FreeTensor: A Free-Form DSL with Holistic Optimizations for Irregular Tensor Programs

Shizhi Tang[1] Jidong Zhai[1] Haojie Wang[1] Lin Jiang[1] Liyan Zheng[1]
Zhenhao Yuan[1] Chen Zhang[1]

[1]Tsinghua University

Presenter: Shiwei Zhang

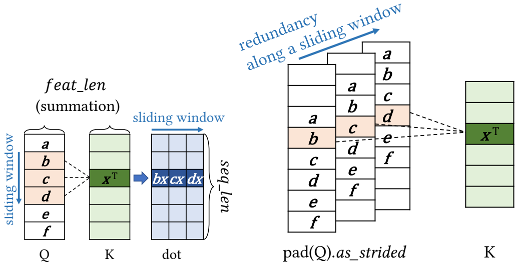
Content

Introduction

Irregular Tensor Programs

Current tensor programming frameworks (Tensorflow, PyTorch, etc.) work on whole-tensor level.

Irregular tensor programs usually include **fine-grained operations** that only use a part of a tensor and **combination of multiple operations** that should be fused.



(a) Longformer computation **(b)** Operator-based implementation

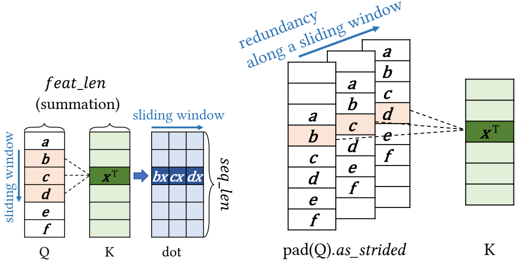
```
Q_strided = pad(Q, ...).as_strided(...)
dot = einsum(..., Q_strided, K)
```

(c) PyTorch implementation of Longformer

Free-Form Tensor Programs

```
# Q = create_var((seq_len, feat_len), "f32", "gpu")
# K = create_var((seq_len, feat_len), "f32", "gpu")
# V = create_var((seq_len, feat_len), "f32", "gpu")
@optimize # define an optimize region
def LongformerFwd(Q, K, V):
  Y = create_var((seq_len, feat_len), "f32", "gpu")
  for j in range(seq_len):
    dot = create_var((2 * w + 1), "f32", "gpu")
    for k in range(-w, w + 1):
      if j + k >= 0 and j + k < seq_len:
        dot[k + w] = sum(Q[j] * K[j + k])
    Y[j] = compute_y(dot, V[j - w : j + w])

@optimize # define an optimize region
def compute_y(dot, V_j):
  attn = softmax(dot)
  ... # the rest code is omitted
```



(a) Longformer computation **(b)** Operator-based implementation

```
Q_strided = pad(Q, ...).as_strided(...)
dot = einsum(..., Q_strided, K)
```

(c) PyTorch implementation of Longformer

Contributions

- FreeTensor DSL to express free-form tensor programs.
- Holistic compilation optimizations including partial evaluation, denpendence-aware transformation, and automatic code generation for different architectures.
- Fine-grained automatic differentiation (AD) with selective tensor materialization (gradient checkpointing).
- Evaluation shows that compared to PyTorch, JAX, TVM, Julia, and DGL, FreeTensor achieves up to 5.10x speedup (2.08x on average) without AD and up to 127.74x speed up (36.26x on average) with AD.

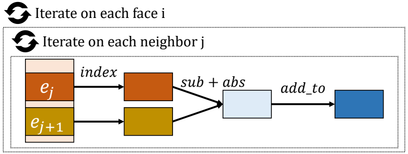
Background and Motivation

Current State

- **Tensorflow** and **PyTorch** use optimized libraries (cuDNN, cuBLAS, Intel MKL) for computation. New kernels have to be developed for new operations used in new models.
- **TVM** is proposed to reduce manual efforts in writing new kernels. However, it does not support irregular tensor programs.
- **Julia** is a general purpose programming language that is capable of expressing irregular tensor programs, but it fails to generate high-performance code due to lacking domain knowledge.

Motivating Example

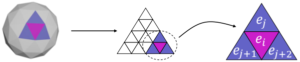
The same operation expressed with free-form tensor program does not include redundant operators (indexing, reshape, cat, etc.) and does not use extra memory.



(a) Free-form implementation

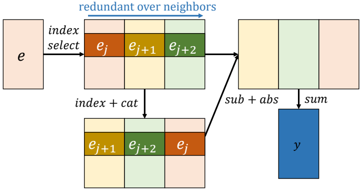
```
for i in range(n_faces):
    y = zeros(in_feats)
    for j in range(3):
        y += abs(e[adj[i, j], :]
                 - e[adj[i, (j + 1) % 3], :])
```

(b) Free-form implementation code



$$y_i = w_0 e_i + w_1 \sum_j e_j + w_2 \boxed{\sum_j |e_j - e_{j+1}|} + w_3 \sum_j |e_i - e_j|$$

(a) SubdivNet computation of a single mesh convolution, where there is a circular difference computation in the red box.



(b) Operator-based implementation of the circular difference.

```
# Step 1
adj_feat = index_select(e, 0, adj.flatten())
               .reshape(n_faces, 3, in_feats)
# Step 2
reordered_adj_feat = cat([adj_feat[:, 1:],
                          adj_feat[:, :1]], dim=1)
# Step 3
y = sum(abs(adj_feat - reordered_adj_feat), dim=1)
```

(c) PyTorch code of the circular difference

Challenges

- **Optimization with dependence**. Fine-grained control flow introduced by FreeTensor often contain data dependence that hinder potential code transformation.
- **Efficient automatic differentiation on complex control flows**. Loops often create a large number of intermediate tensors with AD. FreeTensor incorperates selective tensor materialization to mitigate this problem.

Free-Form DSL

Tensor definition and indexing

```
# declare a 3-D 32-bit floating-point tensor on cpu
A = create_var((2, 4, 6), "f32", "cpu")
# B is a 1-D tensor copied from A[0, 1]
B = A[0, 1]
# C is a 0-D tensor (scalar) copied from A[0, 1, 2]
C = A[0, 1, 2]
# D is a 2-D tensor with shape (2, 6), whose is the
# concatenation of A[0, 1] and A[0, 2]
D = A[0, 1:3]
```

Granularity-Oblivious Tensor Operations

With **integer ranged for-loops**, **branches**, and **always-inlined function calls**, FreeTensor supports tensor operations in any granularity.

```
# Q = create_var((seq_len, feat_len), "f32", "gpu")
# K = create_var((seq_len, feat_len), "f32", "gpu")
# V = create_var((seq_len, feat_len), "f32", "gpu")
@optimize # define an optimize region
def LongformerFwd(Q, K, V):
  Y = create_var((seq_len, feat_len), "f32", "gpu")
  for j in range(seq_len):
    dot = create_var((2 * w + 1), "f32", "gpu")
    for k in range(-w, w + 1):
      if j + k >= 0 and j + k < seq_len:
        dot[k + w] = sum(Q[j] * K[j + k])
    Y[j] = compute_y(dot, V[j - w : j + w])

@optimize # define an optimize region
def compute_y(dot, V_j):
  attn = softmax(dot)
  ... # the rest code is omitted
```

Dimension-Free Programming

Metadata of tensors are tracked and accessible within FreeTensor. This allows the user to write functions that work on tensors with different numbers of dimensions.

```
def add(A, B, C):
  for i1 in range(A.shape(0)):
    for i2 in range(A.shape(1)):
      ...
        for ik in range(A.shape(k-1)):
          C[i1,i2,...,ik] =
              A[i1,i2,...,ik] + B[i1,i2,...,ik]
```

(a) Adding k-D tensors with k nested loops

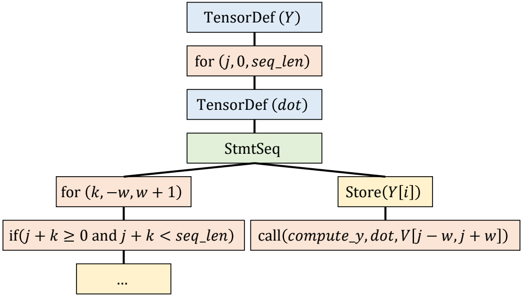
```
def add(A, B, C):
  if A.ndim == 0:
    C = A + B
  else:
    for i in range(A.shape(0)):
      add(A[i], B[i], C[i])
```

(b) Adding tensors with any dimensionality with a finite recursion

Code Generation

Stack-Scoped Abstract Syntax Tree (AST)

Stack-scoped: variables are restricted in subtrees.




```python
# Q = create_var((seq_len, feat_len), "f32", "gpu")
# K = create_var((seq_len, feat_len), "f32", "gpu")
# V = create_var((seq_len, feat_len), "f32", "gpu")
@optimize # define an optimize region
def LongformerFwd(Q, K, V):
  Y = create_var((seq_len, feat_len), "f32", "gpu")
  for j in range(seq_len):
    dot = create_var((2 * w + 1), "f32", "gpu")
    for k in range(-w, w + 1):
        if j + k >= 0 and j + k < seq_len:
            dot[k + w] = sum(Q[j] * K[j + k])
    Y[j] = compute_y(dot, V[j - w : j + w])

@optimize # define an optimize region
def compute_y(dot, V_j):
  attn = softmax(dot)
  ... # the rest code is omitted
```

Partial Evaluation

FreeTensor first evaluates the program using only the metadata (dimensions and shapes of tensors) to inline the recursive function calls.

```python
# def add(A, B, C):
if A.ndim == 0:
  C = A + B
else:
  for i in range(A.shape(0)):
    add(A[i], B[i], C[i])
```

A is a 3-D tensor, always false

Always true

(a) Source program

A[i] is a 2-D tenor

```python
for i in range(A.shape(0)):
  if A[i].ndim == 0:
    C[i] = A[i] + B[i]
  else:
    for j in range(A[i].shape(0)):
      add(A[i][j], B[i][j], C[i][j])
```

(b) The program after first round partial evaluation

Repeated

```python
for i in range(A.shape(0)):
  for j in range(A[i].shape(0)):
    for k in range(A[i][j].shape(0)):
      C[i][j][k] = A[i][j][k] + B[i][j][k]
```

(c) Target program

Dependence-Aware Transformation

The next step is to perform a series of transformations, each turns the AST into an equivalent but more efficient AST.

However, some of the transformations are not applicable with the presence of data dependence. FreeTensor uses a polyhedral analysis tool *isl* to detect these cases.

```
...
for j in range(seq_len):
  dot = create_var((2 * w + 1), "f32", "gpu")
  for k in range(-w, w + 1):
    if j + k >= 0 and j + k < seq_len:
      dot[k + w] = 0
      for p in range(feat_len):
        dot[k + w] += Q[j, p] * K[j + k, p]

  # compute_y, softmax is inlined
  dot_max = create_var((), "f32", "gpu")
  dot_max = -inf
  for k in range(2 * w + 1):
    dot_max = max(dot_max, dot[k])
  dot_norm = create_var((2 * w + 1), "f32", "gpu")
  for k in range(2 * w + 1):
    dot_norm[k] = dot[k] - dot_max
  ...
```

AST Transformation

	Name	Description
Loop Trans.	split	Split a loop into two nested loops
	merge	Merge two nested loops into one
	reorder	Reorder nested loops
	fission	Fission a loop into two consecutive loops
	fuse	Fuse two consecutive loops into one
	swap	Swap two consecutive statements including loops
Parallelizing Trans.	parallelize	Run a loop with multiple threads
	unroll	Unroll a loop into multiple copies of statements
	blend	Unroll a loop and interleave its statements from each iterations
	vectorize	Implement a loop with vector instructions
Memory Hierarchy Trans.	cache	Fetch part of a tensor to a smaller one before some statements, and store it back after that
	cache_reduce	Create a small tensor before reductions, and reduce back to the original tensor after that
	set_mtype	Change where a tensor stores
Memory Layout Trans.	var_split	Split a dimension of a tensor into two
	var_reorder	Transpose two dimensions of a tensor
	var_merge	Merge two dimensions of a tensor
Others	as_lib	Fall back to calling vendor libraries for common computations
	separate_tail	Separate the main body and tailing iterations of a loop, to reduce branching overhead

AST Transformation Strategy

FreeTensor allows users to choose any transformation to apply on any statement. On the other hand, it also provides a heuristic that applies 6 passes of transformations.

- `auto_fuse`: fuse loops to increase locality.
- `auto_vectorize`: implement loops with vector instructions.
- `auto_parallel`: bind loops to threads.
- `auto_mem_type`: try to put tensors near to processors (registers > scratch-pad memory > main memory).
- `auto_use_lib`: replace certain operations with external libraries.
- `auto_unroll`: unroll short loops to allow downstream optimizations.

Native Code Generation

FreeTensor applies further optimizations on the AST after transformations, including simplification on mathematical expressions, merging or removing redundant memory access, and removing redundant branches. FreeTensor also performs some backend-specific post-processing including inserting thread synchronizing statements, generating parallel reduction statements, and computing offsets of tensors in scratch-pad memory.

After that, FreeTensor generates OpenMP or CUDA code from the AST and invoke dedicated backend compilers like gcc or nvcc for further lower-level optimizations, and native code generations.

Automatic differentiation

Each write-after-read (WAR) dependency on the tensor corresponds to a version that need to be saved for backward pass. FreeTensor decides whether a tensor should be materialized at compile time.

```
for i in range(n):
  t = a[i] * b[i] # To be materialized in t.tape[i]
  y[i] = t * c[i]
  z[i] = t * d[i]
```

(a) Original program

```
for i in range(n):
  t.grad = z.grad[i] * d[i] + y.grad[i] * c[i]
  d.grad[i] = z.grad[i] * t.tape[i]
  c.grad[i] = y.grad[i] * t.tape[i]
  b.grad[i] = t.grad * a[i]
  a.grad[i] = t.grad * b[i]
```

(b) Backward pass with reuse

```
for i in range(n):
  t = a[i] * b[i]
  t.grad = z.grad[i] * d[i] + y.grad[i] * c[i]
  d.grad[i] = z.grad[i] * t
  c.grad[i] = y.grad[i] * t
  b.grad[i] = t.grad * a[i]
  a.grad[i] = t.grad * b[i]
```

(c) Backward pass with recomputing

Experiments

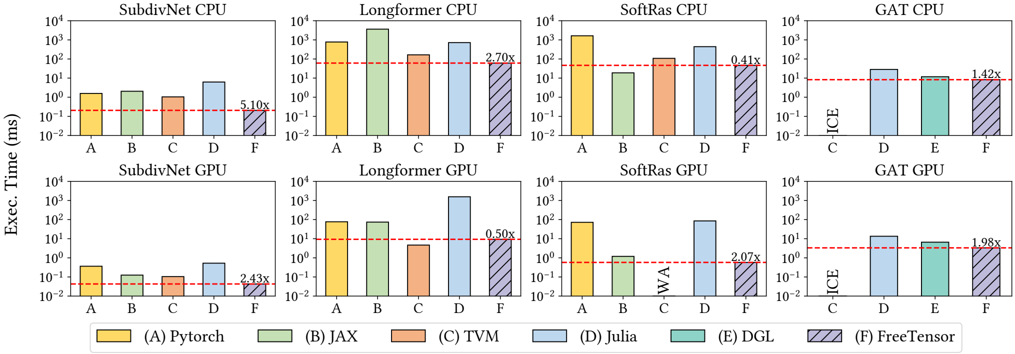
Experimental Setup

Hardware: A server with dual 12-core CPU and a V100 (32G).

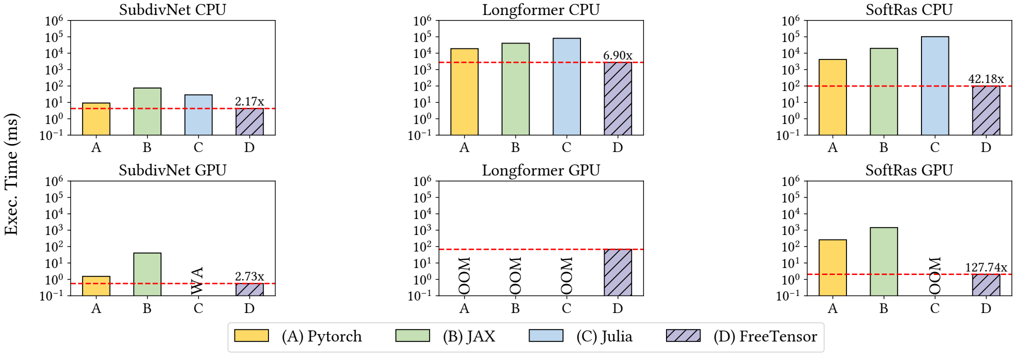
Baselines: PyTorch 1.8.1, Jax 0.2.19, TVM (Nov 4, 2021), Julia 1.6.3, and DGL 0.7.1.

Workloads:
- SubdivNet: a CNN for predicting properties of 3D objects.
- Longformer: a Transformer that only considers nearby tokens.
- SoftRas: a differentiable 3D rendering software.
- GAT: a GNN that uses attention for aggregation.

End-to-End Performance without AD

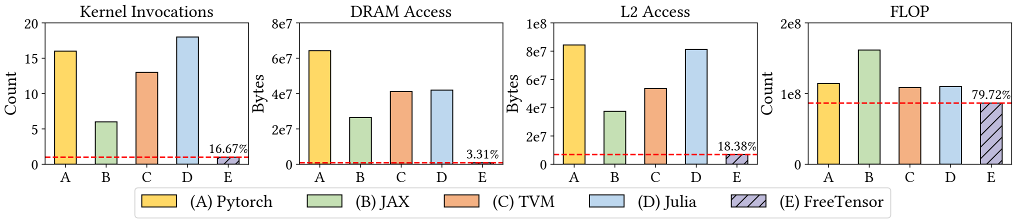


End-to-End Performance with AD



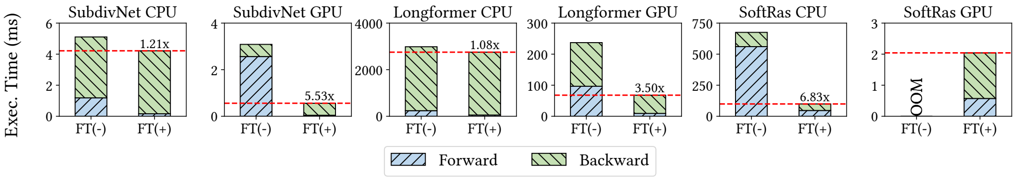
Analysis of the Speedup

By avoiding redundant tensors and using fewer operators, FreeTensor significantly reduces the numbers of kernel invocations, memory and cache access, and FLOPs.



Optimization for AD

For any tensors that FreeTensor decided to recompute it rather than to materialize it, there is a pure performance gain in a forward pass, since we no longer need to allocate memory and write to the memory for the materialization. As for a backward pass, there will also be a performance gain if the recomputing overhead is less than the reusing overhead.



*FT(+) and FT(−) denote using and not using selective tensor materialization

Compiling Time

	FreeTensor time	TVM time (rounds × each)
SubdivNet CPU	12.37 s	196 s (54 × 3.63 s)
SubdivNet GPU	13.10 s	237 s (131 × 1.81 s)
Longformer CPU	3.90 s	7531 s (2944 × 2.56 s)
Longformer GPU	8.30 s	8019 s (2944 × 2.72 s)
SoftRas CPU	4.43 s	2499 s (1024 × 2.44 s)
SoftRas GPU	9.49 s	10 361 s (2060 × 5.03 s)
GAT CPU	5.89 s	ICE
GAT GPU	9.17 s	ICE

*ICE means internal compiler error

Summary

Strength

- New approach (polyhedral analysis) to solve new problem (irregular tensor programs).
- Diverse baselines and benchmark models, with deep analysis for the speedup.
- A lot of concrete code examples.

Limitation

- The optimization strategy is greedy and not cost-based. We don't know if they hand-tuned the heuristics for the benchmark models.
- The benchmark models are all related to convolution operations, while they do not implement them with the convolution operation provided by cuDNN.
- They duplicates some optimizations that would be performed by the backend compiler (gcc and nvcc). Further, the backend compiler may override some decisions made by FreeTensor, like loop fusion, reordering, and unrolling.
- Only supports fully static graphs with the shapes of all tensors known at compile time.

Takeaways

- New models bring new challenges and opportunities to machine learning systems.
- We can find inspiration from other areas, like non-ML distributed systems and non-ML compiler techniques.

Thank you!