A Holistic Functionalization Approach to Optimizing Imperative Tensor Programs in Deep Learning

Jinming Ma[1,*], Xiuhong Li[2,*], Zihan Wang[3], Xingcheng Zhang[1,4,#], Shengen Yan[2], Yuting Chen[3], Yueqian Zhang[1], Minxi Jin[1], Lijuan Jiang[1], Yun (Eric) Liang[2], Chao Yang[2], Dahua Lin[1,5]
[1]Shanghai Artificial Intelligence Laboratory, [2]Peking University, [3]Shanghai Jiao Tong University, [4]SenseTime Research, [5]The Chinese University of Hong Kong
{majinming,zhangxingcheng}@pjlab.org.cn,lixiuhong@pku.edu.cn

Abstract

As deep learning empowers various fields, many domain-specific non-neural network operators have been proposed to improve the accuracy of deep learning models. Researchers often use the imperative programming diagram (PyTorch) to express these new operators, leaving the fusion optimization of these operators to deep learning compilers. Unfortunately, the inherent side effects introduced by imperative tensor programs, especially tensor-level mutations, often make optimization extremely difficult. Previous works either fail to eliminate the side effects of tensor-level mutations or require programmers to manually analyze and transform them. In this paper, we present a holistic functionalization approach (*TensorSSA*) to optimizing imperative tensor programs beyond control flow boundaries. We first introduce *TensorSSA* intermediate representation for removing tensor-level mutation and expanding the scope and ability of operator fusion. Based on *TensorSSA* IR, we propose a *TensorSSA* conversion algorithm that performs functionalization crossing the boundary of control flow. *TensorSSA* achieves a 1.79X (1.34X on average) speedup in representative deep learning tasks than state-of-the-art works.

Keywords: Program Functionalization, Deep Learning Compiler

1 Introduction

In recent years, deep neural networks have brought significant innovations in many domains. Deep learning owes much of its success to the high-performance compute libraries of the frequently used neural network operators, such as convolution and matrix multiplication. However, with the growing adoption of deep learning algorithms in various applications, developers have created new operators to meet the specific requirements of these applications and achieve higher accuracy. Due to a lack of expertise in device code development, algorithm designers often prefer using imperative tensor programming diagrams such as PyTorch and delegate the code generation and optimization to deep learning compilers. These new operators typically consist of a sequence of imperative statements that update the program's running state and have become a significant portion of computing time in real-world deep learning applications. The imperative tensor programs account for up to 90% of the total end-to-end wall duration time in the inference process of several widely used deep learning models, across a range of NVIDIA GPUs from both consumer and data center perspectives.

Kernel fusion is a widely used optimization in deep learning compilers [15], but the effectiveness is limited due to the challenges posed by the inherent side effects in imperative programs, especially tensor-level mutations. Instead of generating new tensors, some operators always modify tensor data in place. Traditional compiler technologies solve the value mutation by static single assignment (SSA) [6]. However, they only target the scalar scenario and lack support at the tensor level. The domain-specific compiler infrastructure (MLIR [12]) and tensor expression-level deep learning compilers like TVM [4], Halide [17], and PyTorch NNC [16], need the program satisfy that the tensors in expression are immutable, which limits the application in imperative tensor programs. Graph-level deep learning compilers like TorchScript [7] always treat operators with tensor-level mutation as graph-breaking points. Functorch [9] and EasyView [11] are devoted to solving tensor mutations, but they are only reliable in pure data-flow computation graphs.

In this paper, we present a holistic functionalization approach to optimizing imperative tensor programs in deep learning. Our approach is based on the insight that by transforming imperative tensor programs into their pure functional forms (i.e., *functionalization*), we can explore a larger kernel fusion optimization space than the previous methods[7, 19]. Our functionalization approach is holistic, meaning that it enables analysis and transformation beyond control flow boundaries. To achieve this, we first introduce a key intermediate representation (IR): *TensorSSA* IR, for the effective functional representative of mutation of tensors. Then, based on *TensorSSA* IR, we propose a comprehensive tensor-level data flow and control flow analysis to enable efficient functionalization.

In summary, this paper makes the following contributions:

1. We design *TensorSSA* IR, an equivalent, immutable, and more compiler-friendly representation for tensor-level mutation.
2. To the best of our knowledge, *TensorSSA* [1] is the first to propose a holistic functionalization approach to optimizing imperative tensor programs in deep learning.
3. *TensorSSA* gets better performance than state-of-the-art deep learning compilers for many imperative tensor programs.

The evaluation shows *TensorSSA* outperforms multiple state-of-the-art deep learning compilers on eight representative deep learning tasks with imperative tensor programs, with up to 1.79X (1.34X on average) speedup from both consumer and data center GPUs.

[*] Equal contribution.
[#] Corresponding author.

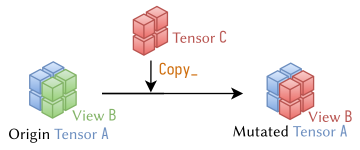
[1]The source code is available here. https://github.com/JimyMa/FuncTs

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

DAC '24, June 23–27, 2024, San Francisco, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0601-1/24/06…$15.00
https://doi.org/10.1145/3649329.3658483

```
B = A[0, ...]
B.copy_(C)
```

(a) Python Code



(b) Execution process

Figure 1. Features of imperative tensor programs.

2 Background

2.1 Tensor-Level Mutation

The imperative programming paradigm is broadly used in deep learning frameworks including PyTorch [16] because of its high development efficiency and lower barrier of learning and understanding. For the imperative paradigm, a step-by-step process is described for executing a program, while being aware of its side effects including mutation of variables. Some parts of deep learning tensor programs, notably the backbones of neural networks, can easily be captured as functional forms by tracing [19, 25], as they only contain data flow among pure functional operators. We do not consider them imperative programs from the perspective of deployment and compilation. However, the domain-specific part, such as the post-processing of computer vision models, cannot be captured as functional programs. The reason is that these imperative tensor programs always contain mutable tensors.

Tensor views are prevalent in imperative tensor programs, which allow a tensor to share the same underlying data with another tensor while having a different interpretation of the data. If a view of a tensor is modified by an in-place operator, the referenced tensor will be implicitly mutated. Figure 1 (a) depicts a Python code snippet of a mutable tensor and Figure 1 (b) depicts the execution process of the code. `B` is defined as a tensor view of `A`. The data of `B` is replaced with `C` using `copy_`. Due to data sharing between `A` and `B`, the referenced tensor `A` is mutated along with view `B` simultaneously.

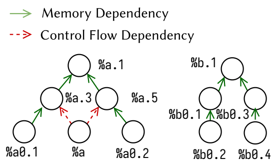
2.2 Graph-Level IR and Optimization

In general, deep learning compilers contain two levels of intermediate representation: graph-level IR and tensor-level IR. The former describes the complete procedure of a deep learning algorithm, while the latter expresses the computation details of operators. This paper mainly focuses on the optimization of graph-level IR. TorchScript [7] is one of the popular graph-level IRs, which is converted from an imperative PyTorch program. The control flow like loop and branch can be represented as special nodes with return values, arguments, and block body. Note that TorchScript uses a functional form of SSA [1] where dependent values are passed as block arguments to successor blocks, which is equivalent to the conventional SSA form with ϕ nodes. In Figure 2, variable `a` in Python code is mutated as a whole entity inner control flow, and variable `b` is partially mutated inner the control flow. As a result, only `a` is added to block returns inner the control flow nodes after SSA transformation. This is because the SSA process in mainstream deep learning compilers like TorchScript is incapable of executing transformations and optimizations for the partial mutation caused by tensor views. This precisely aligns with the objective we seek to address.

Deep learning compilers often employ operator fusion as a prevalent optimization technique to generate high-performance device kernels [4, 7, 12]. However, the effectiveness of fusion in imperative

```
def func(a: Tensor, b: Tensor):
  if idx ≥ 0:
    a += 1
    b[0] = a[0]
  else:
    a -= 1
    b[1] = a[1]
  return a + b
```

(a) Python Code



(c) Alias Graph

```
graph(%a.1 : Tensor, %b.1 : Tensor,
      %idx.1 : int):
  %a = prim::If(%idx ≥ 0)
    block0():
      %a.2 = %a.1 + 1
      %a.3 = aten::copy_(%a.1, a.2)
      %b0.1 = aten::select(%b.1, 0, %idx)
      %a0.1 = aten::select(a.3, 0, %idx)
      %b0.2 = aten::copy_(%b0.1, %a0.1)
      → (%a.3)
    block1():
      %a.4 = %a.1 - 1
      %a.5 = aten::copy_(%a.1, %a.4)
      %b0.3 = aten::select(%b.1, 1, -%idx)
      %a0.2 = aten::select(%a.5, 1, -%idx)
      %b0.4 = aten::copy_(%b0.3, a0.2)
      → (%a.5)
  %64 : Tensor = aten::add(%a, %b.1)
  return (%64)
```

(b) Graph-Level IR

Figure 2. An Example of Alias Graph.

tensor programs is limited due to the presence of tensor-level mutations and control flow. These operators restrict the fusion scope because of the potential implicit side effects caused by tensor-level mutations and control flow further aggravates the difficulties.

2.3 Alias Analysis

Given a segment of graph-level IR captured from an imperative tensor program, the dependencies between variables can be obtained by alias analysis. The result can be represented by a series of points-to edges in a directed acyclic alias graph. If a variable is mutated by a statement, the variables connected to it may implicitly be mutated at the same time. Given a points-to edge `p` ⟶ `q` in an alias graph of an intra-procedure graph-level IR program, there are three main types of dependency:

1. Control-flow dependency: `p` is a block *argument* of `q`, or `q` is a block *return* of `p`.
2. Memory dependency: `p` is a *view* of `q`, e.g., `p` = `q[i]`.
3. Container dependency: a compound data structure `q` *contains* variable `p`, e.g., `q` = `[p]`.

In a complete alias graph, `p` *must* alias `q` if `p` points to a single variable `q`; otherwise, we say `p` *may* alias `q`.

Figure 2 (a), (b), and (c) show the Python code, the corresponding graph-level IR, and the alias graph respectively. For the sub-graph formed by the origin tensor `%a.1`, it has memory dependency by view statements, and there are also control-flow dependencies (`%a` ⟶ `%a.3`, `%a` ⟶ `%a.5`) due to block returns in both the true block (`%a.2`) and the false block (`%a.5`). For the sub-graph formed by the tensor `%b.1`, although both the if block and the then block modify the data stored in `%b.1`, there are no arguments or returns related to `%b.1` in either block, indicating the absence of control flow dependency. In this paper, we mainly deal with the sub-graphs which solely consist of *memory dependencies*. In this situation, the relationships of variables among these sub-graphs are must-alias because the dependencies generated by view operators have a single points-to edge.

3 Representation for TensorSSA

As is mentioned in Section 2.1, tensor-level mutations involved in imperative tensor programs limit the optimization scope. To eliminate the mutation, we introduce our key representation, *TensorSSA*, which eliminates mutation of tensors by in-place operators, for effective functionalization in imperative tensor programs.

In the imperative tensor program, the mutable tensor has two distinct meanings: a tensor can be mutated as a whole entity, performing *whole* mutation; or a tensor can be mutated in tensor view form, performing *partial* mutation. To provide a complete solution, we claim that we can address both cases. While the former can be addressed using scalar SSA techniques [6], we focus on the latter, where the mutation of tensor views can be a significant bottleneck in optimizing imperative tensor programs. *TensorSSA* is designed to handle complex mutation effects introduced by diverse forms of tensor views. As a result, the range of operator fusion can safely include tensor views. In this section, we first provide a unified definition of tensor views and mutations and then introduce the design of the *TensorSSA* IR.

3.1 Tensor View and Mutation

Definition 3.1 (Tensor View)**.** A view v of tensor x is formally defined as $v \leftarrow x[\cdot]$. x is called the *base tensor* whose underlying data is shared. $[\cdot]$ is the abstract `View` operator, which can possibly be one concrete `View` operator or a chain of `View` operators. Specially, we say $v \Leftarrow x[\cdot]$ if $[\cdot]$ is a concrete `View` operator.

The view operators are very common in imperative tensor programs, such as `select`, `slice`, `reshape`, `index`, and `permute`. As introduced in Section 2.3 and according to Definition 3.1, the `View` operator brings out a points-to edge of memory dependency $v \longrightarrow x$ in alias graph given $v \leftarrow x[\cdot]$.

Definition 3.2 (Tensor View Mutation)**.** A tensor view mutation operator

$$\mathtt{Mutate}(v, w)$$

represent that the data in view (tensor) v are mutated by w.

The $\mathtt{Mutate}(v, w)$ is an operator with tensor-level side-effect. The states of v and all of the variables connected with v directly or indirectly in the alias graph may be changed by w simultaneously.

Imperative deep learning frameworks allow tensor views to be mutated by `Mutate` operators like `copy_`, `sigmoid_` and `mul_` (underscore indicates in-place update) just as concrete tensors. The combination of tensor views and mutations often leads to partial tensor mutation and further complicates the functionalization of the programs.

3.2 IR Design

TensorSSA aims to eliminate mutation in programs while keeping the semantics of tensor updates. The key idea can be divided into two parts. The first is to rewrite `View` and `Mutation` operators into their corresponding immutable operators, which guarantees the *single assignment*. The second is to perform functionalization crossing control flow by block propagation [1], which guarantees the *static assignment*. For the first part, we introduce tensor access and tensor assign, the immutable version of the tensor view and mutation, respectively. For the second, we propose a new operator, `Update`, to record the lost semantics during program transformation and guide the block propagation.

3.2.1 Immutable Version of Tensor View and Mutation.

We use tensor access and tensor assign, which are two operator sets to substitute tensor view and mutation. The access operator set contains the immutable version of `View` operators. The assign operator set contains the immutable version of the `Mutate` operators.

Definition 3.3 (Tensor Access)**.** The tensor access is a set of operators with the following form:

$$v' = \mathtt{Access}(v, [\cdot])$$

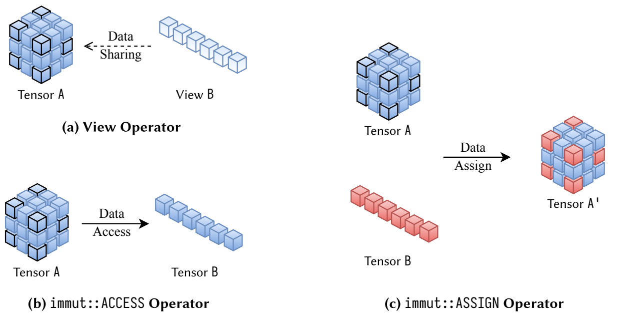


Figure 3. Immutable Operators vs. View Operators

where v is the *base tensor*, $[\cdot]$ is the rules of the data access. v' is a new tensor version generated according to $[\cdot]$.

Definition 3.4 (Tensor Assign)**.** The tensor assign is a set of operators with the following form:

$$v' = \mathtt{Assign}(v, s, [\cdot])$$

which means substituting the data obtained by $\mathtt{Access}(v, [\cdot])$ with source tensor s and generate a new version of tensor v'.

Figure 3 depicts examples of tensor view, access, and assign. The same point between `View` and `Access` operators is that both two operators access the data by the rule $[\cdot]$. However, the `Access` operator takes input tensor A as immutable and generates a tensor B with new storage allocation. The `Assign` operator allocates a new storage owned by tensor A' which combines the data in tensor A with the data in tensor B according to $[\cdot]$.

3.2.2 Tensor Update.

Like ϕ node in traditional SSA form [6], an operator that is able to handle the version updated by tensor mutation needs to be defined. The `Update` operator serves as an annotation and carries no computation semantics.

Definition 3.5 (Tensor Update)**.** The `Update` operator has the following form:

$$\mathtt{Update}(v', v)$$

which means all uses of v are replaced with v' after $\mathtt{Update}(v', v)$.

The `Update` Operator acts as a beacon in the program to guide the compiler to perform functionalization crossing control flow. With the combination of tensor `Access`, tensor `Assign`, and `Update`, *TensorSSA* is able to eliminate the mutation of tensor views safely. In addition, the greatest strength of *TensorSSA* lies in its flexibility, as the operators can either be fused and compiled or be converted back to the original mutable operators.

4 Functionalization and Optimization

In program functionalization, we perform tensor-level data flow and control flow analysis to transform the imperative tensor program into SSA form, which promotes kernel fusion in a larger scope than previous works.

4.1 Conversion to *TensorSSA* IR Form

Functionalization serves as a key technique in enabling efficient optimizations in imperative tensor programs. With *TensorSSA* IR introduced in Section 3, *TensorSSA* transforms the imperative tensor program to its functional equivalent. Before the beginning of the algorithm, we first select memory-dependent sub-graphs from the

Algorithm 1: *TensorSSA* conversion.

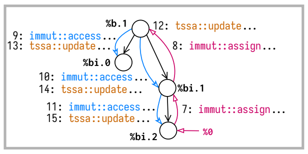
```
Input: Program $\mathcal{P}$ in graph-level IR, alias graph set $\mathcal{T}$.
Output: $\mathcal{P}$ in TensorSSA form.
1  def Traversal (x, x′):
2      Insert Update(x′, x);
3      foreach $V$ := v ⇐ x[·] do
4          if $V$ dominates $N$ then
5              Insert v′ = Access(x′, [·]);
6              Traversal(v, v′);
7  def RewriteMutation ($\mathcal{P}$, $\mathcal{T}$):
8      foreach $N$ := Mutate(v, w) ∈ $\mathcal{P}$ do
9          if $\exists T \in \mathcal{T}, N \in T.\mathcal{M}$ then
10             v′ := w;
               # Pass up
11             do
12                 For v ⇐ x[·], insert x′ = Assign(x, v′, [·]);
13                 v := x, v′ := x′;
14             while x ≠ $T$.t;
               # Pass down
15             Traversal(x, x′);
16             Remove $N$;
17 def BlockPropagation ($\mathcal{P}$):
18     foreach Update(x, x′) ∈ $\mathcal{P}$ do
19         $B$ := x′.def.owningBlock;
20         $B_{\text{end}}$ := x.def.owningBlock;
21         while $B \neq B_{\text{end}}$ do
22             Add x to $B$.returns;
23             $N$ := $B$.owningNode;
24             Add a new value $x_o$ to $N$.outputs;
25             Insert Update($x_o$, x);
26             if $N$ is a loop then
27                 Add x to $N$.inputs;
28                 Add a new value $x_p$ to $B$.params;
29                 Insert Update($x_p$, x);
30             else if $N$ is a branch then
31                 Add x to $\overline{B}$.returns if x is not mutated in $\overline{B}$;
32             $B$ := $N$.owningBlock;
       # Perform renaming
33     foreach Update(x′, x) ∈ $\mathcal{P}$ do
34         Replace all uses of x with x′ after Update(x′, x);
35     Remove all Update operators from $\mathcal{P}$;
36 RewriteMutation($\mathcal{P}$, $\mathcal{T}$);
37 BlockPropagation($\mathcal{P}$);
```

```
0: b = b.clone()
1: for i in range(n):
2:   b[i] = b[i] + 1
3: return b
```

(a) Origin Python Code

```
0: %b.1 : = aten::clone(%b.0)
1:  = prim::Loop(%n.1)
2:   block0(%i : int):
3:     %bi.0 = aten::select(%b.1, 0, %i)
4:     %0 = aten::add(%bi.0, 1)
5:     %bi.1 = aten::select(%b.1, 0, %i)
6:     %bi.2 = aten::copy_(%bi.1, %0)
7:     → ()
8: return %b.1
```

(b) Graph-Level IR



(c) Rewrite Mutation

```
0: %b.1 = aten::clone(%b.0)
1: %b.4 = prim::Loop(%n.1, %b.1)
2:   block0(%i : int, %b.3 : Tensor):
3:     tssa::update(%b.3, %b.1)
4:     %bi.0 = immut::select(%b.3, 0, %i)
5:     %0 = aten::add(%bi.0, 1)
6:     %bi.1 = immut::select(%b.3, 0, %i)
7:     %bi.2 = immut::assign(%bi.1, %0, [])
8:     %b.2 = immut::assign(%b.3, %bi.2, [0, %i])
9:     %bi.4 = immut::access(%b.2, [0, %i])
10:    %bi.5 = immut::access(%b.2, [0, %i])
11:    %bi.6 = immut::access(%bi.5, [])
12:    tssa::update(%b.2, %b.1)
13:    tssa::update(%bi.4, %bi.0)
14:    tssa::update(%bi.5, %bi.1)
15:    tssa::update(%bi.6, %bi.2)
16:    → (%b.2)
17:  tssa::update(%b.4, %b.1)
18:  return (%b.4)
```

(d) Block Propagation

```
0: %b.1 = aten::clone(%b.0)
1: %b.4 = prim::Loop(%n.1, %b.1)
2:   block0(%i : int, %b.3 : Tensor):
3:     %bi.0 = immut::select(%b.3, 0, %i)
4:     %0 = aten::add(%bi.0, 1)
8:     %b.2 = immut::assign(%b.1, %bi.3, [0, %i])
16:    → (%b.2)
18:  return (%b.4)
```

(e) Graph-Level IR in TensorSSA Form

Figure 4. TensorSSA Conversion Example

alias graph of the imperative tensor program in graph-level IR and rewrite it to a set, $\mathcal{T} = \{T_0, \dots T_n\}$. Each T has the following form:

$$T := (\mathsf{t}, \mathcal{V}, \mathcal{M}) \tag{1}$$

$$:= (\mathsf{t}, \{\mathsf{v} \mid \mathsf{v} \leftarrow \mathsf{t}[\cdot]\}, \{\mathsf{Mutate}(\mathsf{v}, \mathsf{w}) \mid \mathsf{v} \leftarrow \mathsf{t}[\cdot]\}). \tag{2}$$

t is the origin tensor that owns the memory storage. $\mathcal{V}$ is the set of all variables that lie in the reachability of t in the alias graph. $\mathcal{M}$ is all Mutate operators which mutate a view of t. Algorithm 1 presents the procedure of *TensorSSA* conversion which takes the imperative tensor program $\mathcal{P}$ and the alias sub-graph set $\mathcal{T}$ as inputs. The algorithm can be divided into two parts, *rewrite mutation* (line 1 - line 16) and *block propagation* (line 17 - line 35).

4.1.1 Rewrite Mutation. A Mutate statement actually modifies the storage owned by the source tensor T.t. According to the view mechanism, both $T.t$ and the variables in T.t share the same memory. Therefore, the Mutate statement indirectly modifies the value of v in v ← t[·] as well. To transform the view and mutation mechanisms into an equivalent single assignment form, RewriteMutation will cope with every Mutate statement. The process includes two key steps: *pass-up* and *pass-down*.

As any variable in $T.\mathcal{V}$ is the alias of a unique tensor T.t, there is a consistent view path v ⇐ x[·], x ⇐ ··· ⇐ t[·] given $v \in T.\mathcal{V}$. In the pass-up step, the algorithm traverses the view path from v to t. When each variable is visited, an x′ = Assign(x, v′, [·]) operator is inserted. After the traversal, a new version of T.t is generated.

In the pass-down step, we traverse all variables in $T.\mathcal{V}$ from t to v. For each variable v, the algorithm traverses along its view path (t[·] ⇒ ··· ⇒ x, x[·] ⇒ v). While each variable is firstly visited, an v′ = Access(x′, [·]) operator is inserted. To annotate the tensor version for subsequent block propagation, an Update(v′, v) statement is generated at the same time.

4.1.2 Block Propagation. The Function BlockPropagation propagates the tensor mutation beyond the control flow. The function visits all generated Update(x′, x). If x and x′ are defined in different blocks, the tensor mutation is performed crossing control flow. Firstly, we add a newly defined variable x_o to the return value of the control flow operator of x′ and generates a new Update(x_o, x) after the control flow operator. Meanwhile, x is included in the return value of the block inside the control flow operator. Since the loop and branch have different block structures, it is necessary to cope with them respectively, referring to lines 26 - 31 in Algorithm 1. As for the branch operator, x needs to be added to the sibling block's return value. As for the loop operator, the operators inner the loop block are executed over and over again. As a result, we need to define a variable x_i and insert it into the block arguments. Meanwhile, a new Update(x_i, x) is generated at the beginning of the block to guarantee that the tensor version is the latest.

Finally, $\mathcal{P}$ is visited again, and it replaces all uses of v with v′ after the operator Update(v′, v). Update operators are removed at last because they are only for annotation and have no computation semantics.

4.1.3 Example. Figure 4 (a) is the original imperative tensor program in Python and Figure 4 (b) is its corresponding graph-level representation. There is a mutation operator aten::copy_ inner the block of loop operator prim::Loop. The rewrite mutation process is shown in Figure 4 (c) and the code after block propagation is shown in Figure 4 (d). Generated Assign operators during pass-up step are located in lines 7-8 of Figure 4. Generated Access operators and Update operators during pass-down step are located in lines 9-11, and 12-15. To perform functionalization beyond the control flow, two extra Update operators are inserted at lines 3 and 17.

After removing `Update` operators and dead code elimination (DCE) at last, we get the graph-level IR in TensorSSA form in Figure 4 (e).

4.2 High-Performance Code Generation

The functionalized program can provide more potential optimization opportunities. We propose two representative optimization techniques for kernel fusion: the *vertical* optimization and the *horizontal* parallelization. This is attributed to its inherent ability to unleash optimization potential.

4.2.1 Vertical Optimization. The `Assign` operators and the `Access` operators can be easily represented using a data flow graph, which can be directly converted to equivalent tensor-level expression using a domain-specific language (DSL) by deep learning compiler backend [4, 17, 29] to generate high-performance code in multiple hardware. As a result, more operators can be fused to one kernel after *TensorSSA* transformation.

4.2.2 Horizontal Parallelization. TensorSSA can indeed facilitate parallelism in the horizontal direction. Firstly, TensorSSA enables the efficient fusion of code inner loop blocks. The program can further enable more aggressive loop parallelism at the graph level once all the operators inner the block are fused into one kernel. This allows horizontal program optimizations, where multiple iterations of a loop can be executed in parallel.

5 Evaluation

5.1 Experimental Setup

Implementation. We implement *TensorSSA* based on TorchScript IR and its compiler pass infrastructure in PyTorch 2.0. The optimization passes unleashed by *TensorSSA*, vertical optimization and horizontal parallelization, are implemented based on PyTorch NNC.
Platform. We conduct experiments on both consumer and data center platforms. The consumer platform has an NVIDIA RTX 1660Ti-6GB GPU and a 16-core Intel Core i7-11700 2.50GHz CPU. The data center platform has an NVIDIA RTX 3090-24GB GPU and a 128-core Intel(R) Xeon(R) Platinum 8369B 2.90GHz CPU.
Workloads. We benchmark four popular computer vision models, YOLOv3 [18], SSD [13], YOLACT [3], and FCOS [22], three natural language processing (NLP) models, NASRNN [30], LSTM [8], and seq2seq [21] and one commonly used basic module, attention [23].
Baselines. Note that we use TensorRT to execute the neural network parts of deep learning workloads, and *TensorSSA* and the following baselines for the imperative tensor programs part. We compare our algorithm with three mainstream deep-learning compiler pipelines within PyTorch including TorchDynamo [25] + TorchInductor [25], TorchScript [7] + nvfuser [20], and TorchScript [7] + NNC [29]. TorchDynamo + TorchInductor is a tracing-based compilation pipeline that emerged after PyTorch 2.0. Specifically, TorchInductor implements data-flow functionalization using functorch [9]. nvfuser and NNC are the default backends of TorchScript before and after PyTorch V2.1.0, respectively. Note that they can also represent the other mainstream deep learning compilers [2, 4, 17].

5.2 Overall Performance

Figure 5 depicts the speedup of the compilation pipelines to the PyTorch eager mode. The evaluation shows that our method achieves a consistent speedup compared with all benchmarks. We achieve up to 1.79X speedup to the best baseline. There are two observations of the evaluation of overall performance. Firstly, the speedup for NLP models is more significant than for CV models. In CV models, there are numerous compute-intensive operators, such as convolutions, which generally have widely used standard libraries. For NLP models, memory-intensive operators constitute a significant proportion, making performance improvements crucial in various compilation pipelines. Secondly, compared with the baselines, we achieve a consistent speedup. View and mutation operations dominate the performance of domain-specific memory-intensive operators in CV models. In terms of NLP models and Attention, tensor views and mutations are commonly distributed within the loop used for iterating over sequence length. Our approach demonstrates the capability to transform and fuse tensor view and mutation operators, resulting in additional acceleration compared to other baselines.

5.3 Kernel Fusion

Figure 6 depicts the counts of kernel launches during the execution. We can have two observations. Firstly, our method can significantly reduce the counts of kernel launches for most cases. This is because we eliminate the side-effect of tensor mutation in imperative tensor programs by *TensorSSA*. Tensor view and tensor mutation operators can be transformed into equivalent tensor `Access` and tensor `Assign` operators without any implicit mutation, which can perform fusion safely. Secondly, for some certain cases, NASRNN and seq2seq, we launch less kernels than TorchScript + NNC and we still achieve performance speedup over all the baselines although the counts of kernel launches are not less than TorchDynamo + TorchInductor. The reason is from two aspects. TorchDynamo relies on the Python interpreter to execute control flow, which incurs non-negligible performance overhead. Besides, after functionalization, the layout of the tensor data can become a performance-friendly dense type, which further contributes to efficient code generation.

5.4 Scalability

To demonstrate the scalability of our method, we evaluate the workloads in different batch sizes, which are depicted in Figure 7. Compared to other baselines, we achieve improvements across various scales. For the SSD, FCOS, and seq2seq models, as the batch size increases, the proportion of memory-intensive operations becomes greater in the overall workload. Therefore, our method provides a more significant acceleration with the increase of batch size. For YOLOV3, YOLACT, and Attention, the models become more computation intensive and the speedup decreases when the batch size increases. Figure 8 depicts the latency metric of NLP and Attention tasks in different sequence lengths scenarios. Our method exhibits linear time growth and consistently outperforms other baselines.

6 Related Work

Program Functionalization. The Single Static Assignment (SSA) [6] form widely used in modern compilers can be regarded as a functionalization of the source program [1]. The original SSA form only eliminates the mutation of scalar variables. To handle other kinds of mutation, several advanced SSA forms have been proposed including memory SSA [5]. The design of *TensorSSA* is inspired by memory SSA. However, memory SSA mainly handle scalar programs, while *TensorSSA* represents implicit tensor mutation in tensor programs.
Graph-Level Optimization. Rammer [14] performs inter-operator schedules to explore more potential parallelization. Cocktailer [26] enables the generation of high-performance GPU code for dynamic

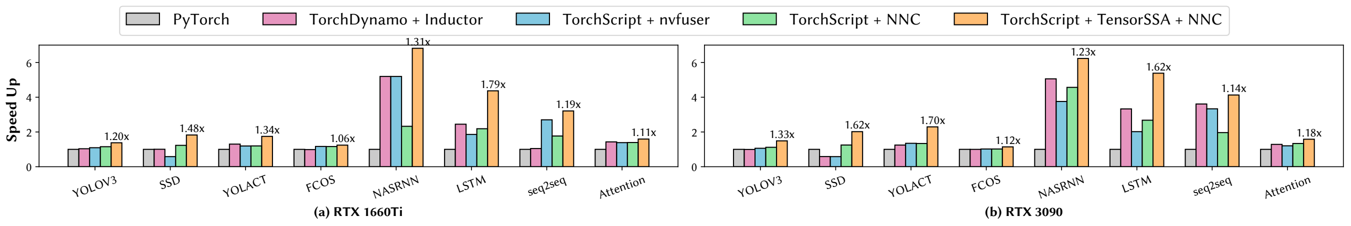


Figure 5. Comparison of end-to-end inference performance on imperative tensor programs.

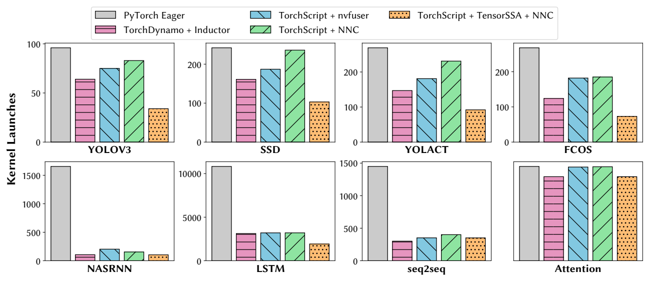


Figure 6. Counts of kernel launches.

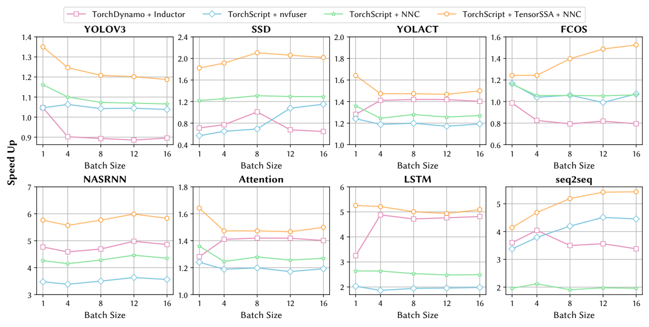


Figure 7. Speedup of different batch sizes.

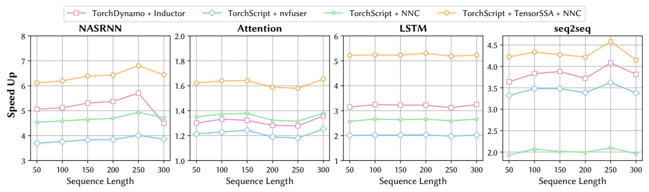


Figure 8. Speedup of different sequence lengths.

neural networks with control flow. TASO [10] and PET [24] substitute a pattern-matched segment in the computation graph to a efficient equivalent or approximation. In *TensorSSA*, we perform substitution globally in a series of operators, including loops and view operators. AStitch [28], DNNFusion [15] and Ansor [27] enable tensor-compute operator fusion and high-performance code generation to reduce latency and off-chip memory access.

7 Conclusion

In this paper, we perform program functionalization to optimize imperative tensor programs. We propose *TensorSSA* IR for functionalization. We convert the tensor view and mutation into a pure immutable function, which can be fused to GPU kernels without worrying about side effects. The experiment shows the efficiency of our method in both consumer and data center GPU platforms.

ACKNOWLEDGMENT

We sincerely thank anonymous DAC reviewers for their valuable comments on this paper. The research is supported under the National Key R&D Program of China (2022ZD0160202), as well as cash and in-kind contributions from the industry partner(s).

References

[1] Andrew W. Appel. 1998. SSA is Functional Programming. *SIGPLAN Not.* (1998).
[2] The IREE Authors. 2019. IREE. https://openxla.github.io/iree/
[3] Daniel Bolya et al. 2019. YOLACT: Real-Time Instance Segmentation. In *ICCV*.
[4] Tianqi Chen et al. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *OSDI*.
[5] Fred Chow et al. 1996. Effective representation of aliases and indirect memory operations in SSA form. In *CC*.
[6] Ron Cytron et al. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* (1991).
[7] Zachary DeVito. 2022. TorchScript: Optimized execution of PyTorch programs. https://program-transformations.github.io/slides/pytorch_neurips.pdf
[8] Sepp Hochreiter et al. 1997. Long Short-Term Memory. *Neural Computation* (1997).
[9] Richard Zou Horace He. 2021. functorch: JAX-like composable function transforms for PyTorch. https://github.com/pytorch/functorch.
[10] Zhihao Jia et al. 2019. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *SOSP*.
[11] Lijuan Jiang et al. 2023. EasyView: Enabling and Scheduling Tensor Views in Deep Learning Compilers. In *ICPP*.
[12] Chris Lattner et al. 2021. MLIR: Scaling compiler infrastructure for domain specific computation. In *CGO*.
[13] Wei Liu et al. 2016. SSD: Single Shot MultiBox Detector. In *ECCV*.
[14] Lingxiao Ma et al. 2020. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In *OSDI*.
[15] Wei Niu et al. 2021. DNNFusion: Accelerating Deep Neural Networks Execution with Advanced Operator Fusion. In *PLDI*.
[16] Adam Paszke et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*.
[17] Ragan-Kelley et al. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *PLDI*.
[18] Joseph Redmon et al. 2018. YOLOv3: An Incremental Improvement. *CoRR* (2018).
[19] James Reed et al. 2022. torch.fx: Practical Program Capture and Transformation for Deep Learning in Python. In *MLSys*.
[20] Christian Sarofeen et al. 2022. Introducing nvFuser, a deep learning compiler for PyTorch. https://pytorch.org/blog/introducing-nvfuser-a-deep-learning-compiler-for-pytorch/
[21] Ilya Sutskever et al. 2014. Sequence to Sequence Learning with Neural Networks. In *NeurIPS*.
[22] Zhi Tian et al. 2019. FCOS: Fully Convolutional One-Stage Object Detection. In *ICCV*.
[23] Ashish Vaswani et al. 2017. Attention is all you need. *NeurIPS* (2017).
[24] Haojie Wang et al. 2021. PET: Optimizing Tensor Programs with Partially Equivalent Transformations and Automated Corrections. In *OSDI*.
[25] Peng Wu. 2023. PyTorch 2.0: The Journey to Bringing Compiler Technologies to the Core of PyTorch (Keynote). In *CGO*.
[26] Chen Zhang et al. 2023. Cocktailer: Analyzing and Optimizing Dynamic Control Flow in Deep Learning. In *OSDI*.
[27] Lianmin Zheng et al. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *OSDI*.
[28] Zhen Zheng et al. 2022. AStitch: Enabling a New Multi-Dimensional Optimization Space for Memory-Intensive ML Training and Inference on Modern SIMT Architectures. In *ASPLOS*.
[29] Mikhail Zolotukhin. 2021. NNC walkthrough: how PyTorch ops get fused. https://dev-discuss.pytorch.org/t/nnc-walkthrough-how-pytorch-ops-get-fused/125
[30] Barret Zoph et al. 2018. Learning transferable architectures for scalable image recognition. In *CVPR*.