Uncovering Nested Data Parallelism and Data Reuse with FractalTensor

Microsoft Research

Tensor Operator: The Dilemma of Expressiveness and Effectiveness

The Imperative implementation of stacked RNN

```
List<List<Vector>> xss  // input sequences
List<Matrix> ws // learnable weights
Vector I  // initial state, constant
List<List<List<Vector>>> ysss //output

for 0 ≤ j < D        // stacked depth
  for 0 ≤ k < L      // sequence length
    for 0 ≤ i < N    // batch
     if j == 0 && k == 0
        s = I
        x = xss[i][k]
     elif j > 0 && k == 0
        s = I
        x = ysss[i][j − 1][k]
     elif j == 0 && k > 0
        s = ysss[i][j][k − 1]
        x = xs[i]
     else
        s = ysss[i][j − 1][k]
        x = ysss[i][j][k − 1]
      // user-defined cell function
      yss[i][j][k] = x@ws[j] + s
```

- Strong expressiveness
    - Most intuitive way of implementation
- Less effectiveness
    - Three-level loops, many branches
    - Hard to analyze data dependency
    - Poor performance

Tensor Operator: The Dilemma of Expressiveness and Effectiveness

Three ways to define tensor operators

1

$for\ 0 \le j < D$
$\quad for\ 0 \le k < L$
$\quad\quad for\ 0 \le i < N$

RNN Cell as Op

The bottom level loop as an operator
- Most expressive
- Less effectiveness

2

$for\ 0 \le j < D$
$\quad for\ 0 \le k < L$
$\quad\quad for\ 0 \le i < N$

RNN Layer as Op

Wrap the two-level loops as an operator
- Less expressive
- More effectiveness

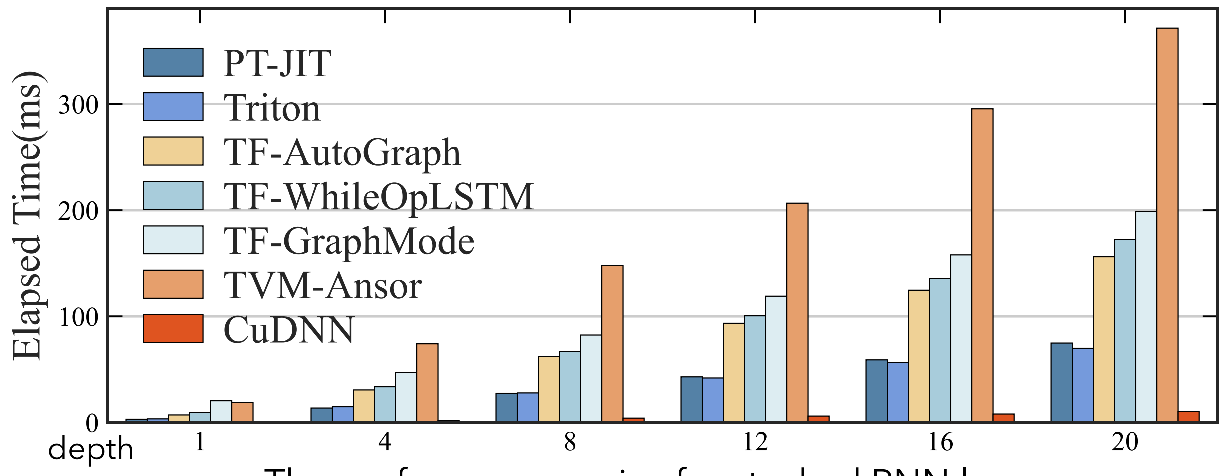
3

$for\ 0 \le j < D$
$\quad for\ 0 \le k < L$
$\quad\quad for\ 0 \le i < N$

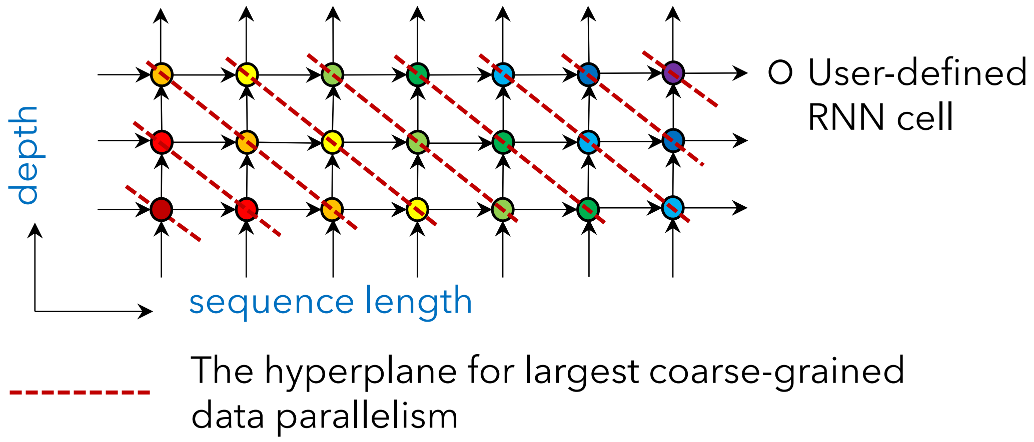
Stacked RNN Layers as Op

The whole three-level loops as an operator
- Worst expressive
- Most effectiveness

The Impact of Expressiveness on Effectiveness: Stacked RNN



The performance varies for stacked RNN layer.



As the implementations become less expressive, the performance increases, i.e., more effective

Another Example: Flash Attention

`List<List<List<Matrix>>>` $qsss, ksss, vsss$
`List<List<List<Matrix>>>` $osss$

for $0 \le i < B$ `// batch size`
  for $0 \le j < H$ `// heads`
    for $0 \le m < L_1$ `// sequence length`
      $M_t = -\overrightarrow{\inf}_{[d_2,1]}$
      $S_t = \vec{0}_{[d_2,1]}$
      $O_t = \vec{0}_{[d_2,d_4]}$
      for $0 \le n < L_2$ `// sequence length`
        $Q = qsss[i][j][m], K = ksss[i][j][n], V = vsss[i][j][n]$ `// load from DRAM`
        $T_1 = Q@K^T$
        $T_2 = max(T_1)$
        $T_3 = exp(T_1 - T_2)$
        $T_4 = T_3@V$
        $T_5 = sum(T_3, dim = -1)$
        $M_t' = \text{maximum}(T_2, M_t)$
        $T_6 = exp(M_t' - M_t)$
        $T_7 = exp(T_2 - M_t')$
        $S_t = T_6 * S_t + T_7 * T_5$
        $O_t = (O_t * S_t * T_6 + T_7 * T_4) / S_t$
        $osss[i][j][n] = O_t$ `// store to DRAM`

1. online normalization instead of compute the full batch at once

2. reusable data cached and computed in SRAM

The imperative implementation of FlashAttention

- The entire algorithm is implemented into one monolithic, opaque operator
- No room for analysis and optimization
- Minor adjustment requires re-implementation
  - FlashAttention 1/2/3
  - Sacrifice expressiveness for effectiveness

The dilemma between tensor operator's expressiveness and effectiveness: a common painpoint

Problem

The DAG of tensor operators, to achieve **effectiveness**, often exposes only single-level parallelism, lacking **expressiveness** and resulting in insufficient global analysis

Observation

- Diverse DNN computation patterns can be expressed by a combination of second-order array **compute operators** like map, reduce, scan, fold
- Data access patterns in DNN computation are highly stylized and can be expressed by a few first-order array **access operators**
- DNN algorithms can be expressed along tensor dimensions with **compute** and **access** operator nesting

It is possible to provide an expressive programming model for DNN, and generate effective, high-performance code.

FractalTensor: Decompose Tensor into Nested Lists of Tensors

- FractalTensor: a list-based ADT, an element is a static-shape tensor, or a FractalTensor
- FractalTensor: decompose dimensions of a tensor into:
  - The innermost statically-shaped dimensions
  - The **programmable dimensions**

Examples: organize data as FractalTensors

batch length

```
xss: FractalTensor[FractalTensor[Tensor['1,512'], float, 'gpu']]] = initialize(…)
ws: FractalTensor[Tensor['512,512'], float, 'gpu']] = initialize(…)
```

depth

RNN

batch head block num block size

```
qsss: FractalTensor[FractalTensor[FractalTensor[Tensor['32,64'], float, 'gpu']]]]= initialize(…)
ksss: FractalTensor[FractalTensor[FractalTensor[Tensor['32,64'], float, 'gpu']]]] = initialize(…)
vsss: FractalTensor[FractalTensor[FractalTensor[Tensor['32,64'], float, 'gpu']]]] = initialize(…)
```

programmable statically-shaped

Attention

The FractalTensor Program

RNN: FractalTensor code

```
// N, L, D stands for batch, length, depth
xss: [N,L]float32[1,512] = ... // load from storage
ws: [D]float32[512,512] = ... // load from storage

// output transformed from existing FractalTensors
ysss: [N,D,L]float32[1,512] = ⋯

// map over the batch dimension of xss
ysss = map xss xs ⟹
   // scan the depth dimension of ws
   yss = ws scanl xs, (x̃s, w) ⟹
      // scan the length dimension of xss
      ys = dilate(x̃s) scanl 0, (s, x) ⟹
         // user-defined small math function
         // [1,512]=[1,512]@[512,512]+[1,512]
         y = x @ w + s
```

Compute operators

Access operators

Functional **array compute** and **access operators** are tied to programmable dimensions.

1. array compute operators
   - map, reduce, fold, scan
2. array access operators
   - contiguously linear
   - constantly strided (dilation)
   - window (convolution)
   - indirect access (gather)

1. *No explicit tensor operators*
2. *Yet analyzable: loop nest with compute and access patterns* ***understandable by the compiler***

FractalTensor Code is Fully Permutable

The nested loops in FractalTensor code can be **reordered arbitrarily**[1]

Because:

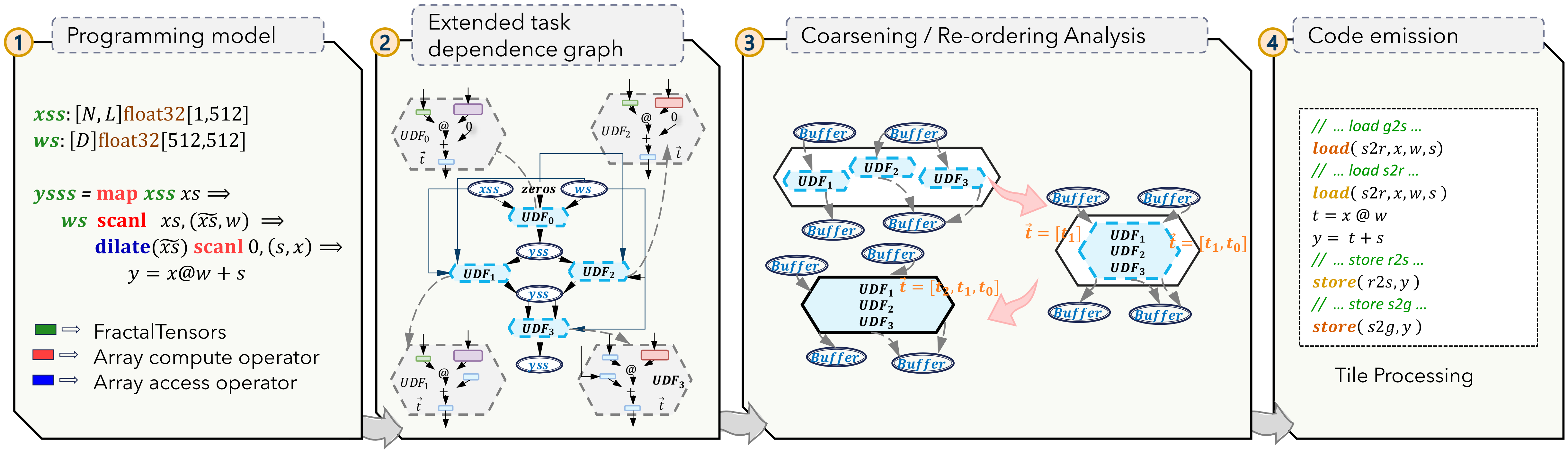
- FractalTensor code must follow SSA (single static assignment)
- Data dependence distance, regulated by array compute operators, is constant

Iteration-level data dependence can be permuted and moved to the outermost loop, allowing all inner loops to be parallel

Inner loops can then focus on data locality

1. Wolf, Michael E., and Monica S. Lam. "A loop transformation theory and an algorithm to maximize parallelism." *IEEE Transactions on Parallel & Distributed Systems* 2.04 (1991): 452-471.

Workflow Overview



FractalTensor code   ETDG (IR)   Logical pass   HW dependent pass

Extended Task Dependence Graph

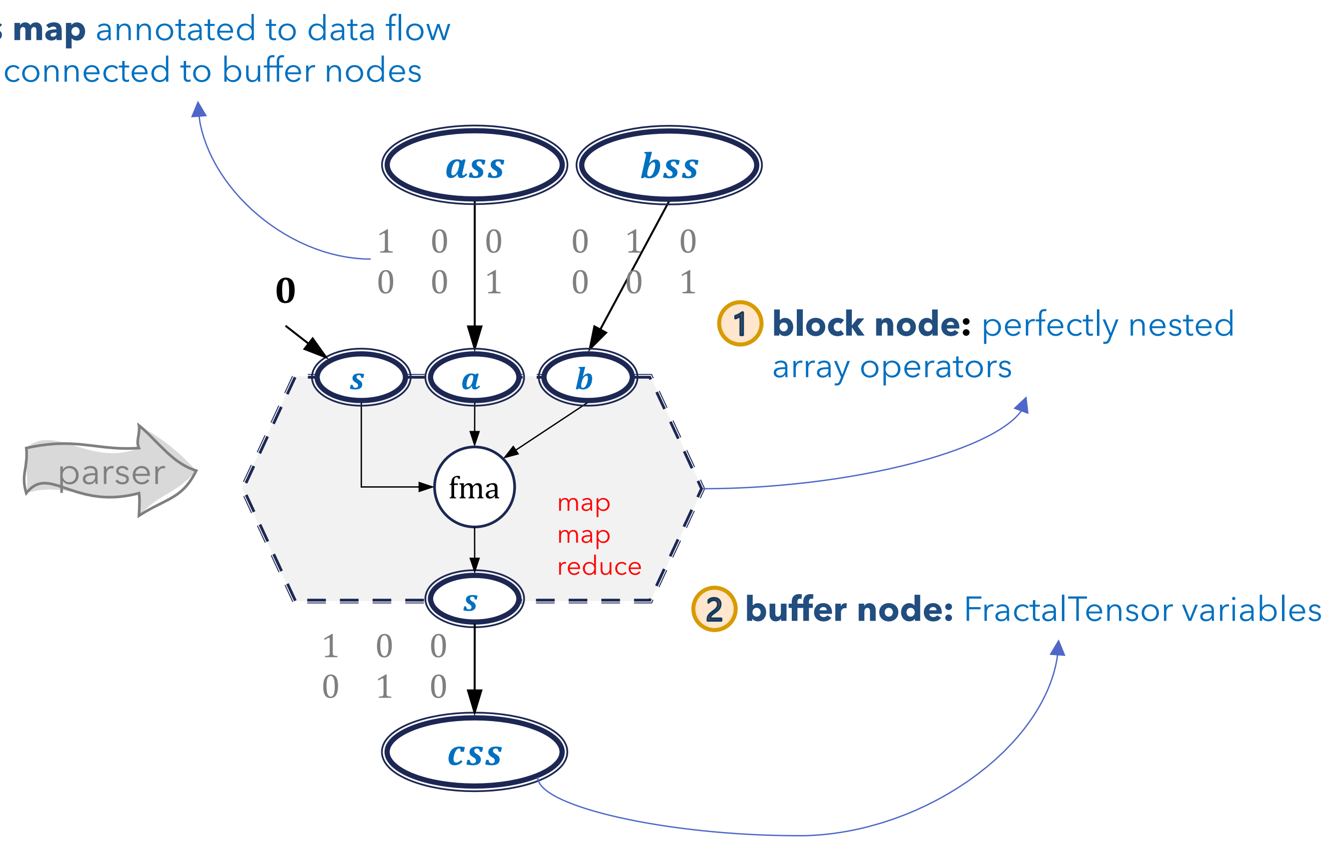
ETDG, parsed from the FractalTensor program, reflects its nested structure.

GEMM: FractalTensor code

```
ass:[16,8]float32[32,32] = ...
bss:[8,16]float32[32,32] = ...
css:[16,16]float32[32,32]  // output

css = ass.map as ⇒
    css = bss.map bs ⇒
        c = zip(as, bs).reduce 0, (s, (a, b)) ⇒
            c = a @ b + s
```

ETDG of GEMM



ETDG Transformation: Coarsening

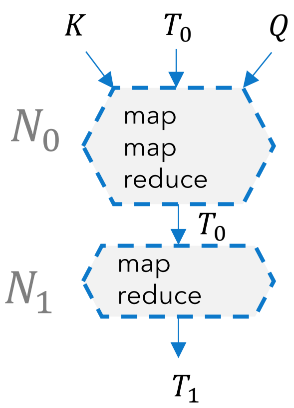
- Multi-level loops introduce control overhead
- Coarsening reduces the overhead

N_0 *for* i_3 *in [0,* d_3*)* // *map*
for i_2 *in [0,* d_2*)* // *map*
for i_1 *in [0,* d_1*)* // *reduce*, 0, +

$$T_0[i_3, i_2] = T_0[i_3, i_2] + Q[i_3, i_1] * K[i_2, i_1]$$

N_1 *for* j_2 *in [0,* d_3*)* // *map*
for j_1 *in [0,* d_2*)* // *reduce*, -inf, max

$$T_1[j_2, 1] = \max(T_0[j_2, j_1 - 1], T_0[j_2, j_1])$$



ETDG Transformation: Coarsening

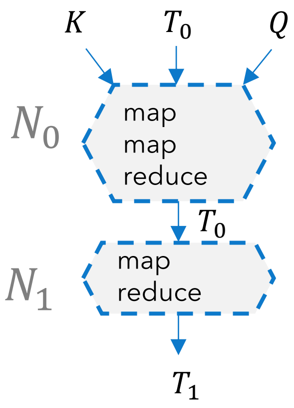
- Multi-level loops introduce control overhead
- Coarsening reduces the overhead

N_0
for i_3 *in [0,* d_3*)* // *map*
  for i_2 *in [0,* d_2*)* // *map*
    for i_1 *in [0,* d_1*)* // *reduce*, 0, +
      $T_0[i_3, i_2] = T_0[i_3, i_2] + Q[i_3, i_1] * K[i_2, i_1]$

Same shape

N_1
for j_2 *in [0,* d_3*)* // *map*
  for j_1 *in [0,* d_2*)* // *reduce*, -inf, max
    $T_1[j_2, 1] = \max(T_0[j_2, j_1 - 1], T_0[j_2, j_1])$

Same shape



ETDG Transformation: Coarsening

- Multi-level loop nest introduce control overheads
- Coarsening reduces the overhead

N_0
for i_3 in [0, d_3) // *map*
  for i_2 in [0, d_2) // *map*

Same shape

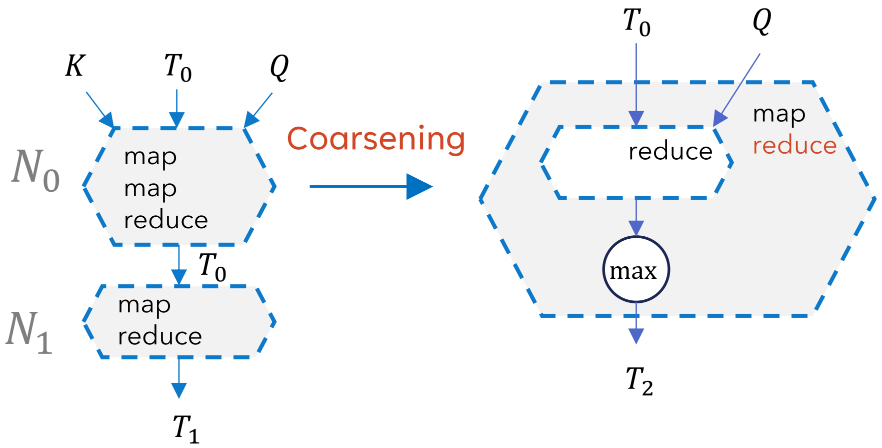
    for i_1 in [0, d_1) // *reduce*, 0, +
$$T_0[i_3, i_2] = T_0[i_3, i_2] + Q[i_3, i_1] * K[i_2, i_1]$$

N_1
for j_2 in [0, d_3) // *map*
  for j_1 in [0, d_2) // *reduce*, -inf, max

Same shape

$$T_1[j_2, 1] = \max(T_0[j_2, j_1 - 1], T_0[j_2, j_1])$$

for i_3 in [0, d_3) // *map*
  for i_2 in [0, d_2) // *map* → *reduce*
    for i_1 in [0, d_1) // *reduce*, 0, +
$$T_0[i_3, i_2] = T_0[i_3, i_2] + Q[i_3, i_1] * K[i_2, i_1]$$
$$T_1[i_3, 1] = \max(T_0[i_3, i_2 - 1], T_0[i_3, i_2]) \quad \text{// max, -inf}$$

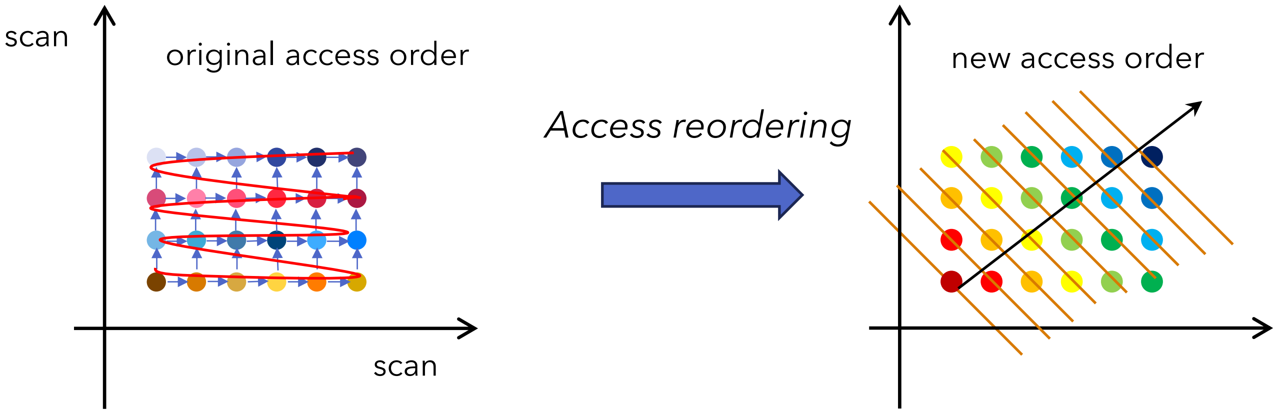


ETDG Transformation: Access Reordering

Enhance exploitable data parallelism and locality

- Permuate FractalTensor and move all data dependencies to the outermost dimension (Hyperplane method[1])
- Dimensions with data reuse moved to innermost to enhance locality (null space of access matrix to detect data reuse dimension[2])

RNN example



Nodes with the same color (a hyperplane) can be executed in parallel

1. Lamport, Leslie. "The parallel execution of DO loops." Communications of the ACM 17.2 (1974): 83-93.
2. Wolf, Michael E., and Monica S. Lam. "A data locality optimizing algorithm." Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation. 1991.

ETDG Transformation: Tile Processing & Code Emission

1. Hardware bottom-up tile processing library
    - BaseTile optimizes compute and memory usage aligned with TensorCore
2. Decompose buffer nodes into BaseTiles
3. Materialize access maps into load/store tiles

```
for (int k1 = 0; k1 < GIteratorA::sc1; ++k1) {
    g2s_a(gAs(k1), sA);
    g2s_b(gBs(k1), sB);
    __copy_async();
    __syncthreads();

    for (int k2 = 0; k2 < SIteratorA::sc1; ++k2) {
        s2r_a(sAs(k2), rA);
        s2r_b(sBs(k2), rB);

        compute::gemm(rA, rB, acc);
    }
}
r2s_c(acc, sC);
__syncthreads();
s2g_c(sC, gC);
```

load tiles from global to shared memory

load tiles from shared memory to register

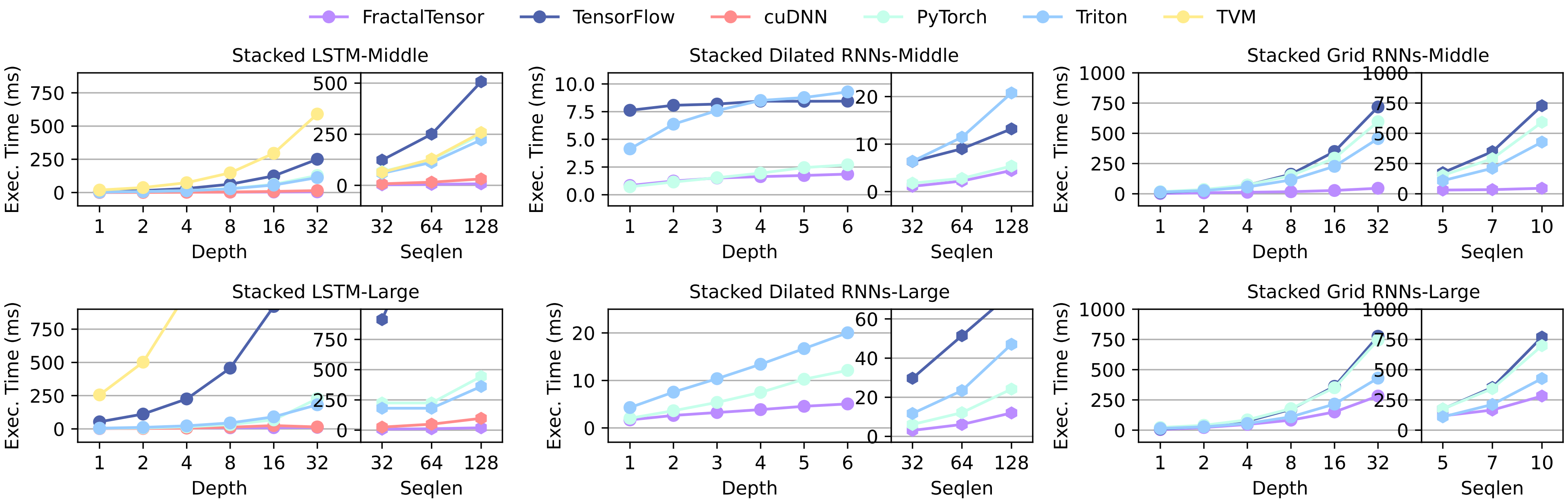
compute

store tiles from register to shared memory

store tiles from shared to global memory

GEMM with our tile library

Overall Performance on Stacked RNN and the Variants



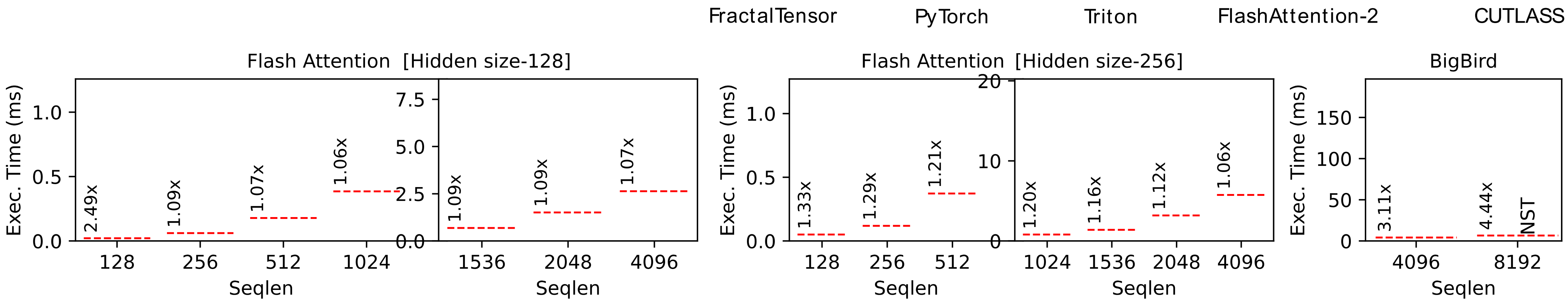
Existing practice

- Only standard stacked LSTM is optimized through the vendor library
- Slight algorithmic changes negate the optimization

In FractalTensor

- Optimizations apply to patterns commonly found in new RNN variants
- Reordering analysis identifies exploitable data parallelism, ensuring stacked RNN performance regardless with variants (e.g., changes in depth)

Overall Performance on Flash Attention and the Variants



Existing practice

- FlashAttention's online normalization algorithm is hard to express as a DAG
- Manual GPU memory optimization is complex due to TensorCore details

In FractalTensor

- Online normalization algorithm fits naturally into map and reduce operator nesting
- Tile library abstracts the hardware programming model and maximizes hardware usage
- Near-direct translation achieves *better* performance in multiple configurations

Conclusion

The dilemma of tensor operator: expressiveness and effectiveness

FractalTensor can solve the dilemma by

- An ADT to capture the key characteristics of tensors in DNN
- A set of array compute and access operator to compose arbitrary nested DNN structure based on FractalTensor
- Evaluation demonstrates that FractalTensor codes can achieve both expressiveness and effectiveness