Sparse LLM Inference through Algorithm & Compiler Co-Design

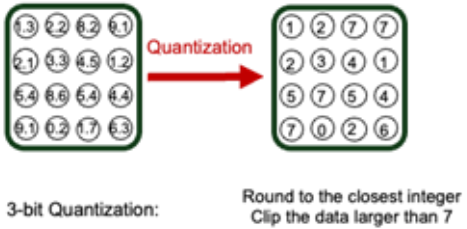
PATCH+STOICC

Maryam Mehri Dehnavi
University of Toronto & NVIDIA Research

Collaborators: Victor Kamel, Arya Rafii, Mohammad Mozaffari, Younes Hourri

Compression Methods

Quantization: Reduce the precision of numbers



Sparsity: Set non-important weights to zero

- **2:4 sparsity**: at most 2 out of 4 consecutive elements are nonzeros: Sparse Tensor cores
- **Unstructured sparsity**: nonzeros can exist anywhere in the matrix: No hardware support



Higher Compression Rates

LLaMA-2-7B Average Accuracy on 8 Zero-shot Tasks[1]

Compression Rate	Method	Average Accuracy
	Dense	54.61%
4x ←	4-bit Quantization	53.63%
8x ←	2-bit Quantization[2]	31.81%

Beyond 4-bit quantization leads to significant loss in model quality

[1] The tasks include MMLU, PIQA, ARC-Easy, ARC-Challenge, WINOGRANDE, and OpenBookQA, RACE, Hellaswag
[2] Best method among AbsMax and OPTQ

Higher Compression Rates

LLaMA-2-7B Average Accuracy on 8 Zero-shot Tasks[1]

Compression Rate	Method	Average Accuracy
	Dense	54.61%
4x	4-bit Quantization	53.63%
	2-bit Quantization[2]	31.81%
8x	4-bit Quantization + 2:4 Sparsity[3]	47.97%

Combining sparsity and quantization gives better accuracy vs. quantization or sparsity alone!

[2] Best method among AbsMax and OPTQ
[3] Best method among MaskLLM, Wanda, SparseGPT

Higher Compression Rates

LLaMA-2-7B Average Accuracy on 8 Zero-shot Tasks[1]

Compression Rate	Method	Average Accuracy
	Dense	54.61%
4x	4-bit Quantization	53.63%
	2-bit Quantization[2]	31.81%
	4-bit Quantization + 2:4 Sparsity[3]	47.97%
8x	4-bit Quantization + Unstructured Sparsity[4]	52.38%

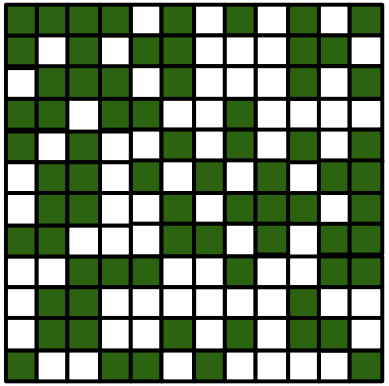
Quantization with **unstructured sparsity** leads to good model accuracy and 8x compression!

[2] Best method among AbsMax and OPTQ
[3] Best method among MaskLLM, Wanda, SparseGPT
[4] Best method among Wanda and SparseGPT

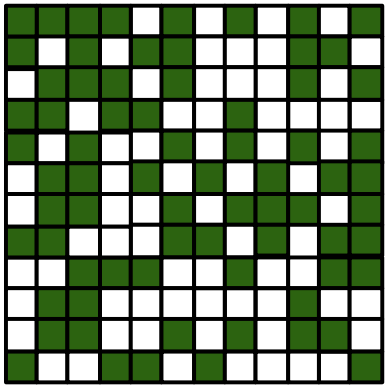
Unstructured Sparsity

Pros: Preserves model quality

Cons: No hardware support to accelerate



Unstructured Sparsity



Pros: Preserves model quality

Cons: No hardware support to accelerate

Current software solutions

- Don’t align with real AI workloads
- Tool builders chase generality; AI practitioners assume support that isn’t real

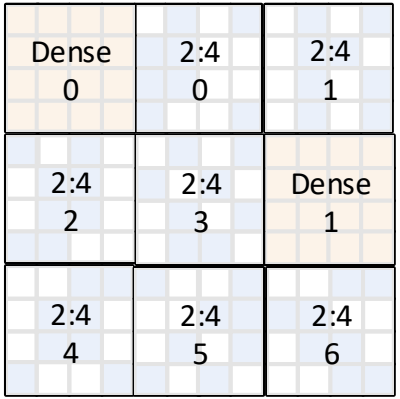
Tile-based (Unstructured) Sparsity

Flexibility to choose sparse vs. dense tiles

Flexibility to adjust nonzero placement in sparse tiles

Preserves model quality ⇒ **PATCH**

Can be accelerated on GPUs ⇒ **STOICC**



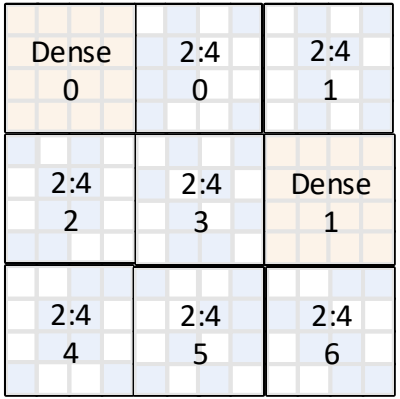
Tile-based (Unstructured) Sparsity

Flexibility to choose sparse vs. dense tiles

Flexibility to adjust nonzero placement in sparse tiles

Preserves model quality → **PATCH**

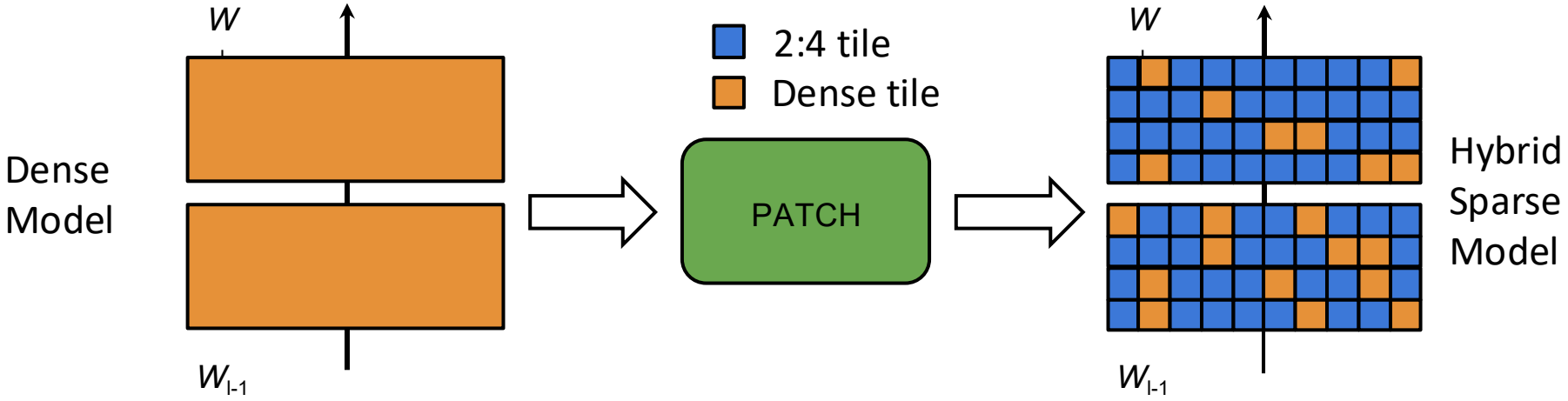
Can be accelerated on GPUs → STOICC



PATCH

Hybrid approach

- Combines the flexibility of unstructured with the efficiency of 2:4
- Jointly learns a mask that selects dense vs. sparse tiles, and the 2:4 pattern within sparse tiles



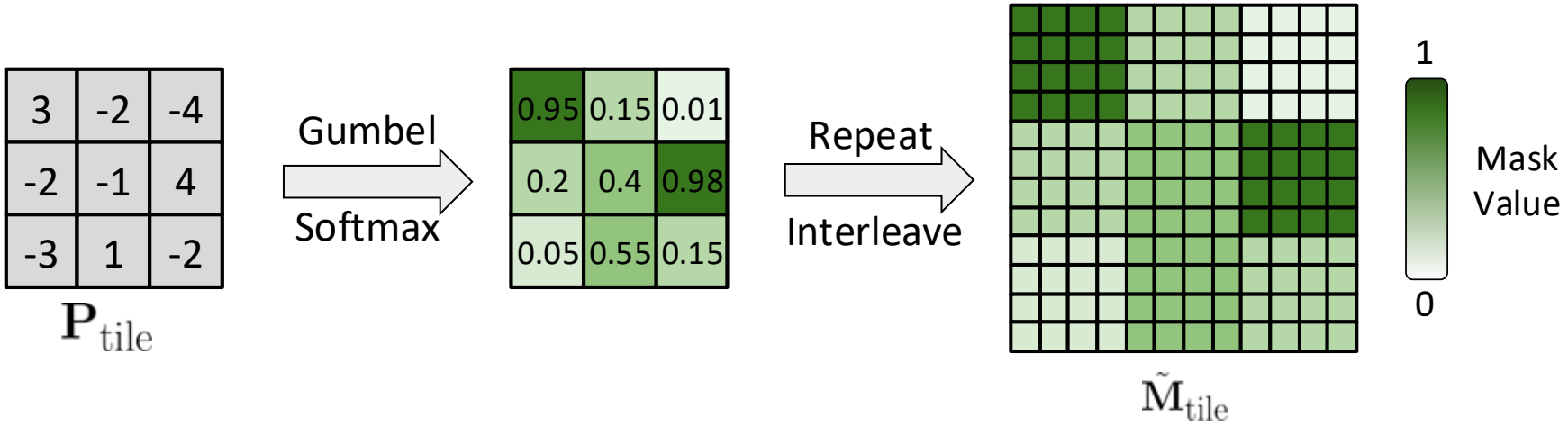
PATCH | Tile Selection

$\mathbf{P}_{\text{tile}}$: Tile logits
$\tilde{\mathbf{M}}_{\text{tile}}$: Sampled Tile Mask

We frame tile selection as a probabilistic sampling problem:

Using Gumbel-Softmax (GS), we turn logits into probabilities and sample sparsity patterns

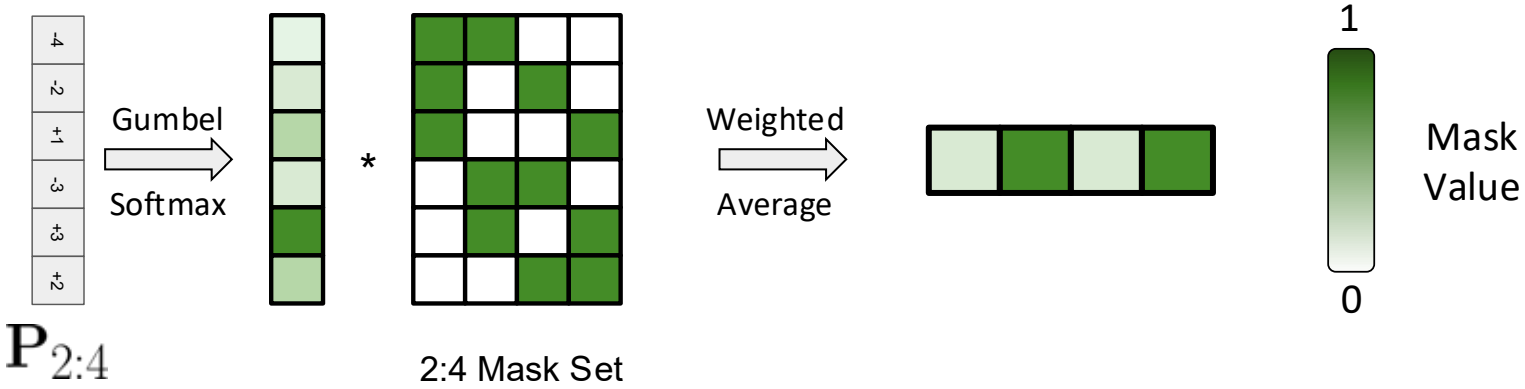
- The randomness will allow us to explore the space of mask combinations



PATCH | Learnable 2:4 Pattern

$\mathbf{P}_{2:4}$: 2:4 logits

For every 4 consecutive weights, it learns 6 logit parameters corresponding to the 6 valid 2:4 masks



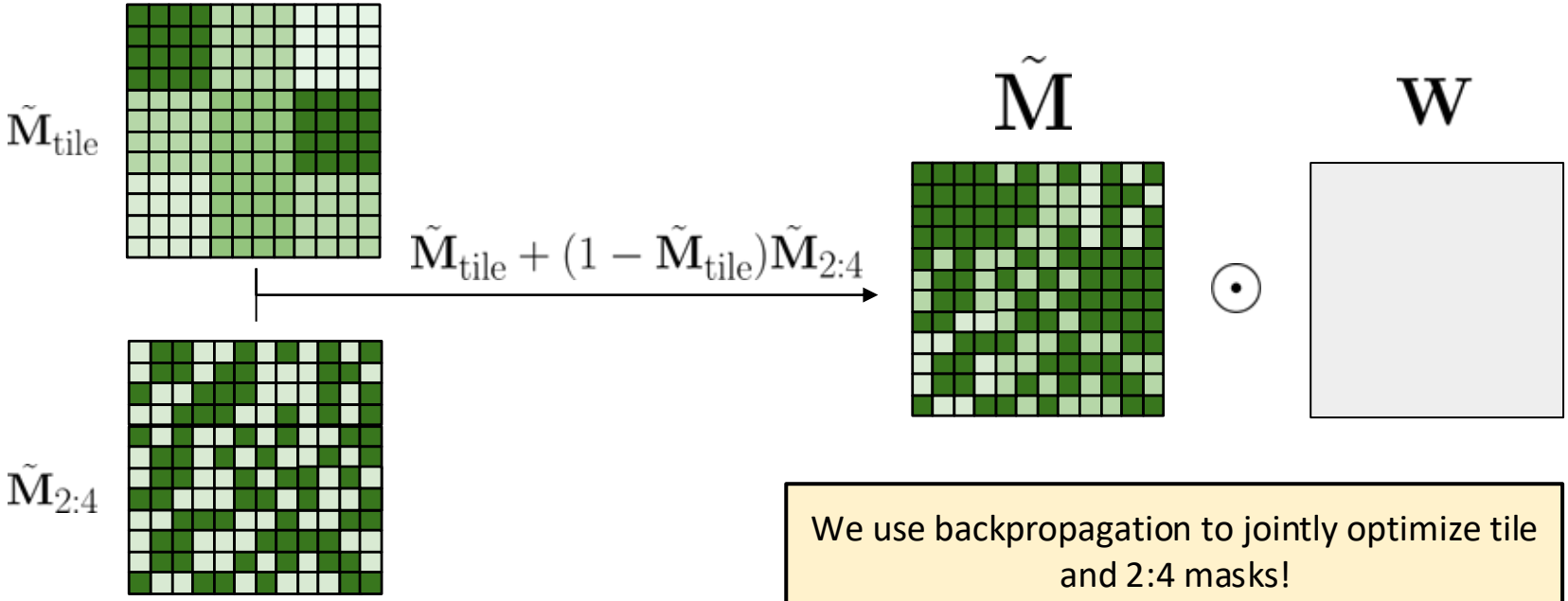
$\mathbf{P}_{2:4}$

PATCH | Combining Masks

$\tilde{M}_{2:4}$: Sampled 2:4 mask
$\tilde{M}_{tile}$: Sampled Tile Mask
W : Weight

Tile-level mask is combined with the 2:4 mask to define the effective sparsity pattern

Each tile is a weighted average between dense and 2:4 sparse, controlled by



We use backpropagation to jointly optimize tile and 2:4 masks!

PATCH | Training Objective

ρ : Target Density
$\tilde{\mathbf{M}}$: Sampled Mask
$\mathbf{W}$: Weight

$$\mathcal{L} = \underbrace{\mathcal{L}_{LM}\left(x; \tilde{\mathbf{M}}_i \odot \mathbf{W}_i\right)}_{\text{Language Modelling Loss}} + \underbrace{\lambda_1 \left\| \frac{\sum_i \tilde{\mathbf{M}}_i}{\sum_i \|\mathbf{W}_i\|_0} - \rho \right\|_1}_{\text{Sparsity Regularizer}} - \underbrace{\lambda_2 \frac{\sum_i \|\tilde{\mathbf{M}}_i \odot \mathbf{W}_i\|_2^2}{\sum_i \|\mathbf{W}_i\|_2^2}}_{\text{Weight Regularizer}}$$

(Increase Model Quality)

(Fix Model Sparsity Ratio)

(Improve Training Stability)

PATCH | Sparsity Regularizer

ρ : Target Density
$\tilde{\mathbf{M}}$: Sampled Mask
$\mathbf{W}$: Weight

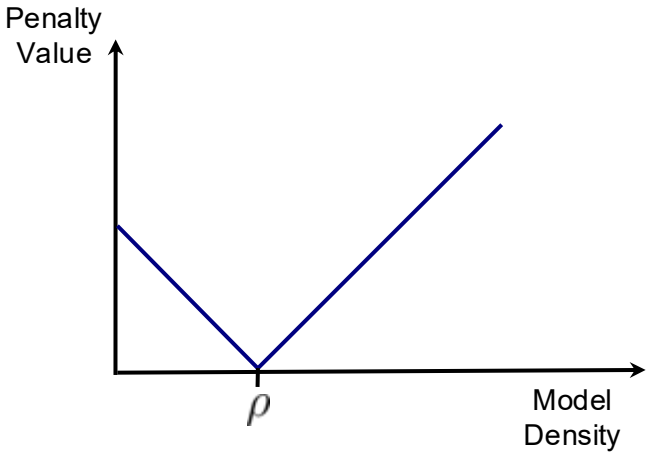
Previous sparsity regularizers cannot control the exact target sparsity ratio.

Our proposed regularizer can fix the model sparsity to a specific target!

$$\left\| \frac{\sum_i \tilde{\mathbf{M}}_i}{\sum_i \|\mathbf{W}_i\|_0} - \rho \right\|_1$$

Current Density Ratio

Target Density Ratio



PATCH | Weight Regularizer

$\tilde{\mathbf{M}}$: Sampled Mask
$\mathbf{W}$: Weight

Pruning can cause gradients to vanish, since zeroed weights diminish gradient[1]

Promote large magnitude weights to preserve gradient propagation

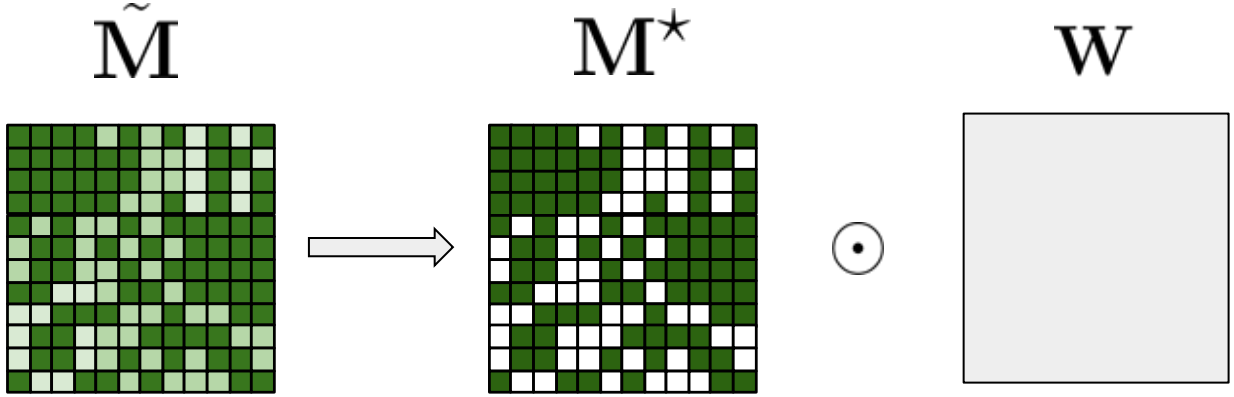
$$\frac{\sum_i \|\tilde{\mathbf{M}}_i \odot \mathbf{W}_i\|_2^2}{\sum_i \|\mathbf{W}_i\|_2^2}$$

1 : Fang, G., Yin, H., Muralidharan, S., Heinrich, G., Pool, J., Kautz, J., ... & Wang, X. (2024). Maskllm: Learnable semi-structured sparsity for large language models. *Advances in Neural Information Processing Systems*, *37*, 7736-7758.

PATCH | Inference

$\tilde{\mathbf{M}}$: Sampled Mask
$\mathbf{M}^{\star}$: Final Mask
$\mathbf{W}$: Weight

At inference, we select the most probable option (highest logit) to form the final tiled and 2:4 mask



PATCH | Accuracy

LLaMA-2-7B Average Accuracy on 8 Zero-shot Tasks

Accuracy Improvement	Method	Average Accuracy	SliM LoRA*	
	Dense	54.61%	-	
	4-bit Quantization	53.63%	-	
	2-bit Quantization	31.81%	-	
	4-bit Quantization + 2:4 Sparsity	47.97%	-	
0.22% ←	4-bit Quantization + PATCH 45%	48.19%	50.71%	→ 2.74%
1.41% ←	4-bit Quantization + PATCH 35%	49.38%	51.91%	→ 3.94%
2.48% ←	4-bit Quantization + PATCH 25%	50.45%	52.62%	→ 4.65%

- SLiM: One-shot Quantized Sparse Plus Low-rank Approximation of LLMs, ICML'25

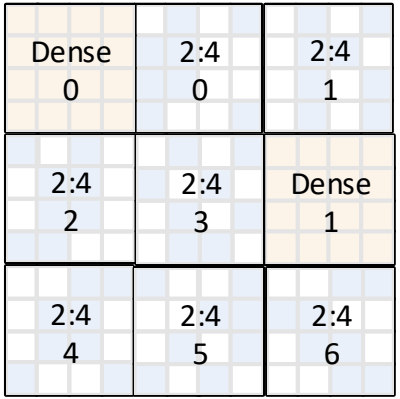
Tile-based (Unstructured) Sparsity

Flexibility to choose sparse vs. dense tiles

Flexibility to adjust nonzero placement in sparse tiles

Preserves model quality → PATCH

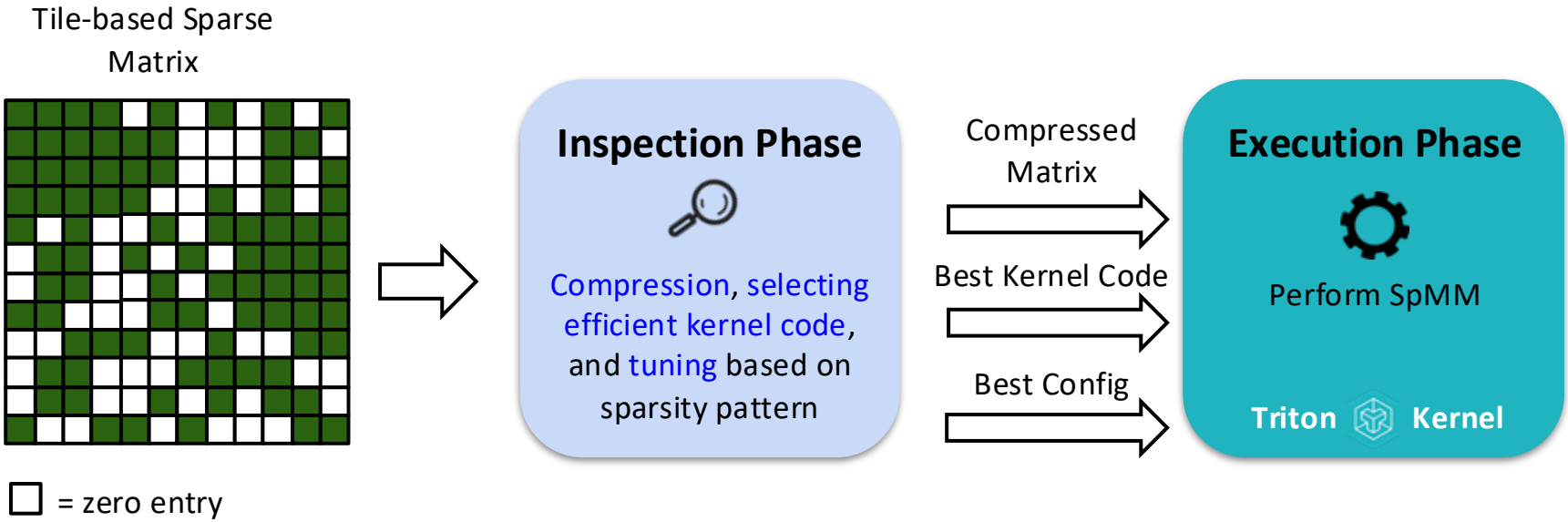
Can be accelerated on GPUs → **STOICC**



STOICC | Sparse Tile MOsaIC Compiler

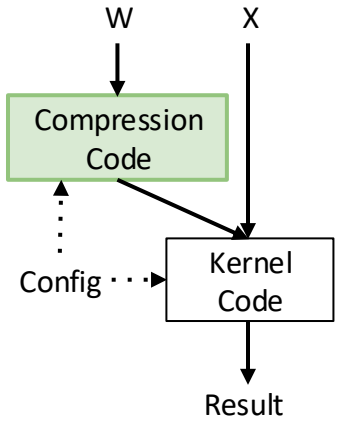
The STOICC compiler accelerates SpMM with a tile-based sparse matrix and a dense matrix.

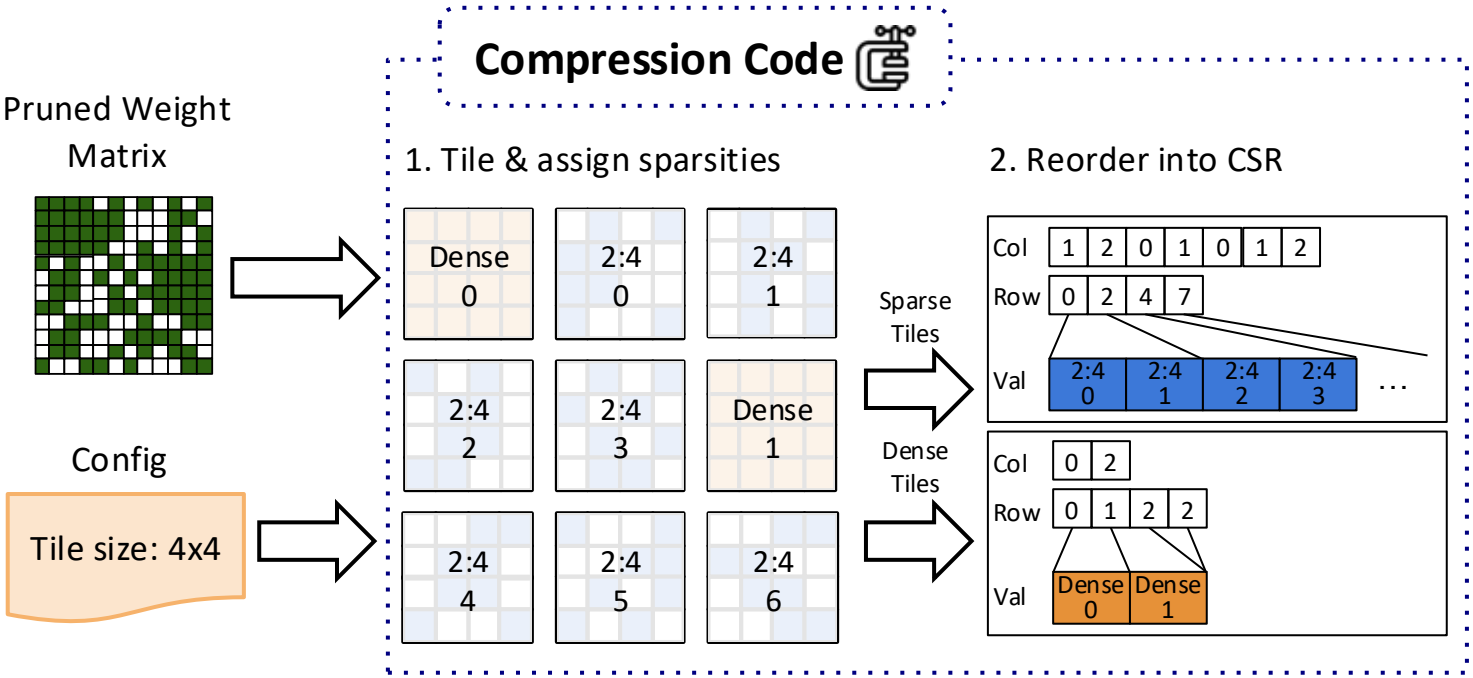
STOICC consists of an *offline* **inspection phase** and an *online* **execution phase**.



STOICC | Inspection: Compressor

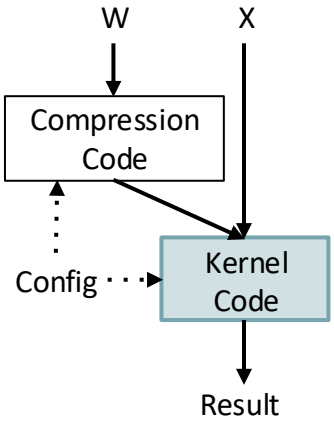
The compressor assigns sparse formats to matrix tiles and reorders data.

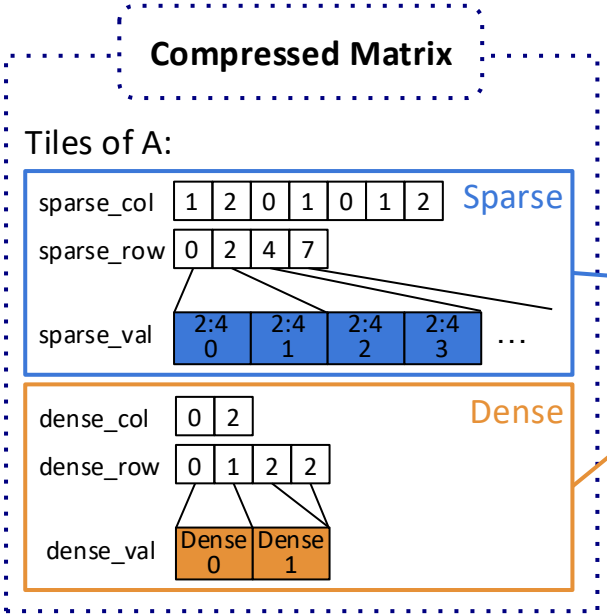




STOICC | Kernel Code

STOICC enables the implementation of different kernels.



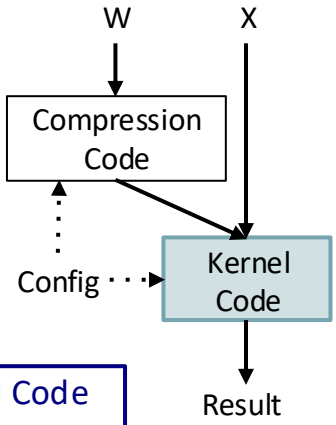


Kernel code

```
for row, col in grid:
    acc_tile = 0
    for i in range(sparse_row[row], sparse_row[row + 1]):
        acc_tile += dot(sparse_val[i], B[sparse_col[i]][col])
    for i in range(dense_row[row], dense_row[row + 1]):
        acc_tile += dot(dense_val[i], B[dense_col[i]][col])
    D[row, col] = acc_tile
```

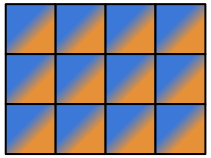
Launch grid with one Triton program per tile of output matrix D.
Each program iterates over a **row of A** and **column of B** according to the schedule.

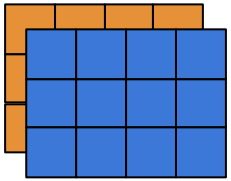
STOICC | Kernel Code



STOICC allows the user to express different schedules.

Launch Grid





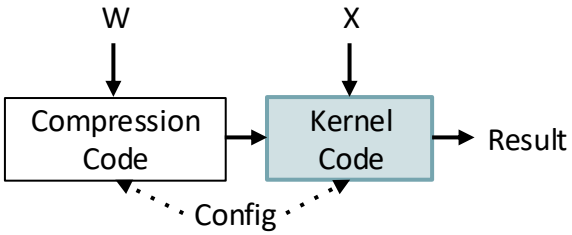
Sequential Kernel Code

```
for row, col in grid:
  acc_tile  = do_sparse_loop(sparse_*, B, row, col, tile_sizes)
  acc_tile += do_dense_loop(dense_*, B, row, col, tile_sizes)
  D[row, col] = acc_tile
```

Concurrent Kernel Code

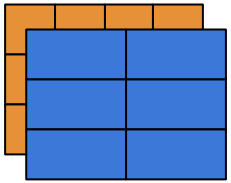
```
D = 0
for row, col, sparsity in grid:
  if sparsity is 1: acc_tile = do_sparse_loop(sparse_*, B, row, col, tile_sizes)
  if sparsity is 0: acc_tile = do_dense_loop(dense_*, B, row, col, tile_sizes)
  D[row, col].atomic_add(acc_tile)
```

STOICC | Kernel Code



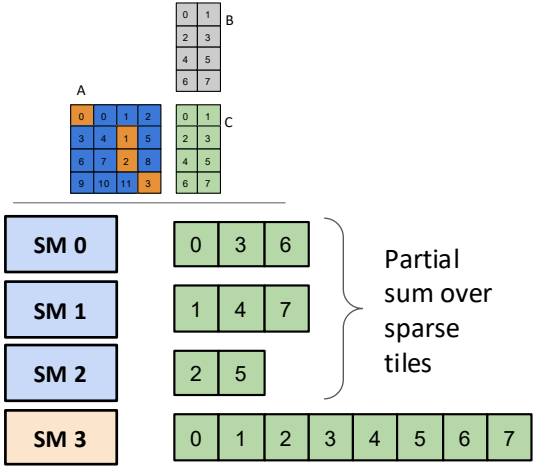
STOICC allows the user to express different schedules.

Launch Grid



Variable Tile Size Kernel

```
D = 0
for row, col, sparsity in grid:
  if sparsity is 1:
      D_sparse_view[row, col].atomic_add(do_sparse_loop(sparse_*, B, row, col, sparse_tile_sizes))
  if sparsity is 0:
      D_dense_view[row, col].atomic_add(do_dense_loop(dense_*, B, row, col, dense_tile_sizes))
```



Sparsity Specialized Kernel

```
output_tile = pid
if pid < NUM_SPARSE_SMS:
  while output_tile < num_tiles:
    acc_tile += do_sparse_loop(sparse_*, B, row, col, tile_sizes)
    atomic_add (output_tile, acc_tile)
    output_tile += NUM_SPARSE_SMS
else:
  while output_tile < num_tiles:
    acc_tile += do_dense_loop(dense_*, B, row, col, tile_sizes)
    atomic_add (output_tile, acc_tile)
    output_tile += NUM_DENSE_SMS
```

STOICC | Sparse Tile Lowering

Sparse tensors are denoted with sparse encodings in IR.

```
@triton.jit
def dot_kernel(a_sparse_ptr: tl.sparse_tensor,
               a_dense_ptr: tl.tensor, b_ptr, d_ptr, schedule ...):
    pid_row = tl.program_id(0)
    pid_col = tl.program_id(1)

    b_ptrs = tl.make_block_ptr(...)  # Tile of B
    acc = tl.zeros((BLOCK_M, BLOCK_N), tl.float32)

    # Sparse Loop
    a_sparse_ptrs = tl.make_block_ptr(...) # Sparse Tile of A
    for i in range(num_sparse_tiles_in_row):
        acc = tl.dot(tl.load(a_sparse_ptrs), tl.load(b_ptrs), acc)

        a_sparse_ptrs = tl.advance(a_sparse_ptrs, [0, BLOCK_K])
        b_ptrs = advance(b_ptrs, schedule)

    # Dense Loop
    a_dense_ptrs = tl.make_block_ptr(...) # Dense Tile of A
    for i in range(num_dense_tiles_in_row):
        acc = tl.dot(tl.load(a_dense_ptrs), tl.load(b_ptrs), acc)

        a_dense_ptrs = tl.advance(a_dense_ptrs, [0, BLOCK_K])
        b_ptrs = advance(b_ptrs, schedule)

    d_ptrs = tl.make_block_ptr(...) # Tile of D
    tl.store(d_ptrs, acc.to(tl.float16))
```

Executor

IR Gen

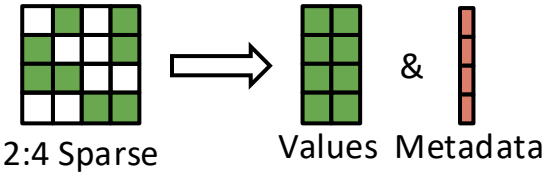
```
@dot_kernel (...) {
…
// Sparse Loop
scf.for k in K {
  %A = tt.load %A_ptrs : tensor<16x64xf16, #tt.sparse<"NV24">>
  %B = tt.load %B_ptrs : tensor<64x16xf16>
  %D = tt.dot %A, %B, %D
  …
}

// Dense Loop
scf.for k in K {
  %A = tt.load %A_ptrs : tensor<16x64xf16>
  %B = tt.load %B_ptrs : tensor<64x16xf16>
  %D = tt.dot %A, %B, %D
  …
}
…
```

Executor IR

*The Triton Kernel / IR has been simplified for brevity.

STOICC | Sparse Tensor Decomposition Pass




```
scf.for k in K {
  %A = tt.load %A_ptrs : !tt.ptr<tensor<16x64xf16, #tt.sparse<"NV24">>>
  %B = tt.load %B_ptrs : !tt.ptr<tensor<64x16xf16>>
  %D = tt.dot %A, %B, %D … : tensor<16x64xf16, #tt.sparse<"NV24">> * tensor<64x16xf16>
}
```

```
scf.for k in K {
  %A = tt.load %A_ptrs : !tt.ptr<tensor<16x32xf16>> // Values
  %B = tt.load %B_ptrs : !tt.ptr<tensor<64x16xf16>>
  %E = tt.load %E_ptrs : !tt.ptr<tensor<1x64xi16>>  // Metadata
  …
  %D = tt.dot %A, %B, %D, %E, sparseIndex=0, … : tensor<16x32xf16>, tensor<1x64xi16> * tensor<64x16xf16>
}
```

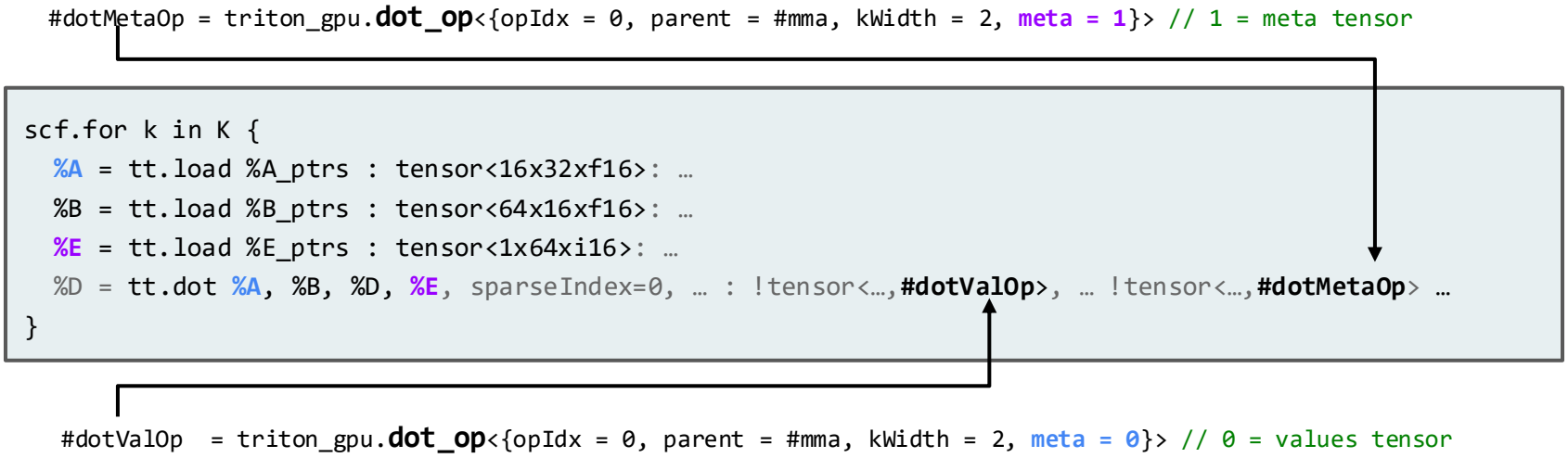
Indicates that the sparse operand is %A

This STOICC pass decomposes all sparse 2:4 tensors into dense buffers (values & metadata).

*The IR has been simplified for brevity.

STOICC | Sparse Dot Operand Encoding

The **dot_op** encoding is augmented with a **meta** field to mark whether operand is metadata.

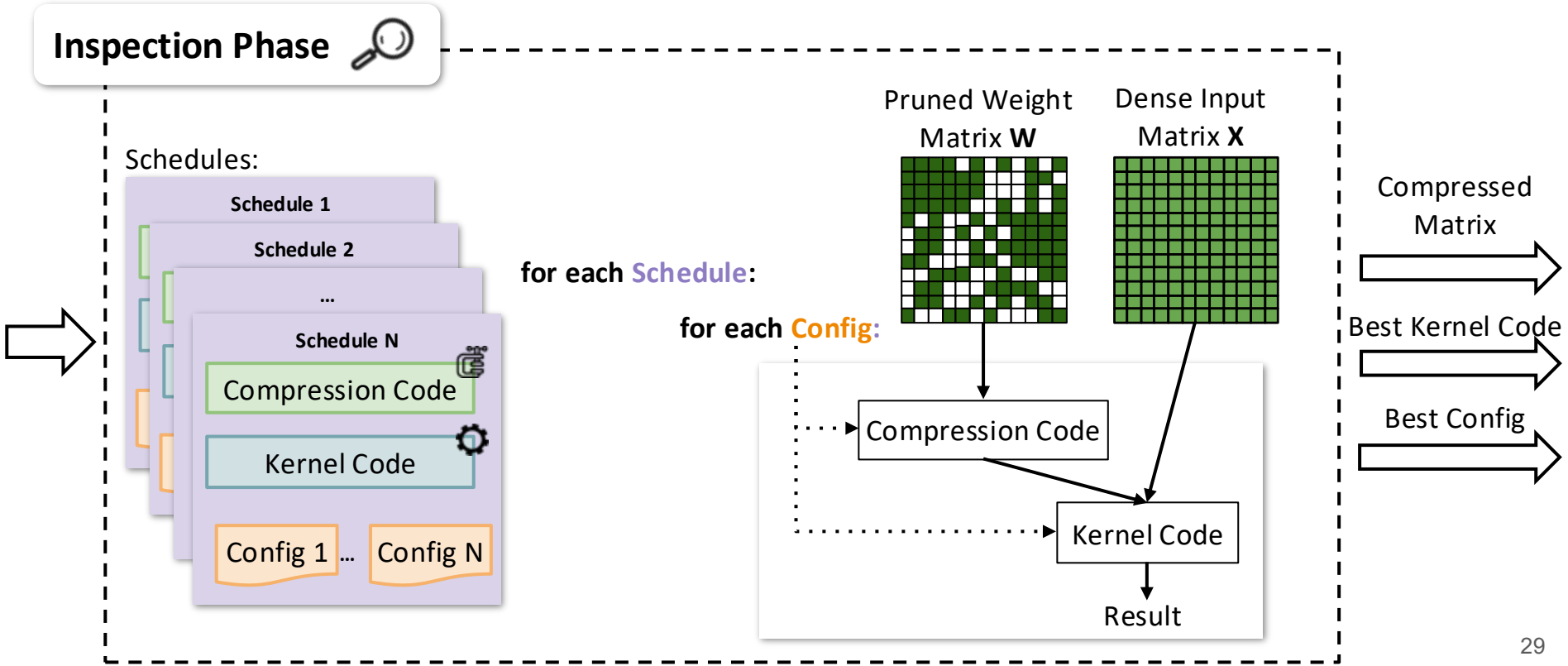


The sparse compiler uses the dot_op encoding to make decisions about how to lower operands for efficient SpMM.

*The IR has been simplified for brevity.

STOICC | Inspection Phase

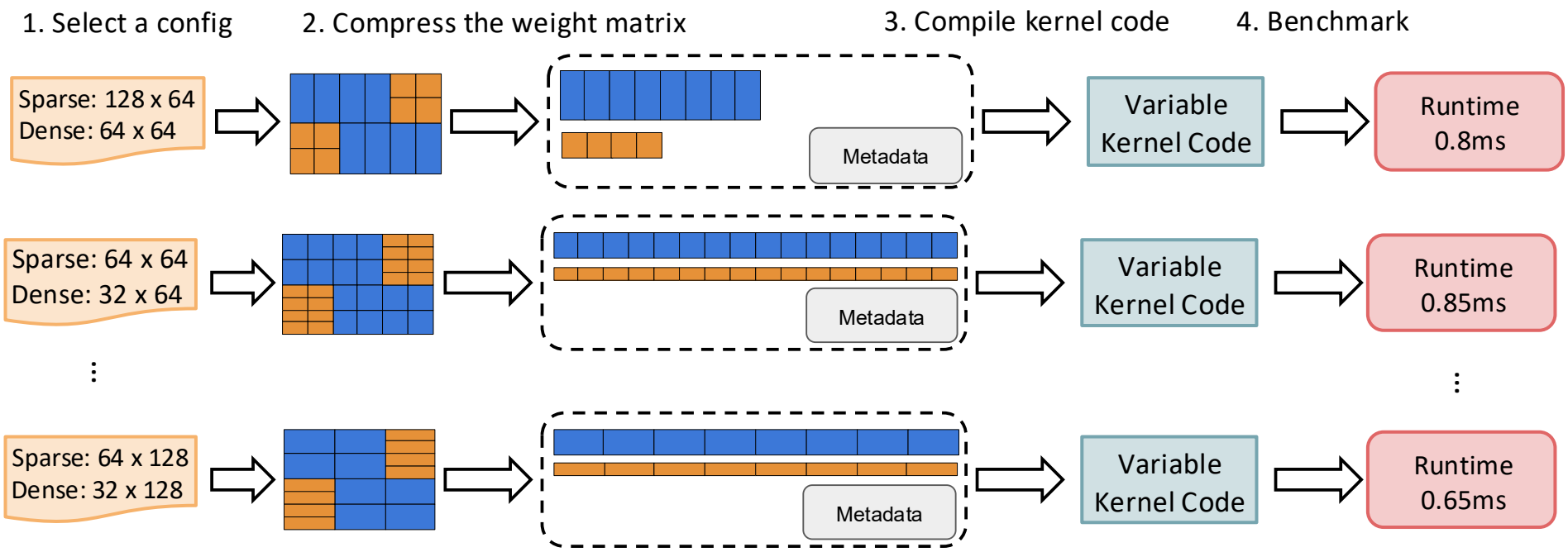
The inspection phase benchmarks different schedules.



STOICC | Kernel Tuning

Variable Schedule

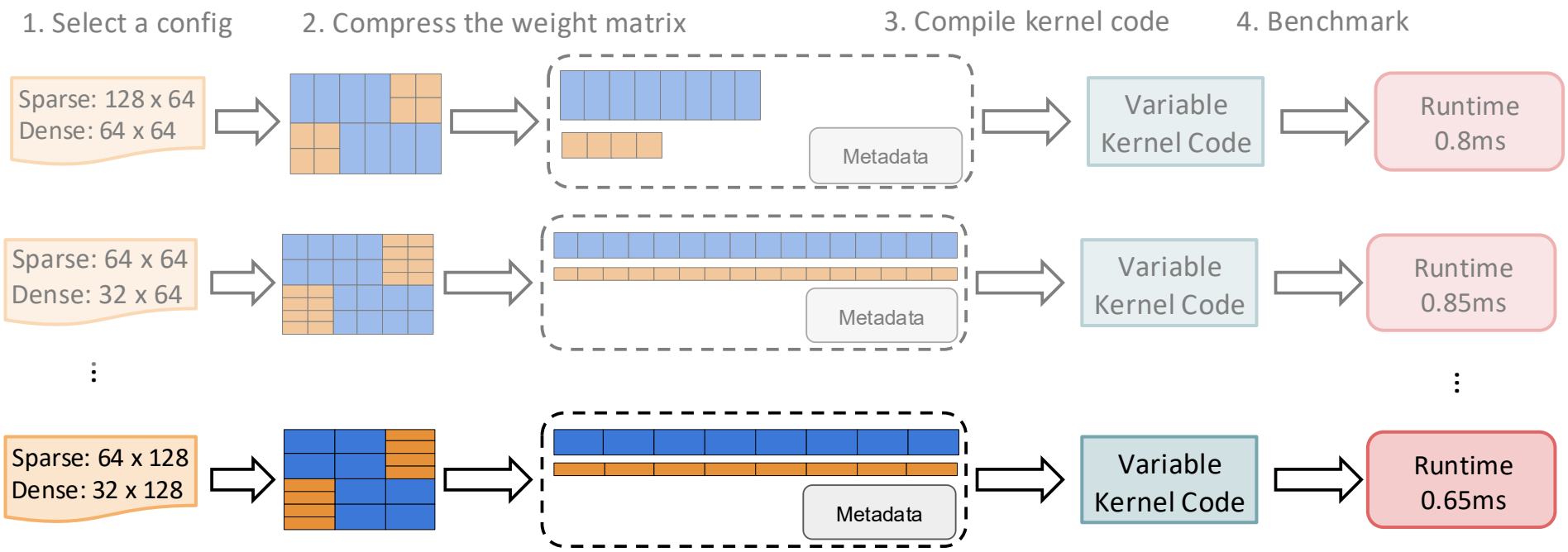
With the variable tile size schedule, configurations must now have both a sparse and a dense tile shape



STOICC | Kernel Tuning

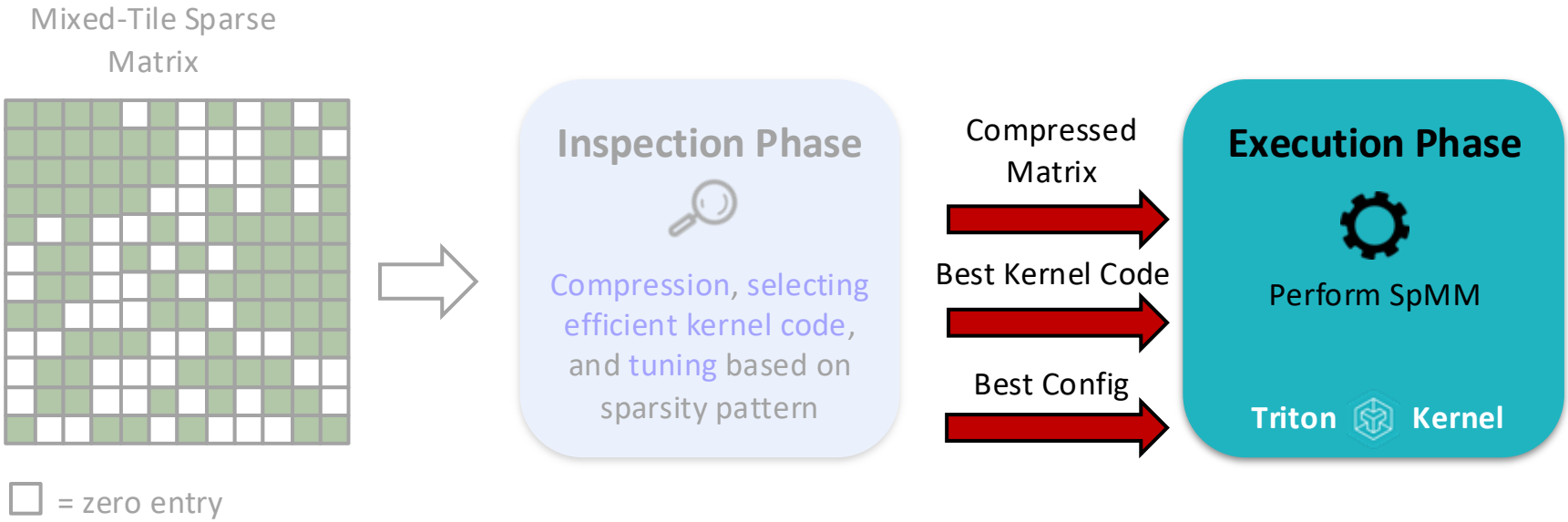
Variable Schedule

With the variable tile size schedule, configurations must now have both a sparse and a dense tile shape



STOICC | Sparse Tile MOsaIC Compiler

The best kernel code, configuration, along with the compressed weight matrix is used for inference execution.



PATCH+STOICC | PATCH/PyTorch Integration

PATCH leverages the STOICC compiler to execute mixed-tile sparsity

```
inspector = STOICC.Inspector(use_default_schedules=True)
W_compressed, best_kernel_code, best_config = inspector.inspect(X, W, sparse_arg=1)

# Replace linear module in your model Executor
module = MixedModule(W_compressed, best_kernel_code, best_config)
```

STOICC can be used out of the box with three lines of code!

PATCH+STOICC | PATCH/PyTorch Integration

PATCH leverages the STOICC compiler to execute mixed-tile sparsity

```
inspector = STOICC.Inspector(use_default_schedules=True)
W_compressed, best_kernel_code, best_config = inspector.inspect(X, W, sparse_arg=1)

# Replace linear module in your model Executor
module = MixedModule(W_compressed, best_kernel_code, best_config)
```

Replace torch matmul with a STOICC call at execution

To use with PyTorch in inference:

```
class MixedModule(torch.nn.Module):
  …
  def forward(self, x):
    # return torch.mm(X, W)
    return self.best_kernel(
      x,        # input
      self.Wc, # compressed weight
      self.best_config,
    )
```

PATCH+STOICC | Experiments



Models

- Joint Optimization
  - Qwen-2.5 0.5B
  - LLaMA-3.2 1B
  - Gemma-3 1B
- Tile-Only
  - LLaMA-2 7B
  - LLaMA-3.1 8B



Data

- SlimPajama
- 2B tokens
  - 2000 steps
  - 256 global batch size
  - 4096 sequence length



Tile Size

- 128x128

PATCH+STOICC | Experiments

PATCH achieves a controllable trade-off between sparsity and accuracy.

Average Accuracy on 8 Zero-shot Tasks*

Sparsity	Method	Pattern	Qwen-2.5 0.5B	LLaMA-3.2 1B	Gemma-3 1B
0%	Dense	-	46.00	47.70	47.01
50%	MaskLLM	2:4	39.33	41.04	41.84
45%	PATCH	Hybrid	40.29	42.08	42.80
35%	PATCH	Hybrid	41.15	42.72	43.30
25%	PATCH	Hybrid	42.39	43.81	44.07

Average Improvement Across Models

1.00%

1.65%

2.69%

*The tasks include MMLU, PIQA, ARC-Easy, ARC-Challenge, WINOGRANDE, and OpenBookQA, RACE, Hellaswag

PATCH+STOICC | Speedup and Memory Savings

Timed end-to-end generation using CUDA Graph capture

- Model: LLaMA-2 7B
- Batch Size: 16

Sparsity Target (%)	Speedup vs Dense (A6000)	Speedup vs Dense (A100)	Memory Reduction vs Dense
0	1x*	1x*	1x
25	1.18x	1.07x	0.76x
35	1.27x	1.11x	0.68x
45	1.38x	1.16x	0.59x

*cuBLAS 12.4.5

Whats next!

New sparse recipes: More tile-based unstructured patterns, pre- and post-training.

STOICC extensions: Support for new tile formats (2:4, CSR, COO) and Hopper/Blackwell.

Compiler updates: New Triton passes for low-level sparse optimizations.

More schedules: Expanding default and fallback schedules in STOICC.

Integration: Better PyTorch hooks, PATCH tests, and cuTACO support.



PATCH : github.com/Paramathic/patch
STOICC : github.com/Paramathic/stoicc



Paper (Neurips26) : PATCH