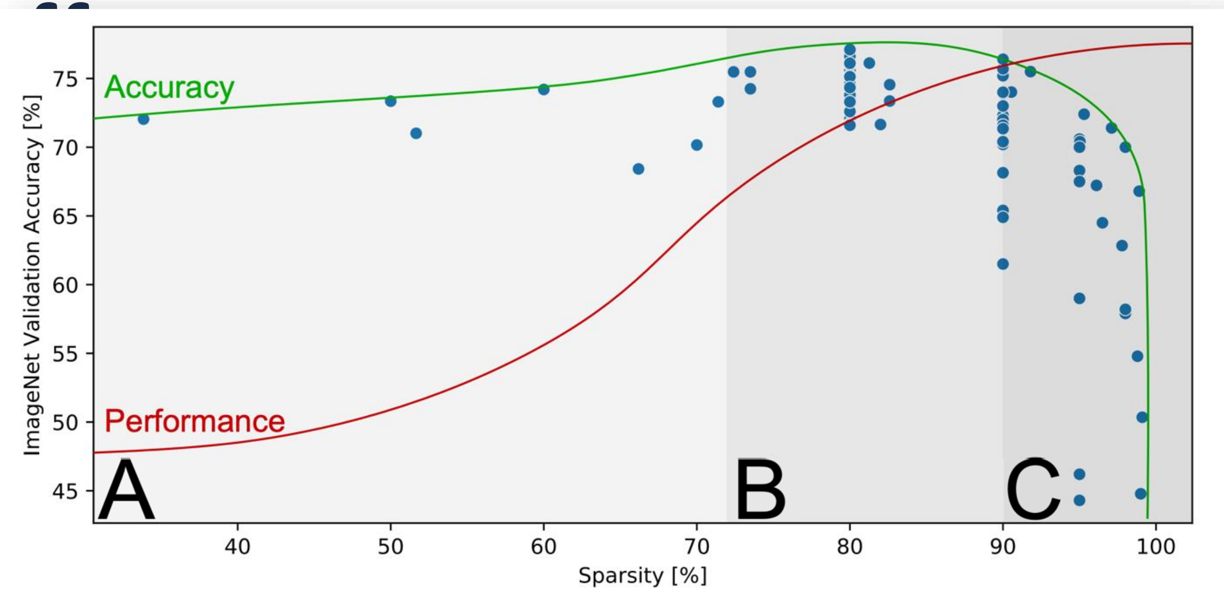
Model Compression



DNN Pruning


Trained Dense Model
Pruned Sparse Model
Fine-tuned Sparse Model
Pruning
Fine-tuning
Memory Footprint ↘
Computational Complexity ↘

Accuracy and Performance Trade

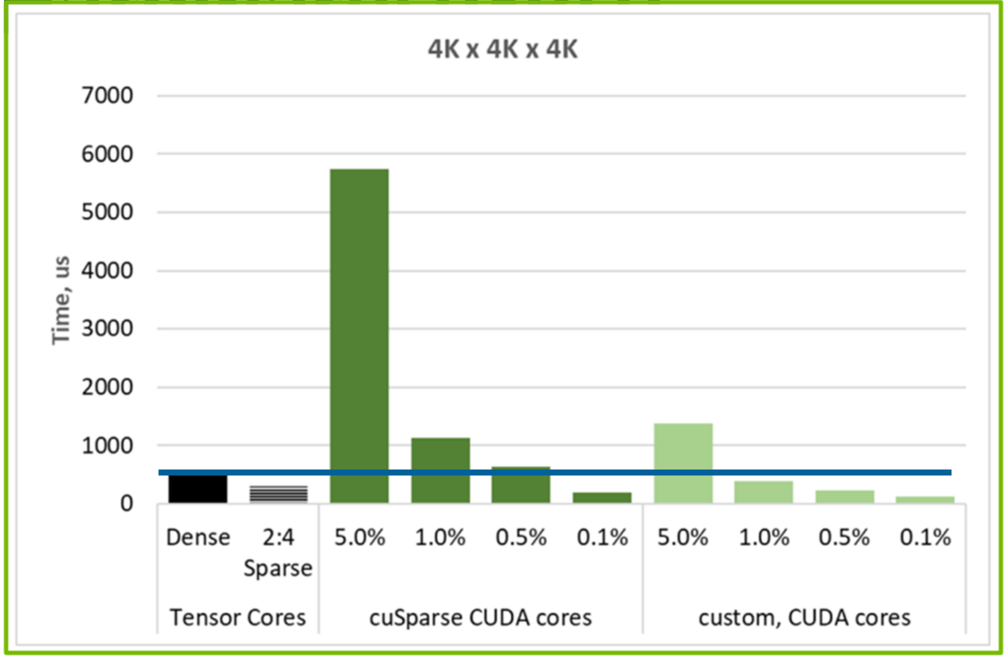


A: Sparsity as Regularization

B: Sweet Point

C: Sparsity Limit

Sparsity Challenge on Hardwares(GPU)

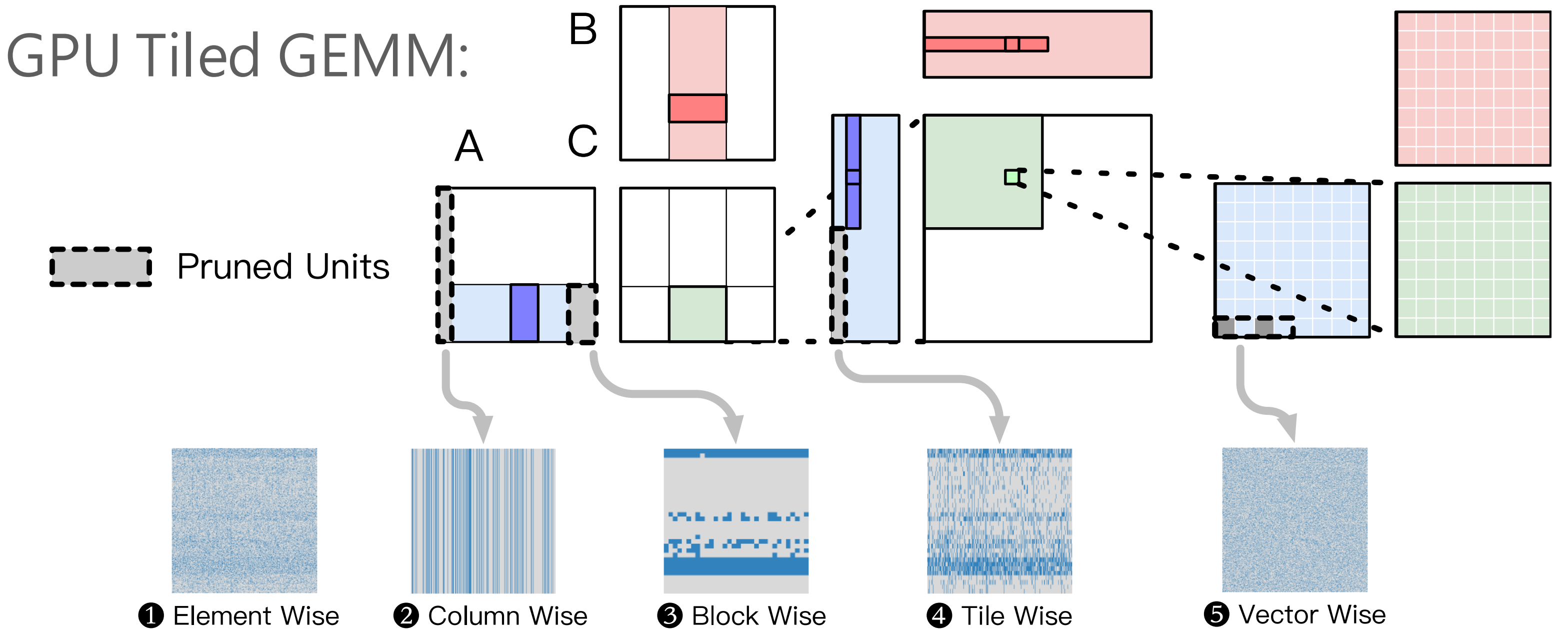


Sparsity is not a friend of GPU

- Random memory access
- Less compute intensity

Over 99% sparsity to beat dense

Structured Pruning Patterns



Maximize hardware performance

Alleviate accuracy degradation

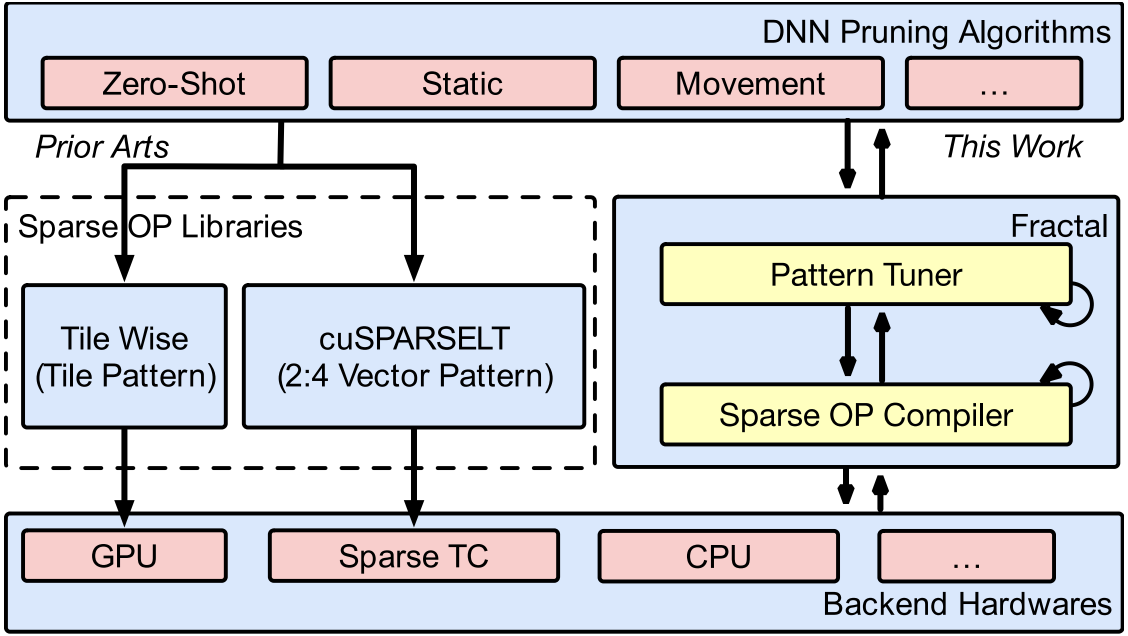
Structured Patterns

Sparse Operator Libraries

Sparse Pattern	Model Accuracy	Operator Library	Hardware Performance	Backends
Element-Wise	😁	cuSPARSE	☹️	CUDA
		Sputnik	☹️	All
Tile-Wise	😐	TileWise	😁	A100,V100
		MagicCube	😐	A100
Block-Wise	☹️	BlockELL	😁	CUDA
		Triton-BW	😐	CUDA
Vector-Wise	😁	cuSPARSELt	😁	Sparse TC

Selecting optimal patterns for: DNN model, operator, backend is challenging.

Need for Pattern Tuning System

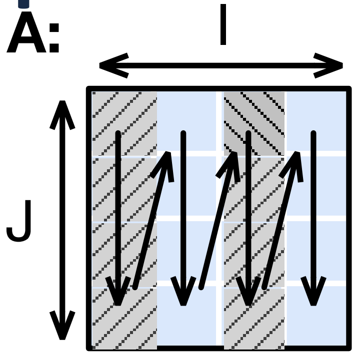


We need a structed sparse pattern tuning system!

UC San Diego

Contents

Loop Perforation



Perforate I 50%

I_indices: 1 | 3

Dense Loop:

```
For (i: int, 0, 4):
    For (j: int, 0, 4):
        A[i, j] = A[i, j]^2
```

Sparse Loop:

```
For (i_index: int, 0, 2):
    i = I_indices[i_index]
    For (j: int, 0, 4):
        A[i, j] = A[i, j]^2
```

Construct structured sparsity pattern with loop perforation.

PatternIR: Loop-based Pattern Represe

UC San Diego

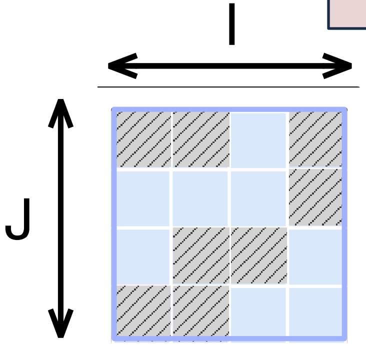
Loop name

Loop dependency

$I_{length}J^{nnz}_{length}$

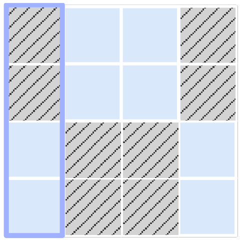
Non-zero elements (sparse length)

Loop full length



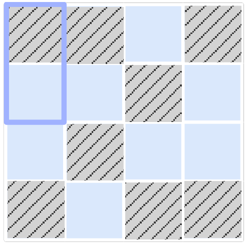

```
For (ij_index: int, 0, 8):
  ij = IJ_indices[ij_index]
```

EW: IJ^{8}_{16}



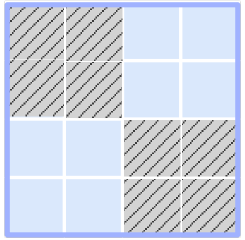
```
For (i: int, 0, 4):
  For (j0_index: int, 0, 1):
    j0 = J0_indices[j0_index]
    For (j1: int, 0, 2):
```

TW: $I_4J0^1_2J1_2$



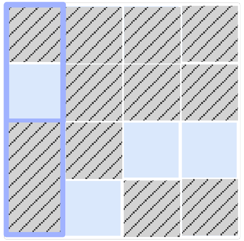
```
For (i: int, 0, 4):
  For (j0: int, 0, 2):
    For (j1_index: int, 0, 1):
      j1 = J1_indices[j1_index]
```

VW: $I_4J0_2J1^1_2$



```
For (i0j0_index: int, 0, 2):
  i0j0 = I0J0_indices[i0j0_index]
  For (i1: int, 0, 2):
    For (j1: int, 0, 2):
```

BW: $I0J0^2_4I1_2J1_2$

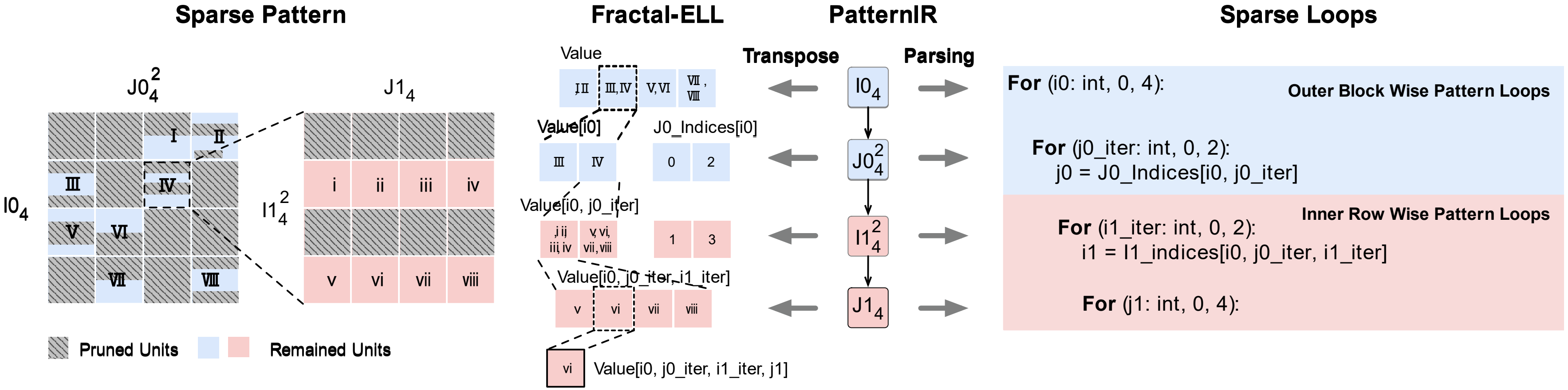


```
For (i: int, 0, 4):
  For (j0_index: int, 0, 1):
    j0 = J0_indices[j0_index]
    For (j1_index: int, 0, 1):
      j1 = J1_indices[j1_index]
```

TW+VW: $I_4J0^1_2J1^1_2$

Explore novel hybrid patterns!

PatternIR Parsing

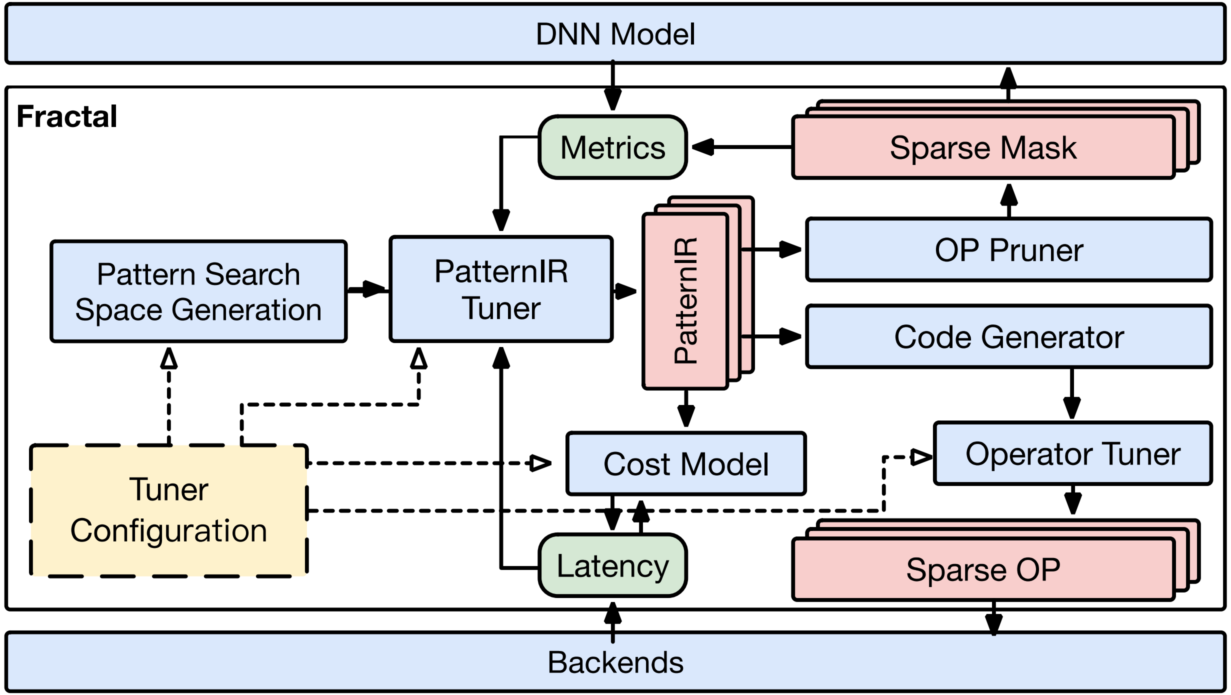


Fractal-ELL: store values and indices given the loop in PatternIR

Sparse Loops: loop up indices associated with the loop

Contents

System Overview



1. Pattern Space Generation

2. Accuracy Estimation

3. Performance Evaluation

4. Optimal pattern tuning

Code Generation

UC San Diego

Dense Operator ❶

```
A = match_buffer(a,[1024,1024],float16)
B = match_buffer(b,[1024,1024],float16)
C = match_buffer(c,[1024,1024],float16)
for i,j,k in grid(1024,1024,1024):
    C[i,j] += A[i,k] * B[k,j]
```

Pattern Search Space Generation

Tiling Schedule ❷

```
i0,i1,i2 = Split(i,4,8,32)
k0,k1,k2 = Split(k,32,8,4)
Reorder(i0,i1,k0,k1,k2,i2)
```

Perforation Schedule ❸

```
Perforate(i1,3)
Perforate(k0,28)
Perforate(k1,5)
```

PatternIR Schedule

PatternIR Parsing

❹ **Sparse Operator Program**

Sparse Axis

```
I0 = dense_axis(4)
I1 = sparse_axis(I0,(8,3),(indices0))
K0 = sparse_axis(I1,(32,28),(indices1))
K1 = sparse_axis(K0,(8,5),(indices2))
K2 = dense_axis(4)
I2 = dense_axis(32)
J  = dense_axis(1024)
```

Sparse Buffer

```
A = match_sparse_buffer(
    a,[I0,I1,K0,K1,K2,I2],float16)
B = match_buffer(b,[1024,1024],float16)
C = match_buffer(c,[1024,1024],float16)
```

Sparse Iteration

```
with sp_iter([I0,I1,K0,K1,K2,I2,J], SSRRRSS, spam) as [
    I0,i1,k0,k1,k2,i2,j
]:    C[i0*256+i1*32+i2,j] += A[i0,i1,k0,k1,k2,i2] *
                              B[k0*32+k1*4+k2,j]
```

Sparse OP Lowering

Loop-based Tensor Program ❺

```
for i0,i1,k0,k1,k2,i2,j in grid(4,3,28,5,4,32,1024):
    C[i0*256+indices0[i0*3+i1]+i2,j] +=
    A[i0*53760+i1*17920+k0*640+k1*128+k2*32+i2] *
    B[indices1[i0*84+i1*28+k0]*32+\
      indices2[i0*420+i1*140+k0*5+k1]*4+k2,j]
```

Operator Tuning

Optimization Schedule ❻

```
condense B
transform_layout A,B,C
k = fuse(k0,k1,k2)
i2_0,i2_1 = split(i2,16)
j_0 , j_1 = split( j,16)
k_0 , k_1 = split( k,16)
reorder(i2_0,j_0,k_0,i2_1,j_1,k_1)
tensorcore_blk = blockize(i2_1,j_1,k_1)
```

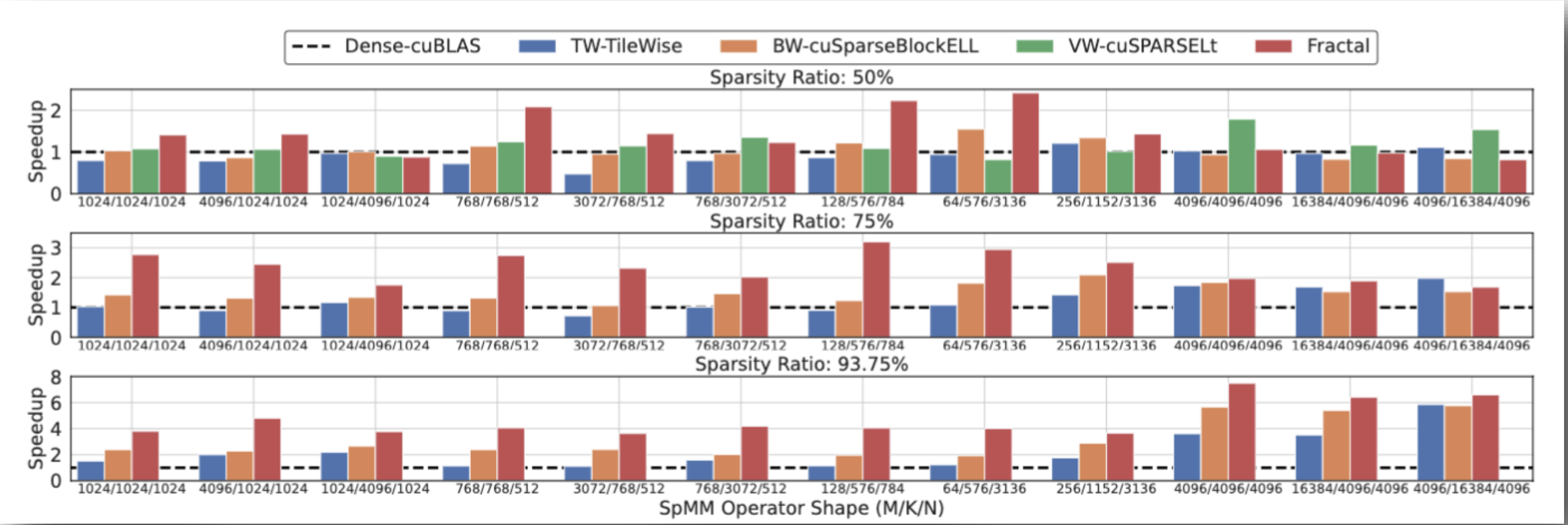
Tune efficient sparse pattern with the loop-based pattern and underlying operator compiler.

Contents

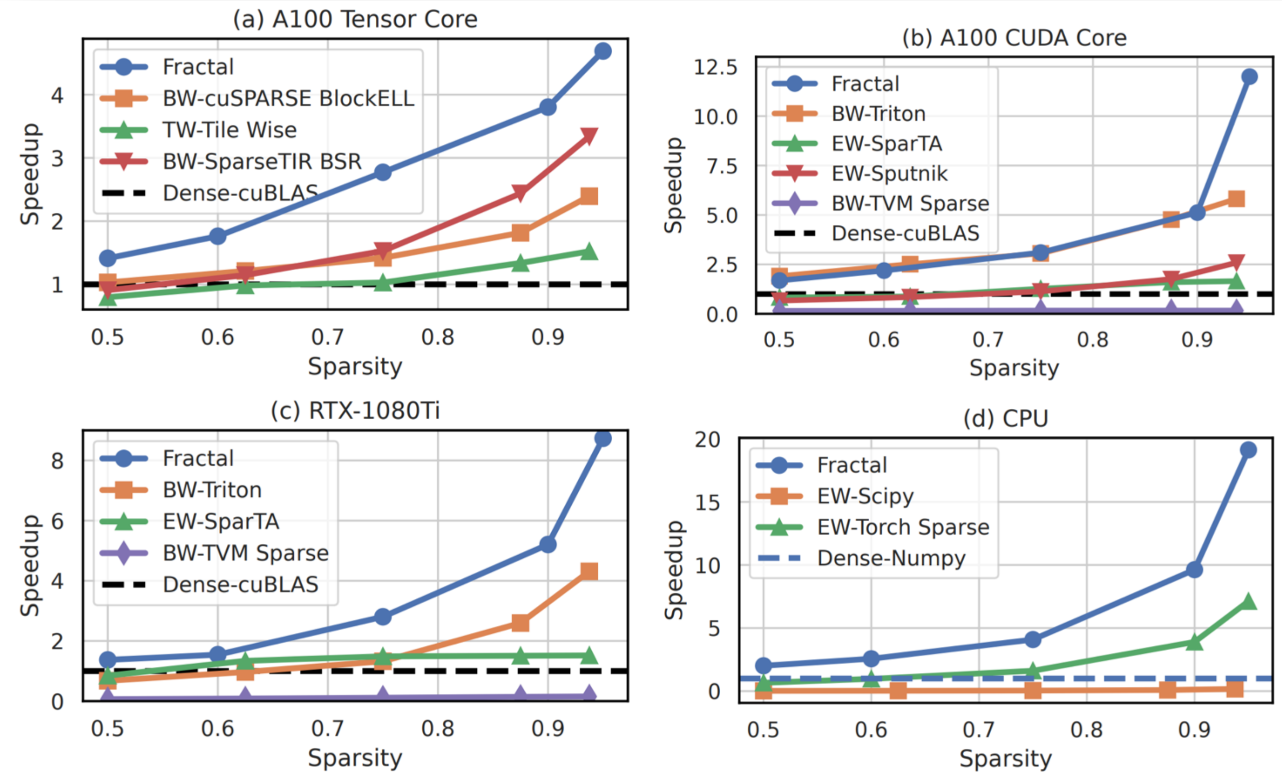
Operator Result

UC San Diego

Latency speedup improvement on different shapes and sparsity.



Backend Results

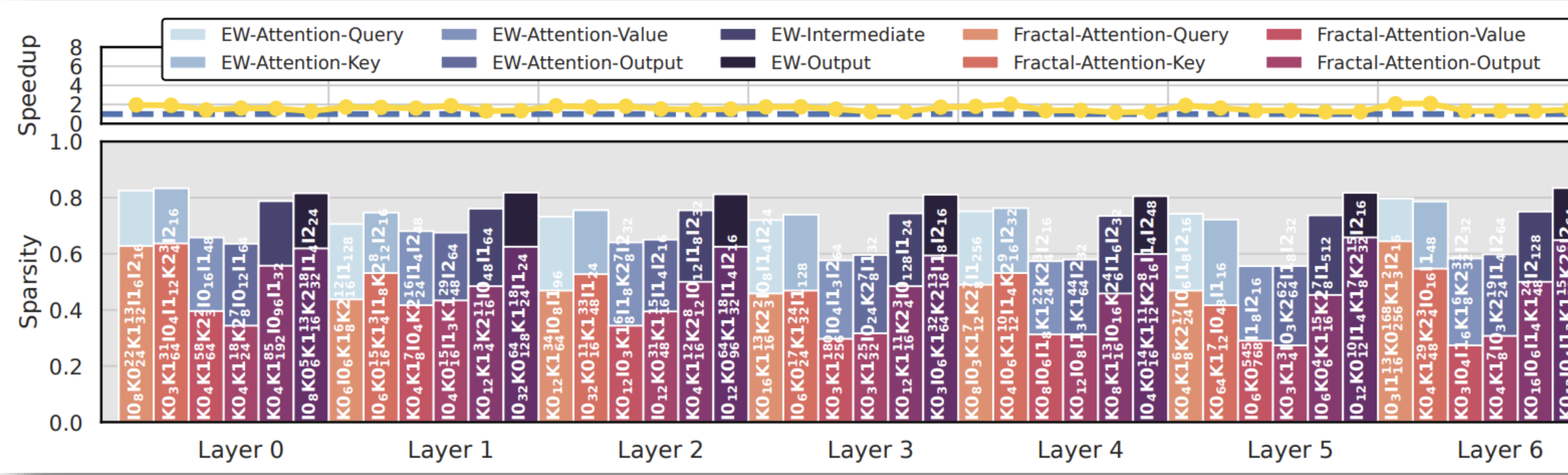


Supports different backends and achieve consistent speedup.

Model Results

UC San Diego

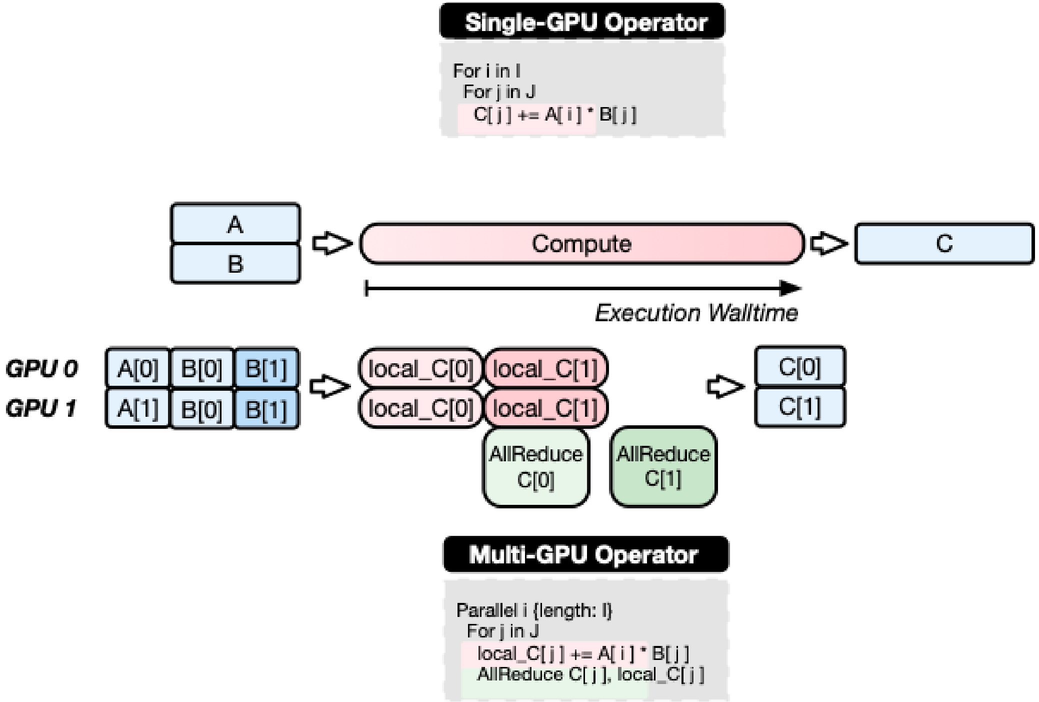
Find optimal sparse pattern and kernel for each operator.

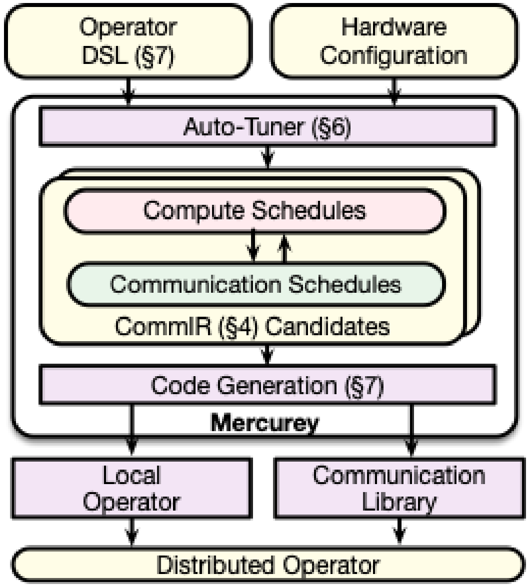


Contents

Multi-GPU Operators

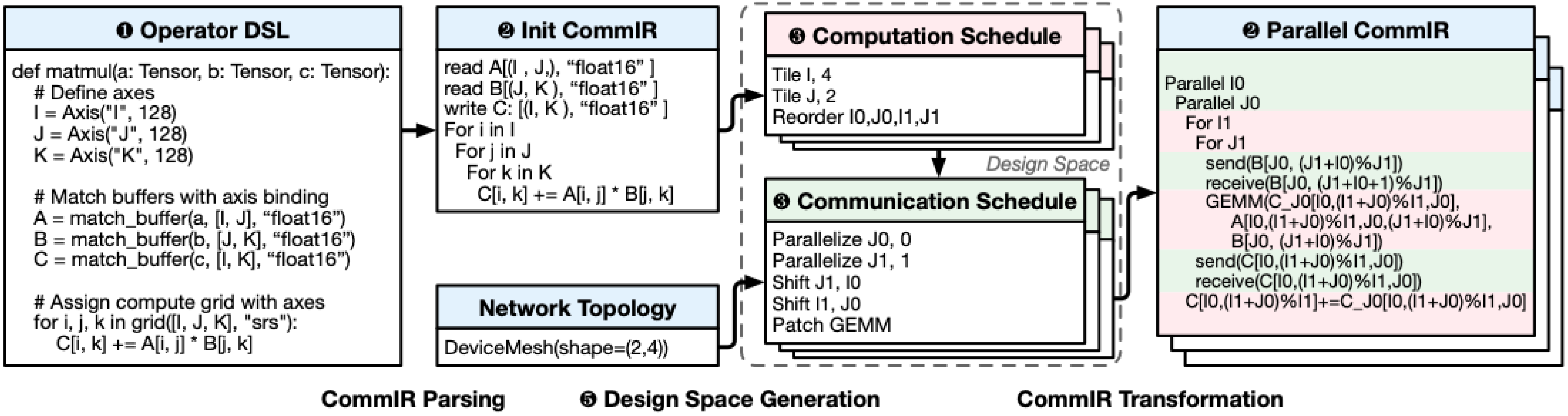
Using distributed GPUs to accelerate computation





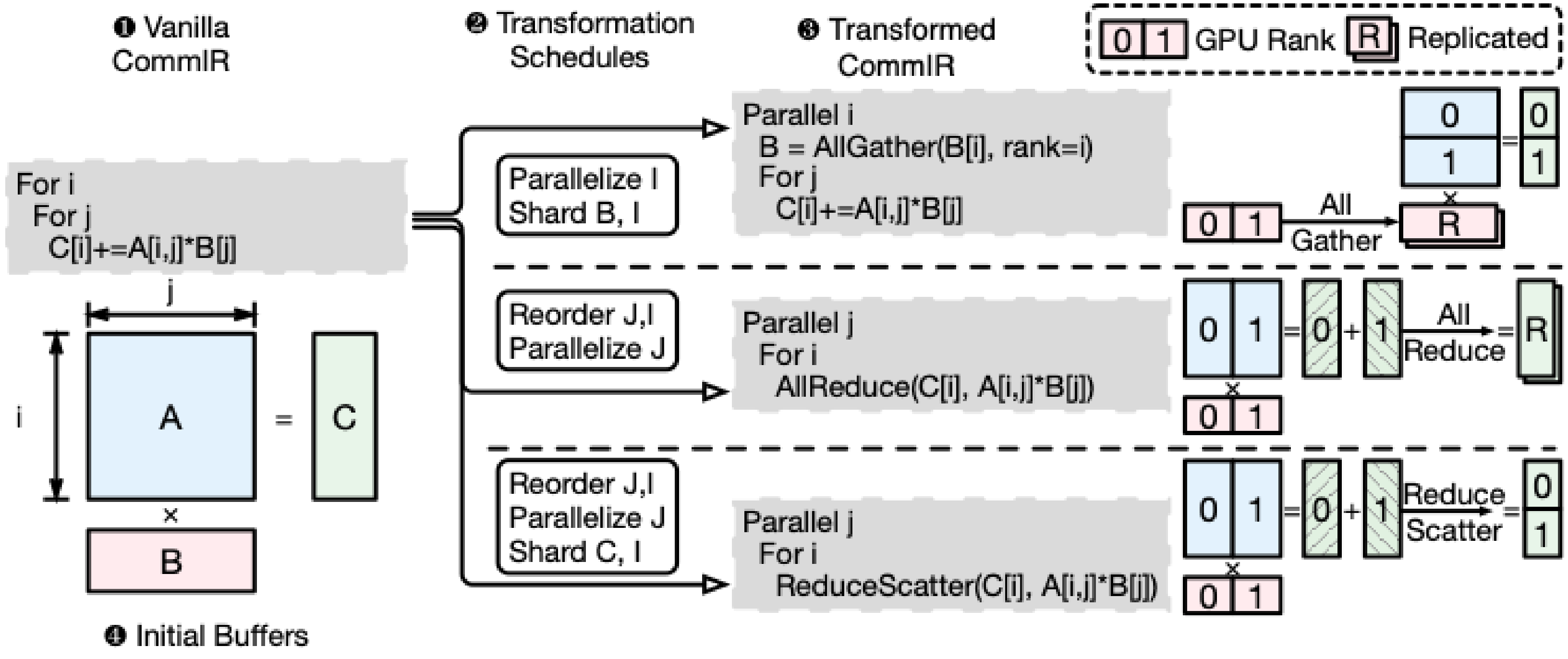
CommIR for Communication

Introducing the communication schedule to represent the data transfer between GPUs.



Communication Patterns

UC San Diego



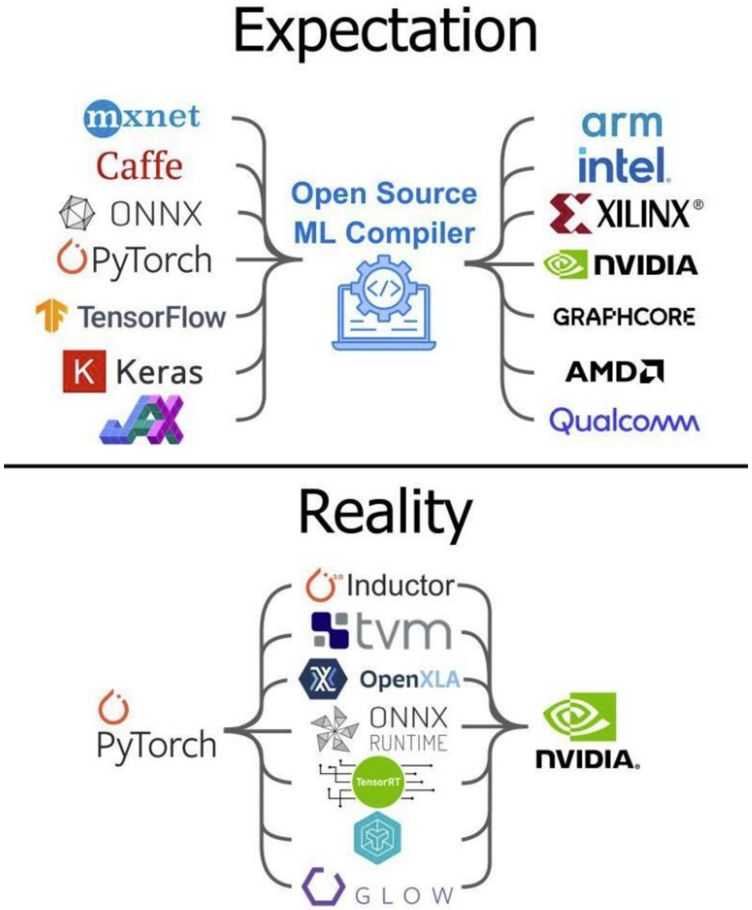
Contents

Summary

UC San Diego

Compilers	Halide	Tensor Comprehension	TVM	Fractal	Mercury
Domain	Image processing	ML	ML	Sparse ML	ML
Hardware	CPU/GPU	GPU	GPU/Accelerator	GPU/Accelerator	Multi-GPU
Scheduling Abstraction	Explicit	Annotations	Explicit	Explicit sparse pattern	Joint communication and computation
Parallelism	parallel, vectorize, tile	Implicit	parallel, bind	parallel, bind	parallelize, shard, replicate, shift
Memory Access	Explicit	Limited	Local/Global	Local/Global	Collective and asynchronous communications
Autotuning	Cost-model guided beam-search	Genetic search	Template-free	Enumeration-based	Enumeration-based
Comments	First scheduling language	Early ML support	Extending to accelerators	Extending to sparse ML	Expanding to multi-GPU

Summary



Still a long way to go.