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

Shizhi Tang
Tsinghua University
Beijing, China
tsz19@mails.tsinghua.edu.cn

Jidong Zhai*
Tsinghua University
Beijing, China
zhaijidong@tsinghua.edu.cn

Haojie Wang
Tsinghua University
Beijing, China
wanghaojie@tsinghua.edu.cn

Lin Jiang
Tsinghua University
Beijing, China
jiangl17@mails.tsinghua.edu.cn

Liyan Zheng
Tsinghua University
Beijing, China
zhengly20@mails.tsinghua.edu.cn

Zhenhao Yuan
Tsinghua University
Beijing, China
yuanzh20@mails.tsinghua.edu.cn

Chen Zhang
Tsinghua University
Beijing, China
zhang-c21@mails.tsinghua.edu.cn

Abstract

Tensor programs are of critical use in many domains. Existing frameworks, such as PyTorch, TensorFlow, and JAX, adopt operator-based programming to ease programming, increase performance, and perform automatic differentiation. However, as the rapid development of tensor programs, operator-based programming shows significant limitations for irregular patterns since a large amount of redundant computation or memory access is introduced.

In this work, we propose FreeTensor, a free-form domain specific language which supports redundancy-avoid programming by introducing fine-grained control flow. With optimizations including partial evaluation, dependence-aware transformations, and fine-grained automatic differentiation, FreeTensor is able to generate high performance tensor programs on both CPU and GPU. Experiments show a speedup over existing tensor programming frameworks up to 5.10× (2.08× on average) without differentiation, and up to 127.74× (36.26× on average) after differentiation, for typical irregular tensor programs.

CCS Concepts: • **Computing methodologies** → *Parallel programming languages*; • **Software and its engineering** → *Source code generation.*

*Corresponding author.

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06.
https://doi.org/10.1145/3519939.3523448

Keywords: tensor computing, optimizing compilers, DSL

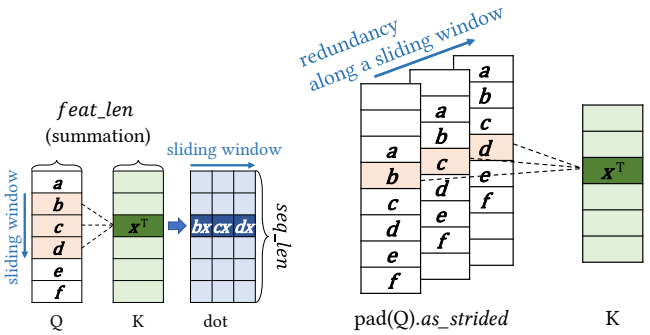
ACM Reference Format:
Shizhi Tang, Jidong Zhai, Haojie Wang, Lin Jiang, Liyan Zheng, Zhenhao Yuan, and Chen Zhang. 2022. FreeTensor: A Free-Form DSL with Holistic Optimizations for Irregular Tensor Programs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3519939.3523448

1 Introduction

Tensor programs are widely used across different domains including deep learning, computer graphics, scientific computing, and so on. Optimizing tensor programs is critical as tensor programs often operate on thousands of elements requiring massive parallelism to achieve high performance. However, such optimizations across various architectures require not only significant human efforts but also expertise on both algorithms and architectures.

Most of existing tensor programming frameworks, like PyTorch [29], TensorFlow [3], and JAX [17], encapsulate typical tensor computations into operators, such as matrix multiplication and convolution. In these frameworks, operators are highly optimized by hand for high performance and provided to users through libraries like cuDNN [15], cuBLAS [16] or Intel MKL [26].

Such operator-based frameworks can cover common tensor programs on certain hardware, but there are still significant performance challenges remaining unsolved. As most optimizations are done inside operators, traditionally, users are required to apply an operator to all elements in a tensor and chain multiple operators to implement a program. However, as the size of models grows larger, recent models tend to compute on part of a tensor to save computation. To express such models with operator-based frameworks,



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

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

(c) PyTorch implementation of Longformer

Figure 1. Partial attention implementation in Longformer.

tensors need to be transformed back and forth and a large amount of redundant computation or memory copies are introduced.

We take Longformer model [8] as an example, as shown in Figure 1(a). Different from traditional attention computing correlations for all tokens, Longformer computes correlations for pairs of nearby tokens that have a distance no greater than a threshold, thus it is capable to process much longer sequences. The range of the nearby tokens can be viewed as a sliding window. One common implementation on typical operator-based frameworks is to first pad and copy feature matrix `Q` along the sliding window, as shown in Figure 1(b), and the corresponding code is shown in Figure 1(c). As tensor `Q` is copied sliding-window-size-folded, significant memory redundancy is introduced.

Such types of tensor programs are increasingly common in emerging deep learning models. For simplicity, we call them irregular tensor programs. Compared with common tensor programs, such programs usually have the following features: 1) **Fine-grained operations.** The data required, used, and reused are not in a whole-tensor level, but determined by context. 2) **Combination of multiple operations.** These tensor programs usually need a series of operations to achieve corresponding functions. Multiple tensor operations should be combined.

Although users can use customized operators in operator-based frameworks, current frameworks only provide limited expressiveness to support irregular tensor programs. For example, `vmap` in JAX and PyTorch supports iterating through a tensor and applies operations to each part of it, but the iteration should be dependence-free. TVM [12] is fully built on top of customized operators, but each operator is limited to perfectly nested loops, with dependence-free loops on the outer side, and reduction loops on the inner side. Due to these limitations, users still have to introduce redundant operators.

In practice, when it is difficult to express a tensor program with an operator-based framework, users can still express the computation of part or whole of it in a general-purpose programming language, such as Python and C++. The main reason is that fine-grained control flow in such languages can easily remove the redundancy. In this work, we call such a program a **free-form tensor program**. However, such a program cannot achieve satisfying performance without careful optimizations, which require significant human efforts and expertise. Architecture-specific optimizations, such as parallelization and explicit utilization of cache or scratch-pad memory, have to be done manually for every hardware backend. Moreover, automatic differentiation (AD), which is desired in typical tensor applications, exacerbates this problem. The original program and its gradient should be implemented separately. General-purpose programming languages such as Julia [9] provide an easy way to interact with tensors and compute gradients. However, performing optimizations on them is still a difficult task.

To address this challenge, we propose FreeTensor, a redundancy-avoid domain-specific language (DSL) for tensor programs. Different from existing works, we introduce *fine-grained tensor operations* to reduce redundant computation and memory access while keeping tensors as first-class citizens to maintain programming simplicity. To analyze complex dependence introduced by fine-grained control flow, our FreeTensor compiler takes advantage of polyhedral techniques for automatic analysis, and thus we apply a series of optimizations including parallelization, loop transformation, and memory hierarchy transformation to generate high-performance code. Moreover, since differentiation is critical in tensor programs, FreeTensor supports fine-grained automatic differentiation with optimizations to reduce memory redundancy by balancing between storing or re-computing intermediate tensors.

In summary, we make the following contributions in this work:

- We propose a free-form DSL, named FreeTensor, which supports redundancy-avoid tensor programming by providing granularity-oblivious tensor operations.
- We provide holistic compilation optimizations in FreeTensor to generate highly efficient tensor programs, including partial evaluation on dimension-free programs with recursions, dependence-aware transformations on fine-grained control flows, and automatic code generation for different architectures.
- FreeTensor supports fine-grained automatic differentiation combined with efficient selective intermediate tensor materialization.
- Evaluation shows that compared with existing tensor programming frameworks, FreeTensor achieves up to 5.10×

speedup (2.08× on average) without differentiation, and up to 127.74× speedup (36.26× on average) with differentiation for typical irregular tensor programs.

The rest of this paper is organized as follows. Section 2 describes the problem with a detailed example. Section 3 and Section 4 describe our DSL and code generation in FreeTensor. Section 5 addresses performance issues in AD. Section 6 evaluates the performance of FreeTensor. Section 7 discusses some related works. Section 8 concludes the paper.

2 Background and Motivation

2.1 Background

Existing tensor programming frameworks including TensorFlow [3], PyTorch [29] express tensor programs as invocations to highly optimized libraries, including cuDNN [15], cuBLAS [16], and Intel MKL [26]. As the rapid development of tensor programs requires many new operators, code generation frameworks like TVM [12] are proposed to reduce manual efforts.

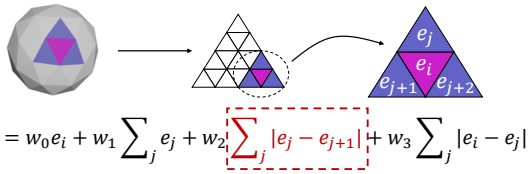
However, these frameworks lack effective support on emerging irregular tensor programs, which have features of partial operations on a complete tensor or complex control dependence. To cater to current frameworks, redundant computation and memory access are introduced in these programs. Although general-purpose programming languages like Julia [9] can partly eliminate these redundancies, they still fail to generate high-performance code due to lacking domain knowledge.

2.2 Motivating Example

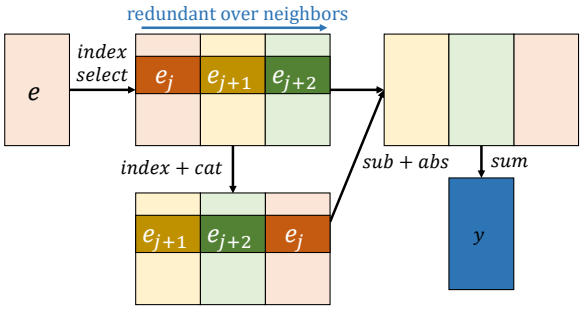
We take SubdivNet [19] in Figure 2 as an example to demonstrate the limitation of existing frameworks and how FreeTensor works. The major component in SubdivNet is multiple convolutions over a mesh. Similar to a traditional convolution over an image that combines the feature of each pixel and its adjacent pixels, a mesh convolution in SubdivNet combines the feature of each face and its adjacent faces. To overcome the order-invariant nature among faces, SubdivNet introduces a circular difference computation, which is described in the red box of Figure 2(a). For each central face e_i, SubdivNet finds the feature vectors of its three adjacent faces e_j, e_{j+1}, and e_{j+2} via an adjacency array, and computes their differences in a circular manner.

In this case, feature vectors e_j, e_{j+1}, and e_{j+2} are fetched and used with respect to a central face, but will never be reused by other central faces without extra indexed access. Ideally, they should be created and used individually, instead of gathering from all central faces before computations.

However, a typical implementation in an operator-based framework requires collectively operating these data, which is shown in Figure 2(b). Specifically, as shown in Figure 2(c), the Pytorch implementation of this program consists of the following steps:



(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

Figure 2. Operator-based implementations of SubdivNet.

- **Step 1:** Construct a 3-D tensor (`adj_feat`) to store the features of adjacent faces, by calling a `flatten`, an `index_select` and a `reshape` in sequence. Each element in the resulting tensor `adj_feat[i, j, k]` stores the `k`-th factor of the feature of the `j`-th adjacent face of face `i`.
- **Step 2:** Slice the `adj_feat` tensor, reorder it, concatenate it back, and now face e_{j+1} has the same index with the original face e_j.
- **Step 3:** Do a subtraction and compute its absolute value. After that, the sum is calculated.

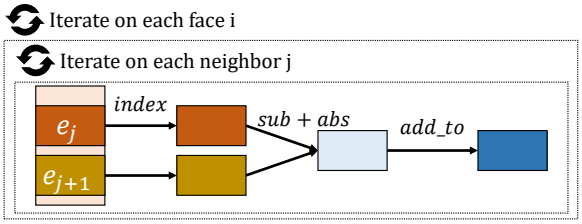
Although every operator benefits from highly optimized native code in a vendor-provided library, every intermediate value should be stored as full-sized tensors, as depicted in Figure 2(b). This introduces significant memory access redundancy: tensor `adj_feat` is of size `n_faces * 3 * n_feat`, which is much larger than input and output tensors and incurs a huge memory access overhead. Moreover, redundant

operators `flatten`, `index_select`, `reshape`, and `cat`, are included, which are only used to rearrange existing data, but do not perform a meaningful computation.

Even though TVM supports highly customized operators, the indirect indexing on tensors still stops it from representing the program without combining traditional operators like in Figure 2(b). A general-purpose programming language like Julia is able to represent such a case in a fine-grained control flow. However, it requires significant manual optimization and parallelization.

2.3 Challenges of FreeTensor

To solve the problem above, we adopt a fine-grained control flow to remove unnecessary memory accesses, shown in Figure 3(b). Specifically, we iterate each face `i` and its neighbor `j`, and directly index the `j`-th and `(j+1)`-th face from the input tensor `e`. After that, we perform fine-grained tensor operations to calculate the difference, and result is directly accumulated to a tensor `y`. Here, e_j, e_{j+1}, and e_{j+2} are individual tensors as shown in Figure 3(a), where each tensor operation is fine-grained and performed individually.



(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

Figure 3. The circular difference of SubdivNet in FreeTensor.

FreeTensor adapts such a fine-grained approach to operate each tensor. Compared to the operator-based implementation, the program in FreeTensor accesses tensor elements in an on-demand manner, without redundant operations to rearrange them beforehand. However, complex control flows brought by FreeTensor introduce significant challenges for efficient code generation. We summarize main challenges here.

- **Optimization with the presence of dependence.** Fine-grained control flow introduced by FreeTensor makes efficient code generation even harder. We expect FreeTensor can automatically generate performant code without too much manual effort. However, complex control and data dependence hinder potential code transformation.
- **Efficient automatic differentiation on complex control flows.** Automatic differentiation (AD) on a program with complex control flow can further introduce much redundancy, which neutralizes the benefit provided by a free-form language. How to design a high-performance AD mechanism is more challenging.

3 Free-Form DSL

This section describes how we design a free-form domain-specific language (DSL) for emerging irregular tensor programs. As a DSL for tensor programs, tensors should be treated as first-class citizens for programming simplification. To support operations on partial tensors to eliminate redundant computation and memory access, FreeTensor provides support for tensor operations in any granularity by introducing fine-grained control flows and partial tensor indexing. Moreover, in order to generate high-performance code, FreeTensor provides extra meta-information and programming guidance for users to assist underlying compilation. The rest of this section will elaborate on the above designs.

3.1 Tensors as First-Class Citizens

Tensor definition. FreeTensor treats tensors as first-class citizens to ease programming difficulty. More specifically, tensors (of various element types) are primary data types in FreeTensor. We call a tensor with dimension N an N-D tensor, and scalar is treated as a 0-D tensor. Tensors are stored in a compact memory layout, and tensor shape is not mutable once it is created. To guarantee that there is no overlap between tensors, tensors are copied by values. Tensor elements can be any primary scalar data type, including integer, single/double/half floating point, etc., which cover typical tensor programs' needs.

```
# 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]
```

Figure 4. Tensor definition and indexing.

Tensor indexing. Figure 4 shows how FreeTensor defines and indexes a tensor. Tensors can be defined on different devices, including CPU, GPU, etc. FreeTensor provides user-friendly NumPy [18]-style indexing rules, which is capable to index any sub-area in a tensor. Such indexing rules allow users to index partial tensors, thus supporting operations on

partial tensors flexibly to avoid unnecessary computation and memory access.

All the operations in FreeTensor's DSL, including arithmetic operators (+, -, *, /, etc.), built-in functions (sum, abs, etc.), and function calls, are directly performed on tensors. These operations will then be lowered to high-performance native code, which will be introduced in Section 4.

3.2 Granularity-Oblivious Tensor Operations

Tensor operator is a widely used abstraction in tensor programs, bringing significant simplification for tensor programming. Since irregular tensor programs usually operate partial tensor instead of a whole tensor to save computation, supporting partial tensor operations is necessary. Users of traditional operator-based frameworks are expected to invoke operators as coarse-grained as possible. As mentioned in Section 2, implementing irregular tensor programs with such tensor operators will bring extensive computation and memory access redundancy. To tackle this problem, we introduce granularity-oblivious operations in FreeTensor to provide the ability to write redundancy-avoid 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
```

Figure 5. Free-form implementation code of Longformer in Figure 1. The range `j-w` to `j+w` marks the sliding window.

To achieve granularity-oblivious tensor operations, we introduce the following semantics in our DSL: *integer ranged for-loops*, *branches*, and *always-inlined function calls*. We will give more explanation about why we introduce these critical features in Section 4. With the help of these semantics, FreeTensor can support tensor operations in any granularity. Figure 5 shows an example how the Longformer example in Figure 1 is implemented using FreeTensor. In this case, we iterate along the input sequence with a for-loop `j`, and iterate alongside the sliding window with a loop `k`. Elements of `K` are directly accessed by index `j+k`, without copying the whole tensor beforehand.

FreeTensor also provides a tensor operator library, called `libop`, supporting operators ranging from basic operator-like element-wise operations, reductions, and matrix multiplications to complex ones like a softmax. We implement `libop` in pure DSL code instead of directly mapping to native code implementation. At compile time, function calls to `libop` will be fully inlined as nested loops, then optimized together with the rest of a program. For example, the tensor-wise `zeros`, `abs`, `-` and `+=` in Figure 3(b) are all provided by `libop`.

3.3 Dimension-Free Programming

Tensor dimension is a key property for tensor computation, and most operations of tensor programs are closely connected to transformations around the tensor dimension. We record dimension-related properties in the meta-data of a tensor, which also enjoys first-class support. The dimensionality, shapes, element types, and device placements can be accessed using the `.ndim`, `.shape`, `.dtype`, and `.mtype` properties, respectively. Particularly, tensor shapes are kept in their expression form. For example, after we flatten an $N \times 2$ -shaped 2-D tensor `A` to a 1-D tensor `B`, we know that the length of `B` should be $2N$, instead of an arbitrary number. We can safely assert that $2N$ is an even number and reshape `B` back to an $N \times 2$ shape.

```
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

Figure 6. Example of element addition for high-dimensional tensors.

In FreeTensor, we express a computation for any dimensionality with a finite recursion. Figure 6 gives an example of how to write dimension-free tensor programs using finite recursions. As shown in Figure 6(a), if a tensor's

shape cannot be determined when writing the tensor program, users cannot write a straightforward nested-loop program, which brings significant programming complexity. In FreeTensor, we suggest users write tensor programs with undetermined dimensionality using finite recursions, as shown in Figure 6(b). Such recursions will be further expanded to nested loops using partial evaluation at compile-time, which will be illustrated in Section 4.1.

4 Generating High Performance Code

Our free-form DSL allows users to write tensor programs without redundant computation or memory access. Programs written by FreeTensor DSL will be parsed to a *stack-scoped* abstract syntax tree (AST), as FreeTensor's intermediate representation (IR), to perform further optimizations and generate high-performance native code. With this design, each tensor is alive only in the sub-tree of its definition node, called `TensorDef` node. The stack-scoped restriction brings significant simplification for IR transformation: 1) we are able to transform AST without breaking an allocation-freeing pair; 2) by limiting the life scope of a tensor to a sub-tree, most of the false dependence in dependence analysis can be eliminated.

Figure 7 shows an AST of `LongformerFwd` function, whose code is in Figure 5. We inline all function calls to perform holistic optimizations across functions. Figure 8 shows the resulting program of the example in Figure 5. After that, we perform multiple optimizations on AST.

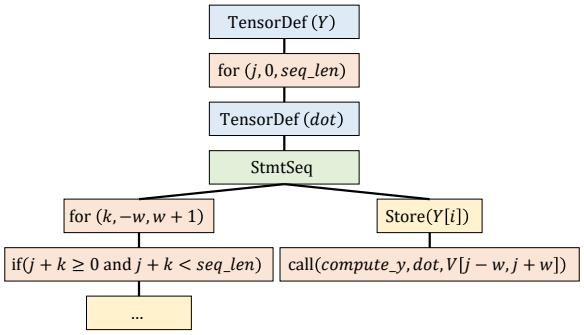


Figure 7. AST of `LongformerFwd` in Figure 5. Some nodes are omitted.

4.1 Partial Evaluation for Dimension-Free Programming

As mentioned in Section 3.3, users can write dimension-free tensor programs with finite recursive functions. FreeTensor supports such a feature by partially evaluating program with respect to the meta-data of the tensors. By providing first-class support on tensors with meta-data, dimensionalities of all tensors in the programs are known at compile-time, which makes it possible to apply partial evaluation for general dimensionality programs implemented with recursion.

```
1  ...
2  for j in range(seq_len):
3    dot = create_var((2 * w + 1), "f32", "gpu")
4    for k in range(-w, w + 1):
5      if j + k >= 0 and j + k < seq_len:
6        dot[k + w] = 0
7        for p in range(feat_len):
8          dot[k + w] += Q[j, p] * K[j + k, p]
9
10   # compute_y, softmax is inlined
11   dot_max = create_var((), "f32", "gpu")
12   dot_max = -inf
13   for k in range(2 * w + 1):
14     dot_max = max(dot_max, dot[k])
15   dot_norm = create_var((2 * w + 1), "f32", "gpu")
16   for k in range(2 * w + 1):
17     dot_norm[k] = dot[k] - dot_max
18   ...
```

Figure 8. Inlined program of the example in Figure 5.

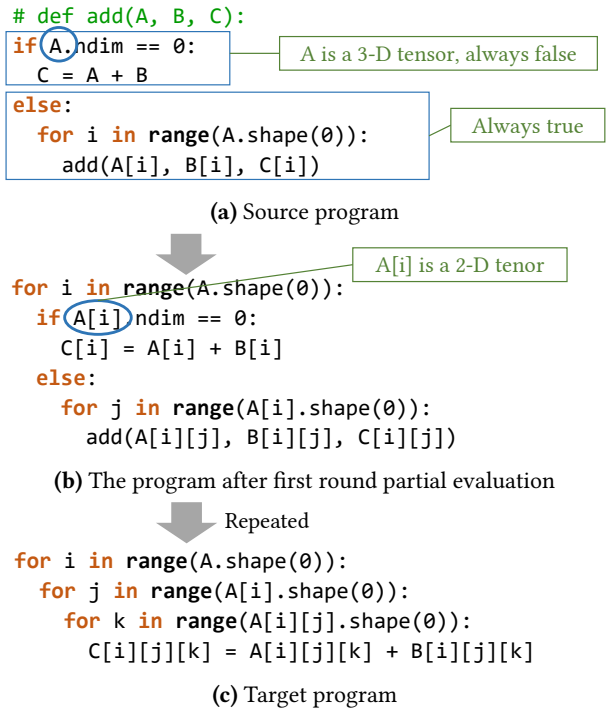
We take the code in Figure 6(b) as an example to illustrate evaluating process, shown in Figure 9. Figure 9(a) is the original program, with a recursive function call `add`. Suppose that `A` is a 3-D tensor, then our compiler knows that the `if` condition is always false, so all the statements in the `if` branch are discarded, while the statements in the `else` branch will always be executed. The function call `add` at the last line is then evaluated, resulting in an optimized program as shown in Figure 9(b). Then the compiler repeats such partial evaluation process in Figure 9(b). Notice that `A[i]` is a 2-D tensor. Finally, the final program after applying partial evaluation is shown in Figure 9(c), where the recursive function call is transformed into a nested loop.

4.2 Dependence-Aware Transformation

After inlining, we need to perform a series of transformations on the AST to generate efficient code from FreeTensor IR. We apply a rich collection of transformations for various optimizations, including transformations on loops, parallelization, memory hierarchy, memory layout, and others, as summarized in Table 1. These transformations are similar to the schedule optimizations of Halide [31] and TVM [12], but the fine-grained control flow brings significant challenges on how to correctly apply these transformations. For example, TVM only supports transformations on a perfectly nested loop, meaning that there is no complex dependence that needs to be considered while transforming. However, after applying fine-grained control flow, complex dependence is introduced. Still taking Figure 8 as an example, fusing loops at Line 4, 13, and 16 can bring better locality. fusing the loops at Line 4 and 13 is possible, but fusing the loops at Line 13 and 16 is incorrect because of the inter-iteration dependence

Table 1. AST transformations

	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



(a) Source program

(b) The program after first round partial evaluation

(c) Target program

Figure 9. Example of partial evaluation on dimension-free recursion. Suppose A, B, and C are 3-D tensors.

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

Figure 10. Fused program of the example in Figure 8.

on `dot_max`. The fused program is shown in Figure 10, where an offset "+w" is applied to iterator `k`, to make the indices consistent.

Since the dependence determines whether a transformation is correct, FreeTensor performs dependence analysis before applying transformations. Different from operator-based frameworks, we need to analyze programs in an instance-of-statement-wise precision instead of statement-wise precision, where an instance of a statement refers to a statement in a specific loop iteration. This means traditional data-flow-graph-level analysis is not enough for FreeTensor.

There have been many studies on how to analyze dependences in an instance-of-statement-wise precision and how

to guide a program transformation given the dependences. Early studies are summarized in [28] and [4]. Later, a mathematical theory named polyhedral analysis is introduced to analyze dependences systematically, and multiple solvers of polyhedral analysis are designed to automate the analysis. Given memory accesses defined as Presburger formulas, dependences can be derived by solving equations and inequations of them [40]. Works on theories and solvers of polyhedral analysis include Omega [30], PolyLib [23], PPL [6] and `isl` [39].

In FreeTensor, we use `isl` for dependence analysis, and implement ideas of dependence-aware program transformations discussed in [28] and [4] in our IR. In the rest of this section, we will illustrate how FreeTensor adopts these techniques to perform dependence-aware transformations, including *loop transformations*, *parallelizing transformations*, and *memory transformations*.

```
a = create_var((N, M), ...)
for i in range(1, N - 1):
  for j in range(1, M - 1):
    a[i + 1, j]          # (1)
      = a[i - 1, j + 1]  # (2)
      + a[i - 1, j - 1]  # (3)
```

Figure 11. Example program with complex inter-iteration dependence.

4.2.1 Loop transformations.

Take the program in Figure 11 as an example. There is a read-after-write (RAW) dependence between `(2)` and `(1)`, and a RAW dependence between `(3)` and `(1)`. We need to keep both dependence when transforming the loops.

In polyhedral analysis, each memory access, `(1)`, `(2)` or `(3)`, is defined as a mapping from an iteration space $\mathbb{Z}^{(N-2)\times(M-2)}$ to an array index space $\mathbb{Z}^{N\times M}$, describing which array index is accessed in which iteration:

$$M_{(1)} = \{(i,j) \to (i+1,j) : 1 \le i < N-1, 1 \le j < M-1\}$$
$$M_{(2)} = \{(i,j) \to (i-1,j+1) : 1 \le i < N-1, 1 \le j < M-1\}$$
$$M_{(3)} = \{(i,j) \to (i-1,j-1) : 1 \le i < N-1, 1 \le j < M-1\}$$

By combining these mappings using `isl`, we infer the RAW dependence between (2) and (1) as:

$$\begin{aligned} &M_{(2)\to(1)} \\ =&\{\mathbf{p} \to \mathbf{q} : \exists \mathbf{r} : (\mathbf{p} \to \mathbf{r}) \in M_{(1)} \wedge (\mathbf{q} \to \mathbf{r}) \in M_{(2)} \wedge \mathbf{p} >_{lex} \mathbf{q}\} \\ =&\{(i,j) \to (i-2,j+1) : 1 \le i < N-1, 1 \le j < M-1\}, \end{aligned}$$

where $>_{lex}$ means lexicographically greater, and $\wedge$ means logical and.

From $M_{(2)\to(1)}$, we know `(1)` should be 2 iterations later in `i` and 1 iteration earlier in `j` than `(2)`. We keep this restriction when transforming the loops. For example, we cannot `reorder` these two loops, otherwise the statement will access an element not computed yet. Restrictions for the dependence between `(3)` and `(1)` are similar.

To explain more specifically how dependence limits the loop transformations, we take the examples in Figure 12 and discuss whether a `reorder` transformation can be applied on the nested loop.

- Applying `reorder` on the program in Figure 12(a) is correct because there is no dependence along the reverse direction of each loop.
- Applying `reorder` on the program in Figure 12(b) is incorrect since there is a dependence $(i, M-1) \to (i+1, 0)$.
- Applying `reorder` on the program in Figure 12(c) is correct because additive communicative law allows us to reduce `b[i,j]` to `a` in any order. FreeTensor introduces a `ReduceTo` node to process any `a=a+b` like statements, and any WAW dependence between `ReduceTo` nodes can be ignored.
- Applying `reorder` on the program in Figure 12(d) is correct because `t` with a WAW dependence is a false dependence. FreeTensor's stack-scoped AST can easily filter out the false dependence. In this case, there is a WAW dependence $(i_1, j_1, k_1) \to (i_2, j_2, k_2)$ on tensor `t`. Since `t`'s lifetime is inside the loop `j`, FreeTensor performs a projection $\{(i,j,k) \to (k) : i,j,k \in \mathbb{Z}\}$ on `t`'s dependence, resulting a dependence $(k_1) \to (k_2)$. Thus, FreeTensor can apply `reorder` transformation on loop `i` and `j`.

```
for i in range(N):
  for j in range(M):
    a[i, j] = b[i, j] + 1
```

(a) Can `reorder`

```
for i in range(N):
  for j in range(M):
    a = a * b[i, j] + 1
```

(b) Cannot `reorder`

```
for i in range(N):
  for j in range(M):
    a = a + b[i, j]
```

(c) Can `reorder`

```
for i in range(N):
  for j in range(M):
    t = create_var((K), ...)
    for k in range(K):
      t[k] = a[i, j, k]
      b[i, j, k] = t[k]
```

(d) Can `reorder`

Figure 12. Correct and incorrect cases for applying a `reorder` transformation on loop `i` and `j`.

4.2.2 Parallelizing transformations.

Since tensor programs always require massive parallelism, parallelizing transformations are critical for high-performance code generation. The challenges of parallelizing transformations not only come from complex dependence introduced by fine-grained control flow but also come from various parallel models on different hardware architectures. We take the examples in Figure 13 to illustrate how FreeTensor applies parallelizing transformations.

```
for i in range(N):
  a[i] = b[i] + 1
```

(a) Can `parallelize`

```
for i in range(N):
  a = a * 2 + b[i]
```

(b) Cannot `parallelize`

```
for i in range(N):
  a[i] = b # b can be either thread-local or shared
```

(c) Depends on parallel model and memory hierarchy

```
for i in range(N):
  a += b[i]
```

(d) Can `parallelize` with a parallel reduction algorithms

```
for i in range(N):
  a[idx[i]] += b[i]
```

(e) Can `parallelize` with atomic operations

Figure 13. Correct and incorrect cases for applying `parallelize` on loop `i`.

- Figure 13(a) shows a parallelizable program without dependence.
- Figure 13(b) shows a non-parallelizable example with cross-thread dependence.
- In Figure 13(c), whether `parallelize` transformation can be applied is relevant with specific parallel models and memory hierarchy. For example, if `b` is thread-local, loop `i` cannot be parallelized because `b` is only visible on one of the threads; but if `b` is stored on shared memory, this loop can then be parallelized, because `b` is visible on all threads.
- Figure 13(d) presents a reduction along with the same index, which can be lowered with parallel reduction algorithms.
- Figure 13(e) presents a random-access reduction, which can be parallelized with atomic instructions.

4.2.3 Memory transformations.

There are two types of memory transformations that need to be applied: memory layout optimization, and memory hierarchy optimization.

Memory layout transformations help improve spatial locality in data access by reordering elements in a tensor. For example, we can transpose an $(N \times M)$-shaped tensor to an $(M \times N)$ shape, so that the N dimension can be iterated contiguously.

Different from the design in TVM, which performs memory layout optimizations by altering the combination of operators, we implement these optimizations as fine-grained AST transformations, which exposes more opportunities to perform holistic optimizations with other transformations.

Memory hierarchy transformations help use cache and scratch-pad memory on processors.

The `cache` and `cache_reduce` transformations introduce new tensors into the AST, and we have to infer their sizes. For the program in Figure 14(b), the problem is how to infer that the newly introduced tensor `a.cache` is of a shape `M` and

```
for i in range(n):
  for j in range(m):
    f(a[i + j])
```

(a) Source Program

```
for i in range(n):
  a.cache = create_var((m,), ...)
  for j in range(m):
    a.cache[j] = a[i + j]
  for j in range(m):
    f(a.cache[j])
  for j in range(m):
    a[i + j] = a.cache[j]
```

(b) Target Program

Figure 14. Example of applying the `cache` transformation for tensor `a` around loop `j`.

`c[j]` maps to `a[i+j]`. To solve this problem, we analyze the lower and upper bounds of each index of tensors, which is `i+j` in this program. `0`, `i`, and `i+j` are all lower bounds of `i+j`, while `n+m-2`, `i+m-1`, and `i+j` are all upper bounds of it. We keep all of these bounds. Since we are caching between the outer loop and the inner loop, where iterator `i` is defined but `j` is not, we look for the tightest bound, which is `[i, i+m-1]`, or `[i, i+m)`. Therefore, we know we are caching `a[i:i+m]` to an `m`-shaped tensor `a.cache`.

4.3 Code Generation

All the transformations we provide are exposed to users. We provide an API to query a statement in the program in order to apply a transformation. We also recognize that it may require some expertise to optimize a program using these transformations, and most users may expect an automatic strategy to apply these transformations. FreeTensor implements a prototype of a rule-based auto-transforming strategy to tackle such challenges. We currently implement 6 passes to try applying the transformations. These passes are driven by heuristics considering specific architectures and invoked one by one. Thanks to the dependence analysis in Section 4.2, we can aggressively try transformations without worrying about their correctness. The passes include:

1. `auto_fuse`: Try to fuse nearby loops to increase locality using the `fuse` transformation. Other transformations like `swap` may be applied to enable it.
2. `auto_vectorize`: Find some loops that access data contiguously, reorder them as inner-most loops using loop transformations, and then implement a loop with vector instructions or warp with the `vectorize` or `parallelize` transformation.
3. `auto_parallelize`: Merge some outer loops with the `merge` transformation, and then bind them to threads with the `parallelize` transformation. For some backends like GPU featuring multiple levels of parallelism, we split loops using `split` before binding them.

4. `auto_mem_type`: Try to put tensors as near to processors as possible. Registers are preferred over scratch-pad memory, which is further preferred over main memory.
5. `auto_use_lib`: Try to replace computation-intensive subprograms with calls to external libraries with the `use_lib` transformation. Transformations like `fission` may be applied to enable it.
6. `auto_unroll`: Unroll very-short loops to unleash optimizing opportunities for backend compilers.

For any user program, these passes are automatically invoked, but users are free to override them and manually apply other transformations. Beyond these basic strategies, we are working on a machine-learning-guided solution similar to Ansor [44], which will be our future work.

We apply further optimizations on the AST after transformations, including simplification on mathematical expressions, merging or removing redundant memory access, and removing redundant branches. We also perform 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, we generate 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. A DSL function is finally compiled as a shared library, which can be dynamically loaded from Python to run.

5 Automatic Differentiation

5.1 Fine-Grained Automatic Differentiation

Automatic Differentiation (AD) is desired for tensor applications. AD helps users generate a gradient program from an original program, where a gradient program is used to compute the gradient of each input with respect to the program's output. A gradient program consists of a forward pass and a backward pass. The forward pass computes the output while keeping some intermediate tensors during its execution. The backward pass computes gradients and reuses intermediate tensors kept by the forward pass.

Inspired by Enzyme [27] and Zygote [20], we design a general AD that is capable to differentiate fine-grained control flow introduced by FreeTensor. Figure 15(b) gives an example of a backward pass generated from the original program in Figure 15(a). We keep in mind that the differentiated program should still be optimizable by FreeTensor. Therefore, we design our AD as a transformation pass on the AST. The resulting program is also an AST, which enjoys the same optimization opportunities as the original program.

One of the problems that hinder optimizing the differentiated program lies in intermediate tensors. In the procedure of AD, some intermediate tensors should be materialized in a forward pass, and then retrieved back in the backward pass. This procedure is also called checkpointing or saving a tensor into a *tape* in some literature. However, an intermediate tensor may be written many times in the program, so it has to be materialized into multiple versions. For example, the scalar `t` in Figure 15(a) (treated as a 0-D tensor) will be materialized in version `i` after its `i`-th assignment. Some existing works like Tangent [36] and Zygote [20] maintain a version number at run time, which hinders further parallelization. Instead, we analyze a symbolic version number in FreeTensor, similar to Enzyme [27]. Specifically, taking advantage of the polyhedral analysis, we look for WAR dependence on `t`, where each WAR dependence corresponds to a version. Thus, the version number is known at compile time as a symbolic expression, which helps further parallelization.

```
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

Figure 15. Example of AD, where `t.tape` means the intermediate value materialized in the forward pass.

5.2 Selective Intermediate Tensor Materialization

Directly transforming the AST does not always result in expected performance, and we observe a significant performance degradation resulting from materializing intermediate tensors. In many AD tools, all intermediate tensors are materialized, which will introduce significant overhead in both memory usage and access. We use an example to illustrate it.

In Figure 15, `t` is stored in the original program until reused with `t.tape`. In the example above, `t` is a scalar, probably stored in cache or even registers. However, as `t`

is updated for every iteration `i`, the forward pass would materialize all its versions for future usage. Therefore, the materialized tensor of `t`, i.e., `t.tape`, comes with a large shape `n`, which incurs a great overhead, both on memory footprint and memory usage.

On the contrary, computing `t` requires only one addition and two accesses to `a[i]` and `b[i]` in this case, where `a[i]` and `b[i]` are well cached because it is shared when computing `a.grad[i]` and `b.grad[i]`. Inspired by a recomputing approach that is previously proposed to reduce memory usage for operator-based frameworks [13], we propose a selective strategy for intermediate tensor materialization.

The core idea of our approach is that we will determine whether an intermediate tensor will be materialized or recomputed at compile time. We balance the overhead between materialization and recomputing and select a suitable strategy. For the program shown in Figure 15, we adopt recomputing strategy by its non-intensive computing. The resulting program is shown in Figure 15(c). The overhead of materializing a tensor depends on how many versions the tensor has. `t` in Figure 15 is updated `n` times so it has to be materialized in `n` versions, where the number of the versions is known at compile time thanks to our symbolic version number analysis.

6 Evaluation

6.1 Experimental Setup

Platforms. We evaluate FreeTensor on a server with dual 12-core Intel Xeon CPU E5-2670 v3 processors (hyper-thread enabled) and an NVIDIA Tesla V100-PCIE with 32GB GPU memory. The major compilers and libraries leveraged by FreeTensor and other baselines include Python 3.8.6, GCC 8.4.0, CUDA 10.2.89, MKL [26] 2020.3.279, and cuBLAS [16] 10.2.89.

Comparisons. We compare FreeTensor with PyTorch [29] 1.8.1 and JAX [17] 0.2.19 for traditional operator-based frameworks, TVM [12] commit c7a01a4 (Nov 4, 2021) for customizable operation-based frameworks, Julia [9] 1.6.3 for tensor-oriented general-purpose languages and DGL [43] 0.7.1 for a dedicated framework on Graph Neural Networks.

Workloads. We evaluate our language and compiler with the following applications: SubdivNet [19], Longformer [8], SoftRas [22], and GAT [38].

SubdivNet is a Convolution Neural Network for predicting properties of 3-D objects represented in meshes, which is described in Section 2. A SubdivNet is built with a chain of Mesh Convolution layers. Each layer renews the features of faces in the mesh by aggregating their neighbor faces with fine-grained tensor computations.

Longformer is an Attention-based Neural Network for natural language processing, which adopts an improved Attention layer. Instead of computing correlations for all tokens against all tokens, Longformer computes correlations only for pairs of nearby tokens that have a distance no greater than a threshold and thus it is capable to process much longer sequences. The improved Attention consists of fine-grained control flows.

SoftRas is a differentiable renderer for 3-D objects in meshes. In SoftRas, the rendering procedure from a 3-D object to an image is modeled as a continuous function to enable differentiation. Soft Rasterizer is the major component in SoftRas, which includes fine-grained computation on every pixel-face pair to determine their geometric relationship.

GAT is a Graph Neural Network model for predicting properties of a graph. A GAT is built with a chain of layers. Each layer renews the features of a node in the graph by combining their neighbor nodes by fine-grained tensor computations.

6.2 End-to-End Performance

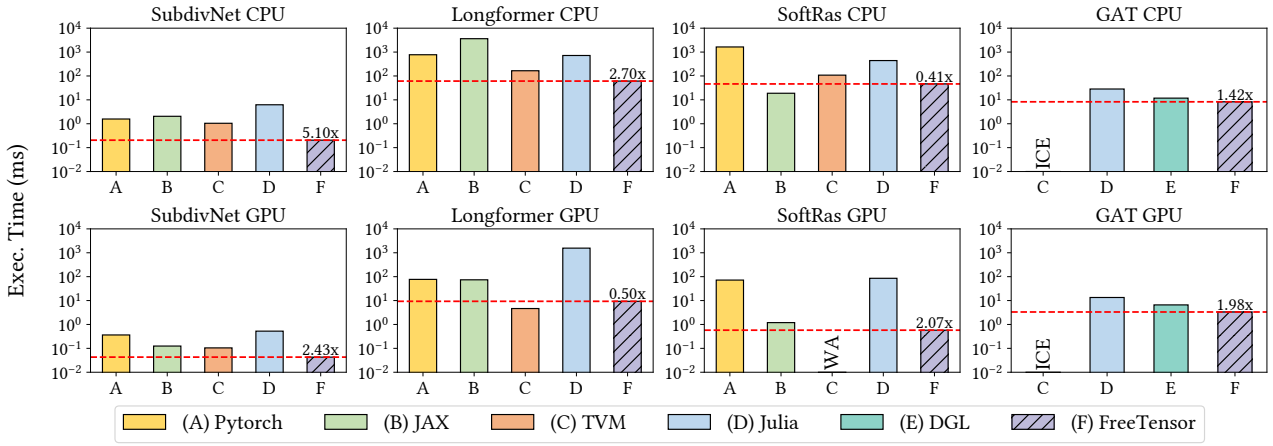
We compare the execution time of each case in Figure 16(a), and the gradient programs generated by AD in Figure 16(b), on both CPU and GPU backends. Each result is measured by averaging 100 repeated runs after 10 warming-up runs[1]. We use random input for the test, and compare the output among each implementation.

We implement each case in PyTorch and JAX purely with existing operators, without custom plugins. For TVM, we implement most computations as custom operators to avoid redundant computation and achieve better performance, but fall back to combining existing TVM operators if it cannot be expressed by TVM tensor expression. We tune with Ansor [44] (the auto-scheduler in TVM) by at most 1000 rounds per operator for each case, if possible. Julia requires human decisions to implement a performant program. We make the following decisions, which we consider typical for a common application programmer: In Julia, a program can either be implemented by fine-grained control flows or by operators. We prefer the former for its low redundancy, but it requires manual parallelization. Because parallelization is non-trivial on GPUs, and because parallelization after AD is unsupported in Julia, we only run the CPU-without-differentiation case with fine-grained control flows, and we run other cases with pure operators. Since GAT requires special operators, we do not compare with PyTorch or JAX in this case, but turn to a dedicated framework DGL. TVM provides such an operator, but fails to build GAT due to the complex indirect memory access in the computation.

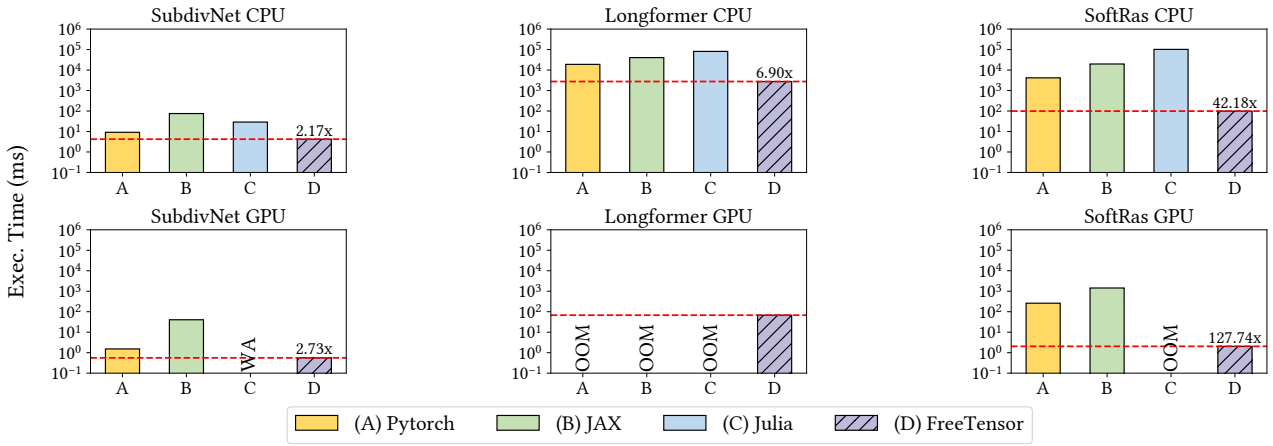
For gradient programs, we report the total time of the forward pass and the backward pass[2], where some variables are saved in the forward pass until reused in the backward pass. TVM does not support AD. Although Julia supports

[1] For cases falling back to sequential programs, we repeat them for 20 times in the evaluation since their execution time is stable and long.

[2] The outputs of the programs are tensors, but JAX and Julia require to compute gradients with respect to a scalar, so we sum the output to compute it, where the overhead is neglectable.



(a) Time without differentiation



(b) Time with differentiation, each including a forward pass and a backward pass

Figure 16. End-to-end time with or without differentiation. ICE (Internal Compiler Error) means the compiler or framework crashes when compiling. OOM (Out of Memory) means the program compiles but is unable to run for consuming too much memory. WA (Wrong Answer) means the parallelized or differentiated program outputs a wrong result, even though the serial non-differentiated program written by users is correct, which reveals a bug in the corresponding baseline.

programming in fine-grained control flows, such a program cannot be parallelized because Julia performs AD in an SSA IR which is not exposed to ordinary users. Therefore, we implement all cases in operators. Since the gradient program of GAT is non-trivial which requires preprocessing a sparse matrix, we do not report the gradient time of GAT.

Without differentiation. Our speedup over the best baseline for each case is up to 5.10×, and 2.08× on average, without differentiation.

SubdivNet can hardly be represented in any of the baselines without redundancy, so we are consistently faster.

Longformer is hard to be implemented in traditional operator-based frameworks without redundancy, but we can implement its sliding windows access as perfect nested loops in TVM. However, we still have to combine other operators including `softmax`. We achieve better performance in all cases except for comparing with TVM on a GPU.

SoftRas includes complex geometric computations, which requires combining a large number of operators in an operator-based framework. Fortunately, in JAX and PyTorch, this application can be accelerated by expressing the computation for individual faces and looping over multiple faces via the `vmap` meta-operator provided in the two frameworks. Taking this into comparison, we still achieve better performance in all cases except for comparing with JAX on CPUs.

For GAT, we achieve better performance even over DGL, which is a dedicated framework for Graph Neural Networks, because we can implement more computations in fewer kernels. Comparing with Julia, although we use fine-grained control flow for CPU cases, we achieve better performance because we are able to apply more optimizations.

With differentiation. For gradient programs, our speedup over the best baseline for each case is up to 127.74×, and

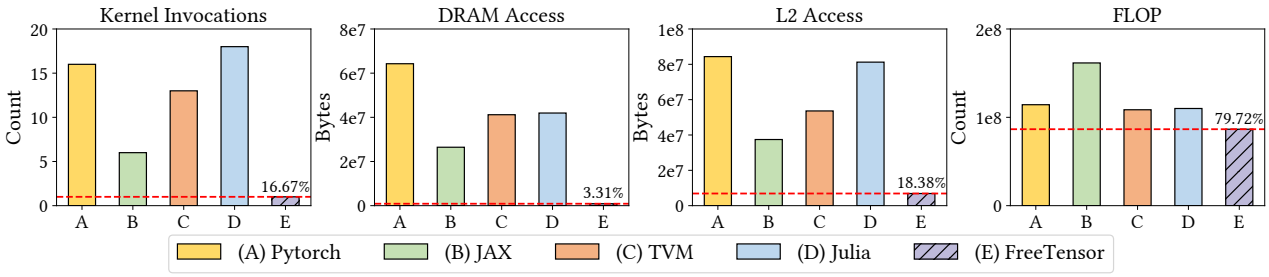


Figure 17. Analysis of the speedup of SubdivNet GPU. The metrics refer to the number of GPU kernel invocations, the total bytes of access to GPU DRAM and L2 cache, and the FLOP count, respectively.

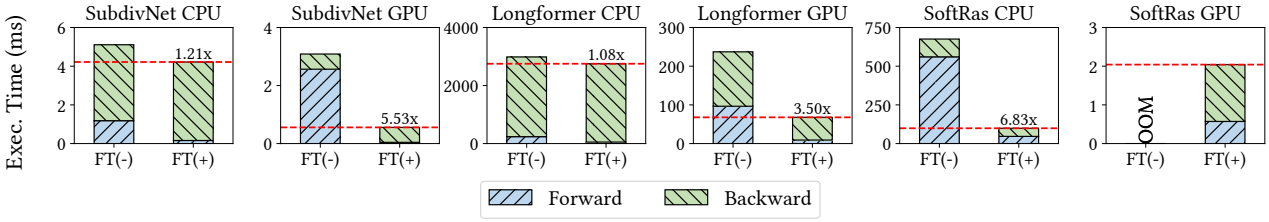


Figure 18. Time of FreeTensor without Selective Intermediate Tensor Materialization (FT(-)), and with Selective Intermediate Tensor Materialization (FT(+)). The time is broken down into a forward pass and a backward pass. OOM (Out of Memory) means the program is unable to run for consuming too much memory.

36.26× on average, where FreeTensor outperforms all baselines. Julia runs for an extremely long time in CPU cases because it falls backs to single-thread execution for many operators. All baselines fail to run Longformer on GPU since the tight GPU memory limitation, but FreeTensor successfully executes it. The recomputing mechanism in FreeTensor allows us to store much fewer intermediate tensors than the baselines, which not only results in a better performance, but also enables FreeTensor to run with a limited memory capacity (32GB on a GPU).

6.3 Analysis of the Speedup

We further profile SubdivNet without differentiation case running on a GPU to understand the reasons of our speedup. As shown in Figure 17, FreeTensor is able to run the case with only one GPU kernel invocation. This is achieved by supporting irregular computation, which enables users to implement the program as a whole. On the contrary, the baselines require chaining multiple operators, which leads to no less than 6 kernel invocations.

By using fewer operators, the memory footprint is also reduced to only 3.31% on DRAM and 18.38% on L2 compared to the best baseline. The reason is that intermediate results can now be kept in registers, shared memory or cache, while the baselines require storing them back to global memory between operators.

As the result also shows, FreeTensor is even able to reduce FLOP counts to 79.72% compared to the baseline, though we do not apply any algorithmic optimizations. A potential reason is that implementing a program with only one GPU kernel reveals more opportunities for backend compilers (nvcc) to apply arithmetic optimizations like Common Subexpression Elimination.

Profiling on the other cases shows similar results.

6.4 Optimization for AD

We analyze our Selective Intermediate Tensor Materialization introduced in Section 5.2 for AD. As Figure 18 shows, compared to materializing all intermediate tensors, our Selective Intermediate Tensor Materialization contributes 1.21× to 6.83× speedup, and prevents one of the cases from running out of memory.

In particular, we can observe a significant speedup in a forward pass, and in some cases, a moderate speedup in a backward pass. For any tensor that our algorithm 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.

6.5 Compiling Time

We compare the compiling time used to compile the program in Figure 16(a) between FreeTensor and TVM, as shown in Table 2. We report the end-to-end compiling time including the

auto-transforming time in FreeTensor, and the auto-tuning time in TVM. We also report the total tuning rounds for the multiple operators in TVM and the average tuning time of each round. For each operator, TVM needs to tune multiple rounds to reach an acceptable performance, resulting in an extremely lengthy procedure. Since TVM cannot tune a computation including indirect memory access, TVM has to divide the application into multiple operators and tune them separately, which further extends the compiling time. With only 0.13% to 22.92% compiling time of TVM, FreeTensor generates faster code on most of the evaluated applications.

Table 2. Compiling time of FreeTensor and TVM. Time of FreeTensor includes auto-transforming. Time of TVM includes auto-tuning, where the tuning rounds and the time per round are marked in parentheses. ICE means Internal Compiler Error.

	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

7 Related Works

Operator-based frameworks. There are multiple operator-based frameworks including Chainer [34], PyTorch [29], MXNet [11], TensorFlow [3], JAX [17], and TVM [12]. Chainer and PyTorch run as high-performance tensor operator libraries, which can be invoked imperatively from Python. MXNet and TensorFlow transform a program to a dataflow graph, where each node represents a call to a tensor operator in a library. Optimizations can be performed on the graph before execution. JAX improves optimization by introducing Just-in-Time compilation to enable optimizations for complex or dynamic programs. TVM supports highly customized operators by introducing a compute-and-schedule programming model, where users first specify the mathematical definition of computation and then optimize it with explicit or machine-learning-guided transformations [14].

XLA [1], TensorRT [2], TASO [21], Rammer [24], and PET [42] optimize tensor programs by re-combining tensor operators. Comparing with these works, we try not to introduce too many operators in the first place.

Compilers based on polyhedral analysis. Multiple compilers adopt optimizations based on Polyhedral Analysis. Pluto [10], PPCG [41], and CHiLL [32] are optimizing compilers for general programs in C language. PPCG designs an analytical cost model and performs optimization by solving an analytical model, while CHiLL implements transformations specified by users that are guided by polyhedral analysis.

Tensor Comprehensions [37] and Tiramisu [5] introduce polyhedral analysis to tensor programs. They improve operator-based framworks by optimizing their existing operators with polyhedral analysis. We adopt polyhedral analysis to guide our AST transformations on user-defined irregular tensor programs.

Tensor-oriented design in general-purpose programming languages. There are also some improvements in general-purpose programming language for better supports on tensors. Julia [9] provides efficient support on tensors, where consecutive calls to tensor operations can be automatically fused with macros. Triton [33] improves CUDA and provides a tiled programming model for implementing tensor operations on a GPU.

Automatic differentiation. There are several ways to implement automatic differentiation [7, 35]. AD implemented by most operator-based frameworks is based on graphs, where a node represents a call to a tensor operation library, and an edge represents a tensor [25, 29]. The AD process replaces all the nodes to their gradient counterpart and reverses the order of the graph using the Chain Rule.

Tangent [36], Myia [35], Enzyme [27], and Zygote [20] implement AD for general tensor programs by directly transforming IR. We adopt this type of techniques in FreeTensor, and further resolve its performance issues.

8 Conclusion

We propose FreeTensor, a free-form DSL for irregular tensor programs. FreeTensor supports granularity-oblivious tensor operations by enabling fine-grained control flows, and integrates a series of optimizations, including partial evaluation, dependence-aware transformation, and automatic code generation, to generate high-performance code for different architectures. FreeTensor also supports fine-grained automatic differentiation to generate efficient gradient programs. Experiments show a speedup over existing tensor programming frameworks up to 5.10× (2.08× on average) for without differentiation, and up to 127.74× (36.26× on average) after differentiation.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Raghu Prabhakar, for their valuable comments and suggestions. This work is supported by National Key R&D Program of China under Grant 2021ZD0110202, National Natural Science Foundation of China (U20A20226), and Beijing Natural Science Foundation (4202031).

References

[1] 2017. XLA: Optimizing Compiler for TensorFlow. https://www.tensorflow.org/xla.

[2] 2021. Nvidia TensorRT Documentation. https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html.

[3] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[4] Randy Allen and Ken Kennedy. 2001. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann.

[5] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman P. Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*, Mahmut Taylan Kandemir, Alexandra Jimborean, and Tipp Moseley (Eds.). IEEE, 193–205. https://doi.org/10.1109/CGO.2019.8661197

[6] Roberto Bagnara, Elisa Ricci, Enea Zaffanella, and Patricia M. Hill. 2002. Possibly Not Closed Convex Polyhedra and the Parma Polyhedra Library. In *Static Analysis, 9th International Symposium, SAS 2002, Madrid, Spain, September 17-20, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2477)*, Manuel V. Hermenegildo and Germán Puebla (Eds.). Springer, 213–229. https://doi.org/10.1007/3-540-45789-5_17

[7] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2017. Automatic Differentiation in Machine Learning: a Survey. *J. Mach. Learn. Res.* 18 (2017), 153:1–153:43. http://jmlr.org/papers/v18/17-468.html

[8] Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The Long-Document Transformer. *CoRR* abs/2004.05150 (2020). arXiv:2004.05150 https://arxiv.org/abs/2004.05150

[9] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59, 1 (2017), 65–98. https://doi.org/10.1137/141000671

[10] Uday Bondhugula, Albert Hartono, J Ramanujam, and P Sadayappan. 2008. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer.

[11] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* abs/1512.01274 (2015). arXiv:1512.01274 http://arxiv.org/abs/1512.01274

[12] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: End-to-End Optimization Stack for Deep Learning. *CoRR* abs/1802.04799 (2018). arXiv:1802.04799 http://arxiv.org/abs/1802.04799

[13] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. *CoRR* abs/1604.06174 (2016). arXiv:1604.06174 http://arxiv.org/abs/1604.06174

[14] Tianqi Chen, Lianmin Zheng, Eddie Q. Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 3393–3404. https://proceedings.neurips.cc/paper/2018/hash/8b5700012be65c9da25f49408d959ca0-Abstract.html

[15] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *CoRR* abs/1410.0759 (2014). http://arxiv.org/abs/1410.0759

[16] cuBLAS 2016. Dense Linear Algebra on GPUs. https://developer.nvidia.com/cublas.

[17] Roy Frostig, Matthew James Johnson, and Chris Leary. 2018. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning* (2018).

[18] Charles R. Harris, K. Jarrod Millman, Stéfan van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nat.* 585 (2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[19] Shi-Min Hu, Zheng-Ning Liu, Meng-Hao Guo, Junxiong Cai, Jiahui Huang, Tai-Jiang Mu, and Ralph R. Martin. 2021. Subdivision-Based Mesh Convolution Networks. *CoRR* abs/2106.02285 (2021). arXiv:2106.02285 https://arxiv.org/abs/2106.02285

[20] Michael Innes. 2018. Don't Unroll Adjoint: Differentiating SSA-Form Programs. *CoRR* abs/1810.07951 (2018). arXiv:1810.07951 http://arxiv.org/abs/1810.07951

[21] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 47–62.

[22] Shichen Liu, Weikai Chen, Tianye Li, and Hao Li. 2019. Soft Rasterizer: A Differentiable Renderer for Image-Based 3D Reasoning. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*. IEEE, 7707–7716. https://doi.org/10.1109/ICCV.2019.00780

[23] Vincent Loechner. 1999. PolyLib: A library for manipulating parameterized polyhedra.

[24] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 881–897. https://www.usenix.org/conference/osdi20/presentation/ma

[25] Dougal Maclaurin, David Duvenaud, and Ryan P Adams. 2015. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML workshop*, Vol. 238. 5.

[26] MKL 2003. Intel(R) oneAPI Math Kernel Library. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html.

[27] William S. Moses and Valentin Churavy. 2020. Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/9332c513ef44b682e9347822c2e457ac-Abstract.html

[28] David A. Padua and Michael Wolfe. 1986. Advanced Compiler Optimizations for Supercomputers. *Commun. ACM* 29, 12 (1986), 1184–1201. https://doi.org/10.1145/7902.7904

[29] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[30] William Pugh. 1991. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings Supercomputing '91, Albuquerque, NM, USA, November 18-22, 1991*, Joanne L. Martin (Ed.). ACM, 4–13. https://doi.org/10.1145/125826.125848

[31] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 519–530. https://doi.org/10.1145/2491956.2462176

[32] Michelle Mills Strout, Mary W. Hall, and Catherine Olschanowsky. 2018. The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code. *Proc. IEEE* 106, 11 (2018), 1921–1934. https://doi.org/10.1109/JPROC.2018.2857721

[33] Philippe Tillet, Hsiang-Tsung Kung, and David D. Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2019, Phoenix, AZ, USA, June 22, 2019*, Tim Mattson, Abdullah Muzahid, and Armando Solar-Lezama (Eds.). ACM, 10–19. https://doi.org/10.1145/3315508.3329973

[34] Seiya Tokui, Ryosuke Okuta, Takuya Akiba, Yusuke Niitani, Toru Ogawa, Shunta Saito, Shuji Suzuki, Kota Uenishi, Brian Vogel, and Hiroyuki Yamazaki Vincent. 2019. Chainer: A Deep Learning Framework for Accelerating the Research Cycle. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, Ankur Teredesai, Vipin Kumar, Ying Li, Rómer Rosales, Evimaria Terzi, and George Karypis (Eds.). ACM, 2002–2011. https://doi.org/10.1145/3292500.3330756

[35] Bart van Merrienboer, Olivier Breuleux, Arnaud Bergeron, and Pascal Lamblin. 2018. Automatic differentiation in ML: Where we are and where we should be going. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 8771–8781. https://proceedings.neurips.cc/paper/2018/hash/770f8e448d07586afbf77bb59f698587-Abstract.html

[36] Bart van Merriënboer, Alexander B. Wiltschko, and Dan Moldovan. 2017. Tangent: Automatic Differentiation Using Source Code Transformation in Python. *CoRR* abs/1711.02712 (2017). arXiv:1711.02712 http://arxiv.org/abs/1711.02712

[37] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018). arXiv:1802.04730 http://arxiv.org/abs/1802.04730

[38] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=rJXMpikCZ

[39] Sven Verdoolaege. 2010. *isl*: An Integer Set Library for the Polyhedral Model. In *Mathematical Software - ICMS 2010, Third International Congress on Mathematical Software, Kobe, Japan, September 13-17, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6327)*, Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama (Eds.). Springer, 299–302. https://doi.org/10.1007/978-3-642-15582-6_49

[40] Sven Verdoolaege. 2016. Presburger formulas and polyhedral compilation. (2016). https://doi.org/10.13140/RG.2.1.1174.6323

[41] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4 (2013), 54:1–54:23. https://doi.org/10.1145/2400682.2400713

[42] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. 2021. PET: Optimizing Tensor Programs with Partially Equivalent Transformations and Automated Corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, Angela Demke Brown and Jay R. Lorch (Eds.). USENIX Association, 37–54. https://www.usenix.org/conference/osdi21/presentation/wang

[43] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J. Smola, and Zheng Zhang. 2019. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. *CoRR* abs/1909.01315 (2019). arXiv:1909.01315 http://arxiv.org/abs/1909.01315

[44] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 863–879. https://www.usenix.org/conference/osdi20/presentation/zheng