

Uncovering Nested Data Parallelism and Data Reuse in DNN Computation with FractalTensor

Siran Liu^{1,2*†} Chengxiang Qi^{1,3 *†}
 Ying Cao¹ Chao Yang² Weifang Hu^{1,4*} Xuanhua Shi⁴
 Fan Yang¹ Mao Yang¹

¹Microsoft Research, ²Peking University, ³University of Chinese Academy of Sciences

⁴Huazhong University of Science and Technology

Abstract

To speed up computation, deep neural networks (DNNs) usually rely on highly optimized tensor operators. Despite the effectiveness, tensor operators are often defined empirically with ad hoc semantics. This hinders the analysis and optimization across operator boundaries. FractalTensor is a programming framework that addresses this challenge. At the core, FractalTensor is a nested list-based abstract data type (ADT), where each element is a tensor with static shape or another FractalTensor (i.e., nested). DNNs are then defined by high-order array compute operators like map/reduce/scan and array access operators like window/stride on FractalTensor. This new way of DNN definition explicitly exposes nested data parallelism and fine-grained data access patterns, opening new opportunities for whole program analysis and optimization. To exploit these opportunities, from the FractalTensor-based code the compiler extracts a nested multi-dimensional dataflow graph called *Extended Task Dependence Graph* (ETDG), which provides a holistic view of data dependency across different granularity. The ETDG is then transformed into an efficient implementation through graph coarsening, data reordering, and access materialization. Evaluation on six representative DNNs like RNN and FlashAttention on NVIDIA A100 shows that FractalTensor achieves speedup by up to 5.45x and 2.14x on average through a unified solution for diverse optimizations.

CCS Concepts: • Software and its engineering → Source code generation.

*Work done as an intern at Microsoft Research.

†Both authors contributed equally to this research.

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. SOSP '24, November 4–6, 2024, Austin, TX, USA

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-1251-7/24/11

<https://doi.org/10.1145/3694715.3695961>

Keywords: Deep learning compiler, Loop program analysis, Nested data parallelism

ACM Reference Format:

Siran Liu, Chengxiang Qi, Ying Cao, Chao Yang, Weifang Hu, Xuanhua Shi, Fan Yang, Mao Yang. 2024. Uncovering Nested Data Parallelism and Data Reuse in DNN Computation with FractalTensor. In *ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24)*, November 4–6, 2024, Austin, TX, USA. ACM, New York, NY, USA, 18 pages. <https://doi.org/10.1145/3694715.3695961>

1 Introduction

Deep neural networks (DNNs) usually rely on highly optimized tensor operators to improve performance [2, 24]. The operators corresponding to the compute-intensive portions in a DNN, such as matrix multiplication or convolution, are implemented as high-performance kernels executed by accelerators like GPU. The entire computation is then abstracted as a directed acyclic graph (DAG) of tensor operations.

While it enables much-needed performance improvements, the DAG abstraction creates unintended barriers to the rapid evolving DNNs. First, DAG is less *expressive* and problematic to support many DNN algorithms [7]. Second, due to the first point, users either use a more flexible, imperative programming interface like PyTorch [24] to implement new DNNs while sacrificing *efficiency*, or keep introducing new tensor operators with optimized performance but ad-hoc semantics based on developer's experiences [12]. The same optimizations often have to be applied manually and repeatedly in new operators, which requires significant effort and complicates end-to-end program analysis and optimization [12, 28].

We argue that the conflict between expressiveness and efficiency for DNN programming has its root in the DAG abstraction, which is insufficient for the fast-evolving DNN algorithms. Essentially, DNN computations heavily rely on *nested loops* that iterate over elements in tensors, i.e., high-dimensional arrays. They can exhibit complex parallelism with complicated data dependency across multiple loop levels. This is particularly true for DNNs that cannot be naturally expressed by a *single-level* DAG of tensor operators [5, 9, 12, 18, 31, 35, 38] (§2). Such complex parallelism and data dependency are buried and hidden in the single-level DAG, which introduces semantic boundaries and makes holistic program analysis and optimization difficult.

This paper presents FractalTensor, a framework that offers a high-level array programming interface as well as the underlying compiler infrastructure to facilitate diverse and efficient DNN algorithms. The core of FractalTensor is a nested list-based abstract data type (ADT), where each element is a static-shape tensor or another FractalTensor (hence the term nested). A DNN program is then defined by FractalTensors, along with the high-order array compute operators like map/reduce/scan and array access operators like window/stride operated at the use-specified nested level of the FractalTensor. The computation is organized and iterated over the (possibly nested) elements across different dimensions defined in the FractalTensor. The entire FractalTensor program has no ad-hoc or empirically defined operator boundaries.

From the FractalTensor-based code, the FractalTensor compiler extracts a nested multi-dimensional dataflow graph called the *Extended Task Dependence Graph* (ETDG), which provides a holistic view of parallelism and dependencies across different levels of control and data granularity. The ETDG represents both task parallelism and data parallelism at various nested levels, thereby preserving the intricate dependencies necessary for whole program analysis and optimizations. Specifically, the compiler performs three steps to transform an ETDG into an efficient implementation. First, to minimize the overhead associated with nested control, it attempts to flatten the nested graph hierarchy of the ETDG (termed graph coarsening) by merging the array operators within and across different nested levels when possible. Second, it reorders data accesses to enhance exploitable data parallelism and improve data locality. Finally, it defers the materialization of data access until the data are transferred between the hardware memory hierarchy, and compute-intensive operations are fused properly. This way, FractalTensor makes various optimizations accessible to a wide range of rapidly evolving DNN models via high-level program constructs and the algorithmic specification.

FractalTensor is capable of expressing diverse DNNs, including representative ones like RNN, flash attention, and their variants. Evaluations on the six DNNs on NVIDIA A100 show that FractalTensor achieves speedup by up to 5.45x and 2.14x on average through a unified solution for seemingly diverse optimization. The code is available at [1]. In summary, this paper makes the following contributions:

- The design of FractalTensor. A nested list-based ADT of statically-shaped tensors, a small set of functional second-order array combinators, access operators, and user-defined math function can effectively expressive a wide range of DNN algorithms (§4.1).

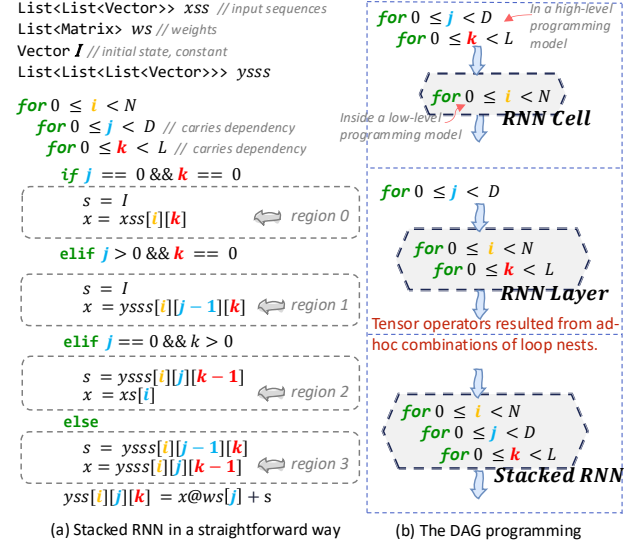


Figure 1. The limitations of existing programming models.

- *Extended Task Dependence Graph*, a nested multi-dimensional dataflow graph to abstract DNN computation, which elaborates the dependency relationship at all control levels, allowing for more efficient whole-program analysis (§4.4).
- Three steps to transform an ETDG into an efficient execution: ETDG coarsening (§5.1) minimizes nested control overheads; reordering analysis (§5.2) enhances exploitable data parallelism and access locality; the materialization of data and computation to match hardware resource constraints (§5.3).
- The demonstration of the proposed techniques on six innovative DNNs that fundamentally challenges the DAG abstraction (§6).

2 Background and Motivation

Limitations of existing approaches. Figure 1 shows two typical implementations for stacked Recurrent Neural Network (RNN) [31]. These two approaches demonstrate the dilemma between expressiveness and efficiency for the development of emerging DNNs. Figure 1(a) shows an intuitive way, i.e., using the imperative interface, to implement the stacked RNN. As shown, the stacked RNN involves complicated cyclic data dependencies carried by nested loops. This makes program analysis and optimization difficult, especially for the global analysis across different regions marked in Figure 1(a). Hence the performance of this approach is often suboptimal.

To improve performance while still preserving a certain degree of expressiveness, three levels of tensor operators within the DAG-based approach are introduced. This includes operators implementing RNN cell, RNN layer, and

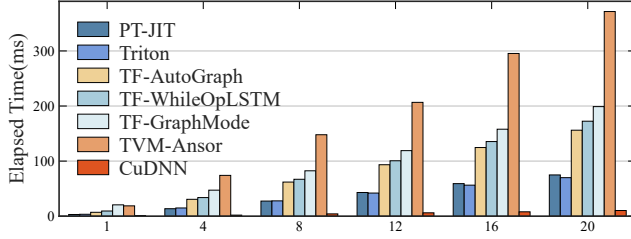


Figure 2. The performance varies based on the stack depth N (N RNN layers are stacked). PT/TF: PyTorch/TensorFlow.

stacked RNN network, respectively, as shown in Figure 1(b). Each tensor operator is an isolated loop nest, introduced to achieve efficiency. Developing these three operators requires significant effort, and similar optimizations must be applied manually and repeatedly across them. Additionally, the operators are defined in an ad-hoc manner, with their semantics understood only by experienced developers. Moreover, implementing these operators in a low-level programming model (e.g., CUDA) creates ad-hoc boundaries across operators.

Although certain domain-specific languages (DSLs) [6, 10, 33, 34], notably TVM [10, 27] or Triton [33], can alleviate the development effort of tensor operators, the scope of DSLs is often constrained within one or several adjacent operators [29, 41, 45] due to the limited expressiveness of the DSLs. The incurred operator boundaries make holistic program analysis and optimization difficult, if not impossible [10, 21, 29, 32–34, 39–41, 45, 46].

Figure 2 illustrates the performance of various approaches for the stacked RNN, which offer different trade-off between expressiveness and efficiency. The figure shows that the execution time increases with the depth of the stacked layers (i.e., the depth of nested loops). Note that the time of the most performant stacked RNN implementation, i.e., the handcrafted cuDNN version [11], only increases slightly with the increase of depth. But the remaining DAG-based approaches, including TVM, PyTorch, or TensorFlow, all slow down more significantly. This is because the compiler cannot understand the sophisticated dependency and obscure parallelism across nested loops buried in the single-level DAG used by these approaches. Without a proper abstraction, advanced dependency analysis on loop transformation is inapplicable [36]. While the handcrafted cuDNN version can manually implement advanced scheduling (e.g., [5]) for the entire stacked RNN as a single operator, thus the superior performance.

Challenges. From the above example we conclude the following fundamental challenges. The first challenge is to identify and exploit the *obscure data parallelism*, especially in the presence of complex, fine-grained data dependencies across operator boundaries (for DAG-based approaches) or across

nested loops (for imperative approaches). Although well-studied compiler techniques like Polyhedral offer a powerful mathematical framework to represent complex scheduling for nested loop programs based on a data dependence model, they can impose a combinatorial complexity to find a reasonable scheduling within the space constructed by general imperative nested loops [4, 26].

The second challenge is the difficulty to identify and exploit the *subtle data reuse opportunities* across operators or nested loops. Most existing programming interfaces cannot provide a high-level data access pattern for DNNs. Without this information, the DNN codes are often directly materialized into data movement across memory hierarchies, leading to a memory-bound behavior. Optimizing the memory-intensive data movement locally is often insufficient, as the data reuse opportunities can exist in a larger scope, e.g., across operator or loops boundaries, and are intertwined with computation. To achieve better performance, a compiler needs to precisely identify both memory-intensive and computation-intensive operations and jointly fuse these two types of operations.

Moreover, even the global optimization opportunities are identified, the optimizations often require addressing the complex control structures inherent in DNN computations, such as imperfectly nested loops and conditional branching, which pose significant scheduling challenges.

Opportunities. Our investigation to a comprehensive set of conventional and innovative DNNs has led to two key observations. First, the diverse DNN computation patterns can be expressed by a combination of second-order array compute operators like *map*, *reduce*, *scan*, *fold*, and their nesting. These operators explicitly declare the high-level computation patterns iterated over different dimensions in tensors, which significantly reduces the program analysis space. And parallelism opportunities can therefore be exploited by the combination of loop skewing, interchange, fusion, distribution, and blocking, in a way more affordable than conventional Polyhedral approaches due to the explicit data parallel semantics. Moreover, the use of these array operators avoids unnecessary conditional branching commonly used in imperative DNN implementations.

Second, the high-level data access patterns during DNN computation are highly stylized and can be expressed by a few first-order array access operators, including linear, window, stride. The distinct data access patterns expose data reuse opportunities across loops, thus can be exploited to improving memory performance.

The two observations motivate the FractalTensor framework. The system is designed based on two key decisions. First, FractalTensor requires users to program nested fine-grained loops attached to a list-base ADT through high-level array compute operators with explicit parallelism semantics.

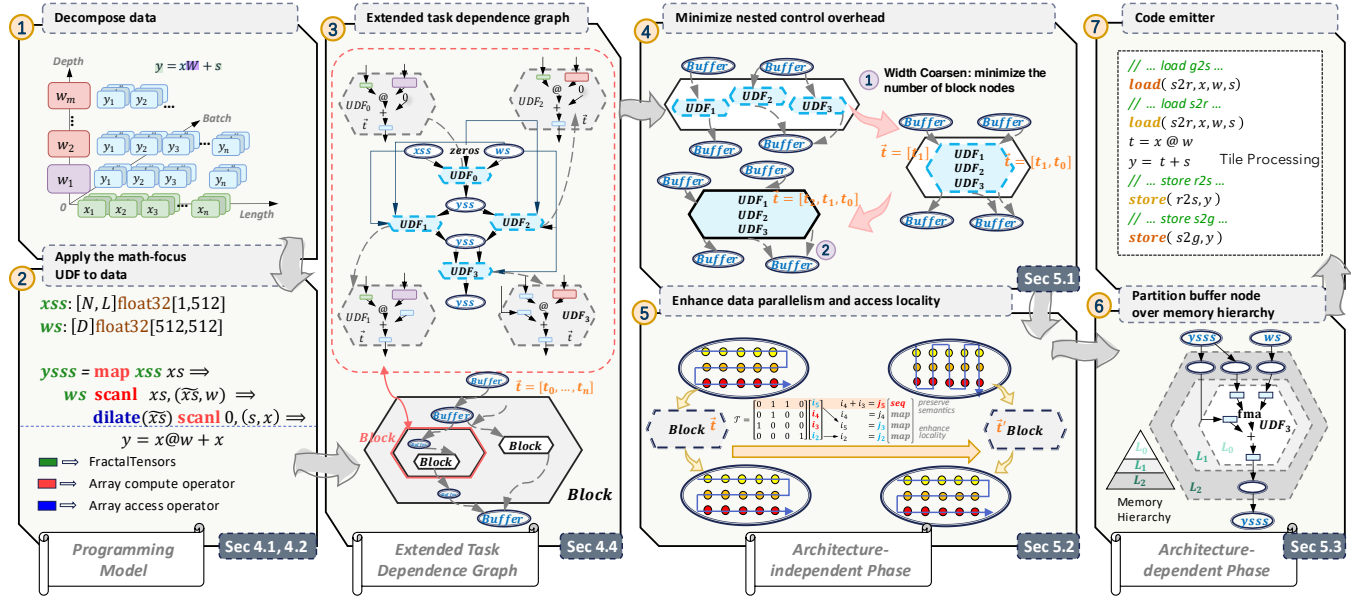


Figure 3. FractalTensor Overview. (I) Programming model (①②③), (II) Dependency-driven global analysis (④⑤⑥), (III) Code emitter (⑦).

Second, the FractalTensor, once constructed as a list-based data structure, can only be accessed through a set of array access operators that defer data movement and computation materialization until necessary.

3 Design Overview

Figure 3 overviews the design of the FractalTensor framework, which is primarily composed of three components:

(I) **Programming model.** DNN computations are organized around a list-based ADT called FractalTensor (§4.1). An intermediate representation (IR) called the *Extended Task Dependence Graph* (ETDG) is then parsed from user code in the form of a nested multi-dimensional dataflow graph (§4.4). This graph encodes fine-grained iteration(element)-level data dependencies and accurate data movement semantics using an edge annotation called access map. (II) **Dependence-driven global analysis.** To satisfy data dependency constraints, the scheduling transforms the user’s original algorithmic description into an efficient implementation through two steps. First, ETDG coarsening constructs coarse-grained data parallel tasks (§5.1) by fusing ETDG block nodes whenever possible and minimizing their depth. Second, access re-ordering identifies an execution order that exposes enhanced data parallelism and reuses in a more easily exploitable way (§5.2). (III) **Code emitter.** This component maps the ETDG onto an execution platform in both space and time (§5.3) and emits macro-kernels during ETDG traversal. We elaborate details of the components in the following sections.

4 Programming FractalTensor

4.1 FractalTensor

The shape of a generic DNN tensor is often unknown at the compile time. For example, the length of a sentence can vary. To accommodate such dynamism, the shape of a tensor is often defined using variables, i.e., dynamic shape. FractalTensor is an abstract data type that captures such dynamism with static-shape tensors. Specifically, a FractalTensor is a linearly ordered list. Each element in the list is either a tensor with static shape or another FractalTensor. A FractalTensor can be nested to any depth, but once declared, its depth is fixed and known at compile time. FractalTensor separates the dimensions of a generic tensor in DNN programming into two components:

- The innermost **static dimensions**, consisting of static-shape tensors, serve as basic elements of a FractalTensor. Math operations are defined only on static-shape tensors.
- The enclosing **programmable dimensions** always encompass the static dimensions. These dimensions are ordered, with each assigned a specific depth in a FractalTensor, where the outermost dimension has the highest depth.

4.2 High-level array operators

The expressiveness of FractalTensor comes from its programmable dimensions, which are explicitly controlled by a set of high-level functional operators.

A FractalTensor can only be initialized or derived by transforming existing FractalTensors using a set of high-level functional array operators, which we elaborate next.

Table 1. A summary of array compute operators

Name	Interface	Definition	Function Type
map	$map(f, xs)$	$[f(x_0), \dots, f(x_m)]$	$(\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]$
reduce	$reduce(\oplus, xs)$	$x_0 \oplus x_1 \dots \oplus x_m$	$(\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow [\alpha] \rightarrow \alpha$
	$reduce(\oplus, s_0, xs)$	$s_0 \oplus x_0 \oplus x_1 \dots \oplus x_m$	$(\beta \rightarrow \alpha \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow \beta$
foldl	$foldl(\oplus, xs)$	$x_0 \oplus x_1 \dots \oplus x_m$	$(\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow [\alpha] \rightarrow \alpha$
	$foldl(\oplus, s_0, xs)$	$s_0 \oplus x_0 \oplus x_1 \dots \oplus x_m$	$(\beta \rightarrow \alpha \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow \beta$
scanl	$scanl(\oplus, xs)$	$[x_0, x_0 \oplus x_1, \dots, x_0 \oplus x_1 \dots \oplus x_m]$	$(\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow [\alpha] \rightarrow [\alpha]$
	$scanl(\oplus, s_0, xs)$	$[s_0 \oplus x_0, s_0 \oplus x_0 \oplus x_1, \dots, s_0 \oplus x_0 \oplus x_1 \dots \oplus x_m]$	$(\beta \rightarrow \alpha \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow [\beta]$

1. $xs = (x_0, x_1, \dots, x_m)$ is a FractalTensor.
2. \oplus is associative in reduce, fold, scan; \oplus is left/right associative in foldl/r or scanl/r.

Array compute operators utilize a set of second-order array combinators (SOACs)[15, 16], which inherently operate in parallel and are independent of specific implementations. These operators accept user-defined functions composed of side-effect-free primitive tensor operations, which can themselves be array compute operators. The use of array compute operators promotes explicit nested data parallelism.

Based on the degree of parallelism, FractalTensor supports two kinds of array compute operators. ① The *fully parallel apply-to-each* operator includes `map`. Such operators require no communication between user-defined functions, thus incurring no inter-iteration dependencies. ② The *partially parallel aggregate* operator comprises `reduce`, `foldl/r`, and `scanl/r`. Each of these operators uses a user-defined binary associative operator \oplus to process FractalTensor elements. The linear order of FractalTensor elements, along with the associativity of \oplus , dictates the desired execution order. Due to inter-iteration data dependencies, successive iterations of this operator can be partially overlapped, thus exposing parallelism opportunities.

Table 1 summarizes the key interfaces of array compute operators, including `reduce`, `foldl`, and `scanl` as examples to illustrate aggregate patterns, omitting other patterns for brevity. We use Greek letters such as α and β to denote type variables, and $[\alpha]^d$ to represent a FractalTensor with α -typed elements and a fixed depth d . The depth of a FractalTensor is omitted when it does not cause ambiguity.

Array access operators are a set of first-order array combinators (FOACs) that take a FractalTensor of type $[\alpha]^{d_1}$ and return a new FractalTensor of type $[\beta]^{d_2}$, where α and β can be of the same or different types. These access operators are pure functions that do not perform computations but prepare data for array compute operators. The compiler leverages access operators to derive a formal representation internally, known as access map (§4.4), and defers the materialization of data access until necessary. Access operators are extensible, and FractalTensor has identified four types of access patterns that are particularly relevant to deep learning applications.

① The *contiguously linear* access operator refers to the contiguous access of a FractalTensor. It also supports a constant

shift of the starting position and a forward or reverse access order. ② The *constantly strided* access operator refers to a sequence of non-contiguous accesses defined by a starting position, a distance between adjacent accesses (the stride), and ending positions. Linear and constantly strided patterns are commonly found in sequence processing DNNs[9, 18]. ③ The *window* access operator refers to a sequence of accesses that are in spatially local regions and contain a certain degree of overlapping in data accessed by neighboring regions. This pattern is commonly found in convolution and stencil operations. ④ The *indirect* access operator is a sequence of accesses to a FractalTensor where the elements accessed are indexed by another array. In this pattern, a FractalTensor is usually accessed randomly. Indirect patterns commonly occur in irregular traversals or gather/scatter operations.

4.3 Stacked RNN in FractalTensor

Array compute and access operators can be combined and nested in various ways, allowing the expression of structured compound patterns that represent a wide range of DNN algorithms.

Listing 1. Stacked RNN in FractalTensor.

```

1 xss:[N,L]float32[1,512] = ...//input sequences
2 ws:[D]float32[512,512] = ...//input weights
3 ysss:[N,D,L]float32[1,512] //output
4 // map over the depth-2 of xss
5 ysss = xss.map xs =>
6     // scanl the depth-1 of ws
7     yss = ws.scanl xs, (xs, w) =>
8     // scanl the depth-1 of xss
9     ys = xs.scanl 0, (s, x) =>
10    // access statically-shaped tensors
11    // [1,512]=[1,512]@[512,512]+[1,512]
12    y = x @ w + s // udf math function

```

Listing 1 illustrates the pseudo code that implements Stacked RNN (shown in Figure 1) using FractalTensor constructs. A more formal definition of the abstract syntax of FractalTensor is provided in Appendix A.

Lines 1-3 in Listing 1 define two input FractalTensors, `xss` and `ws`, and one output FractalTensor, `ysss`. The innermost

dimension of these FractalTensors, specified on the rightmost side in the type declaration, represents static-shape tensors. Specifically, the $[1, 512]$ in xss denotes a 512-dimension vector, i.e., a single token in a sentence. The $[512, 512]$ in ws represents a 512×512 weight matrix. And the $[1, 512]$ in $ysss$ denotes a resulting token computed by the Stack RNN. The innermost dimension represents the minimum data unit in the DNN algorithm.

The outer dimensions of the FractalTensors, specified on the left side in the type declaration, are programmable dimensions. In Listing 1, L stands for the length of a sentence, N denotes the batch size of the sentences, and D represents the stacking depth of the stacked RNN.

The combination of the outer programmable dimensions and the innermost static dimension allows FractalTensor to maintain critical shape information for tensors across multiple dimensions while also providing the necessary flexibility to express generic tensors in DNN programming.




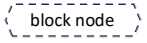
FractalTensor's programmable dimensions allow users to naturally program algorithm-level parallelizable loop nests. In lines 5-12 of Listing 1, the array compute operators `map` and `scanl` are applied in a nested way to the programmable dimensions of the defined FractalTensors to program stacked RNN. Specifically, the `map` in Line 5 iterates on the batch dimension N of xss (depth-2 of xss), processing each sentence in parallel. And the `scanl` in Line 7 works on dimension D of ws , iterating a stack of D weight matrices. And the `scanl` in Line 9 scans the dimension L of xss , computing each static-shape token x in xss with a weight matrix w in ws and producing the resulting tokens y of the shape $[1, 512]$ (line 12). The computation on the innermost dimension is specified by a user-defined math function. As shown in line 12, the shape of the final resulting FractalTensor, $ysss$, can be inferred through shape inference: the programmable dimensions N, D, L of $ysss$, as well as the static innermost dimension $[1, 512]$, are inferred from the corresponding dimensions of the input FractalTensors (line 1-2) through operators `map` and `scanl` (line 5-12). In this example, the compute operators use the default contiguously linear array access operator, which does not need to be specified explicitly.

By utilizing the high-level array compute and access operators, a compiler can understand the high-level data parallelism and data movement patterns, independent of specific implementations. This approach reduces the necessity for data dependence testing to detect complex parallelism [26].

4.4 Extended task dependence graph

From the FractalTensor-based code, the compiler extracts a nested multi-dimensional dataflow graph called Extended Task Dependence Graph (ETDG), a unified intermediate representation that preserves a holistic view of parallelism and dependency across different control and data nested levels

Table 2. Elements of ETDG.

Components	Description
	Operation node: represents user-defined. math tensor operations.
	Buffer node: an addressable compound value that has a single assignment property.
	Access map: An annotation attached to an dataflow edge that flows into or out of a buffer node.
	Block node: a control node that represents computations mixed with data parallelism and task parallelism.

on the code. To facilitate code analysis and the later low-level code generation, an ETDG concisely encodes complex control structures and precisely represents the iteration-level data dependencies with an acyclic graph. For a clear exposition, ETDG borrows the concepts from the reduced dependence graph used in classical compilers [4, 13] with the Static Control Program (SCoP) modeling employed in polyhedral compilers [19, 36].

The execution of an extended task dependence graph usually starts from one or more *buffer nodes*, each representing an instantiation of a certain FractalTensor. A buffer node connects to one or multiple *block nodes*, each of which consumes one or more buffer nodes and produces a new buffer node, i.e., a new instantiation of FractalTensor. Eventually the ETDG flows to a final output buffer node, which keeps the final result.

The block node represents the computation control logic for the incoming buffer nodes. As the computation can be nested, a block node can contain another node or even a sub ETDG. And the edge between a buffer node and a block node is annotated with an *access map*, which denotes the access pattern involved in the computation.

The four ETDG components are summarized in Table 2. In general, an **Extended Task Dependence Graph (ETDG)** is denoted as $\mathcal{G}_{ETDG} = (V, E, \mathcal{A})$, satisfying the conditions:

1. The node set V consists of operation nodes, buffer nodes, and block nodes, explained in Table 2.
2. Only block node can be nested. Each node $v_i \in V$ has at most one block node as its parent (the enclosing node). This ensures the correct relationship between nodes.
3. There exists a set of buffer nodes $v_r \in V$ with no parent. They serve as root nodes in the ETDG hierarchy.
4. Given a directed edge (v_i, v_j) where v_i or v_j is a buffer node, an access map $\mathcal{A}_{i,j}$ must be annotated.
5. No cycles are allowed. This prevents circular dependencies between nodes.

Next we explain the detailed meaning of the four elements of ETDG and the corresponding notations used in the later analysis and implementation.

Operation node. An operation node is a user-defined math-oriented tensor operations with no side effects.

Buffer node. A buffer node Λ_m is an addressable instance of a FractalTensor declared in user code. $\Lambda_m = (\vec{i}_m, \mathcal{D}_m, \Theta)$, where \vec{i}_m is an m -dimensional index vector $\vec{i}_m = [i_m, i_{m-1}, \dots, i_1]$, \mathcal{D}_m denotes the index's legitimate data domain defined by the index's range constraint Θ . For a buffer node representing an instance of *ysss* in Listing 1, $m = 5$, i.e., it is a 5-dimension tensor with the shape $[N, D, L, 1, 512]$. The shape also implies the range constraint for each dimension in the index, i.e., Θ . For example, the constraint for the first dimension is $0 \leq \Theta_1 < 512$. Thus the index vector \vec{i}_m satisfying the range constraints of *ysss* can address an element or a part of the 5-dimension tensor. All legitimate index vectors of the buffer node constitute its data domain, denoted as $\text{dom}(\Lambda_m) = \mathcal{D}_m$.

The high-level operators working on a buffer node must ensure its *single assignment* property. This means a buffer node can be read multiple times but written only once. By assigning a unique identifier to each buffer node, it can safely appear both at the source and the sink of a dataflow edge. This makes buffer node a valuable tool to control the merge of dataflow edges and avoid cycles in the graph when analyzing iteration-level data dependencies.

Block node. A block node is a d -dimension control node corresponding to a depth- d array compute operator nesting. In a block node, an array compute operator nesting is required to be *perfect*, where each array compute operator in the nesting is contained entirely within another array compute operator. Only array access operators can intervene in between the nesting, and no operation nodes are allowed in between.

A block node $\Gamma_d = (\vec{i}_d, \mathcal{P}_d, G_T = (V, E), \vec{p}_d)$, where \vec{i}_d is a d -dimensional iteration vector, \mathcal{P}_d denotes \vec{i}_d 's legitimate iteration domain, G_T denotes a sub-graph, and \vec{p}_d denotes a vector of array compute operator names. Iteration vector \vec{i}_d denotes the computing state of the block node. It represents an iteration state across a loop nesting with depth d . The iteration domain \mathcal{P}_d defines the set of all legitimate values \vec{i}_d can take. In the sub-graph G_T , the nodes V can be any valid ETDG elements, resulting in nested block nodes. Given any $v_i \in V$, the parent of v_i is Γ_d and v_i is consider a child of Γ_d .

A block node represents a computational task with data parallelism at multiple control levels. The hierarchy of block nodes forms a tree structure that defines a parent-child control relationship.

To execute a block node Γ_d , buffer nodes are required as both input and output. The execution of a block node can be viewed as the continuous processing of data flows from the

input buffer node(s) through the repeated execution of task G_T . This process is regulated by the d -dimensional iteration vector \vec{i} of the block node. We use *lexicographic order* [4, 13] to determine the sequence in which data is processed. Each dimension of \vec{i} is associated with an array compute operator, parsed from the user program, and is denoted by the operator's name: `map`, `reduce`, `foldl`, `scanl`. This association is represented by vector \vec{p}_d , which prescribes the inherent order in which G_T processes data in the corresponding buffer node(s), independent of any specific implementation.

Access map. An access map $\{\mathcal{A} : \mathcal{P}_d \rightarrow \mathcal{D}_m\}$ represents the positions where a d -dimensional block node Γ_d accesses an m -dimensional buffer node Λ_m . Given a compute state \vec{i}_d of Γ_d , the access position to Λ_m is represented by \vec{i}_m , calculated using a quasi-affine mapping [3, 17]: $\vec{i}_m = \mathcal{A}(\vec{i}_d) = M_{m \times d} \vec{i}_d + \vec{o}_m$. $M_{m \times d}$ is an access matrix representing the access patterns associated to Γ_d and Λ_m , and \vec{o}_m denotes an access offset.

$$\begin{bmatrix} i_3 \\ i_2 \\ i_1 \end{bmatrix} = M_{m \times d} \vec{i}_d + \vec{o}_m = \begin{bmatrix} 1, 0, 0 \\ 0, 1, 0 \\ 0, 0, 1 \end{bmatrix} \begin{bmatrix} t_3 \\ t_2 \\ t_1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}.$$

To illustrate how the access map functions, consider an operation $x = \text{ysss}[i][j][k-1]$ as depicted in Figure 1(a). In the FractalTensor code, this single read access is equivalent to the data access behavior of a depth-3 block node, defined by the compute operators `map`, `scanl`, and `scanl` nesting that reads the 3-dimensional buffer node *ysss* (omitting the two static-shape dimensions for brevity). And in ETDG, it is represented as a dataflow edge annotated with an access map shown below (e_{13} in Figure 4):

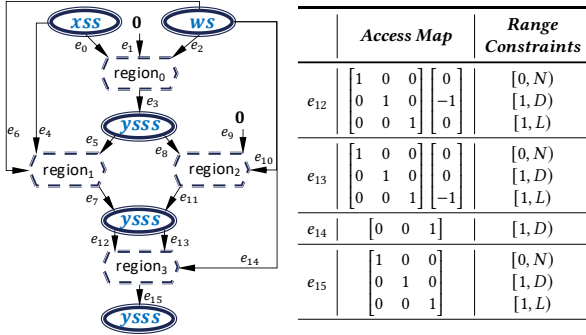
\vec{i} is the iteration vector of the block node, and \vec{i} is the index vector of the buffer node *ysss*. The offset vector is $\vec{o} = [0, 0, -1]^T$. And if an element $[m', d']$ in M is set to 1, this denotes the m' 'th dimension in the buffer node will be accessed by the d' 'th nested array compute operator. In the example, $M_{2,2} = 1$ (the lowest diagonal position of M), this means dimension 1 (counting from right to left) of *ysss*, i.e., L , will be accessed by the innermost array operator `scanl`.

To measure the complexity of nested control structures in an ETDG, we define its depth and dimensions.

The depth of an ETDG, denoted as $d(G)$, is defined as the number of block nodes on the longest path from the root node to a leaf operation node in the ETDG.

The dimension of an ETDG, denoted as $\|G\|$. As each block node can have multiple dimensions, $\|G\|$ is defined as the sum of the dimensions of all the block nodes along the longest path from the root node to a leaf operation node in the ETDG.

Figure 4 shows the ETDG of the FractalTensor code in Listing 1. The depth of the ETDG is 2 and the dimension is 5. For brevity, only the outer depth of the ETDG is shown. The block nodes `region0~3` write to distinct, non-overlapping

Figure 4. The depth-1 ETDG for the running example.

instances of the buffer node *ysss* (following the SSA property). Thus the parser can safely translate the separate conditional branches induced by aggregate pattern nesting into distinct block nodes, and the buffer node at the end of a dataflow edge functions as a control merge node.

The block node *region₃* carries data dependencies at two control depths, exemplifying the typical complexity. The corresponding access map, annotated for dataflow edges that flow into and out of this block node, as well as the range constraints for the corresponding buffer node, are shown on the right of Figure 4. Section 5 further explains the transformations applied to these access maps.

This ETDG is semantically equivalent to the imperative code depicted in Figure 1(a), but it encodes the critical information on the high-level compute and access patterns presented in the FractalTensor code, thus facilitating the global analysis and optimization.

5 System Implementation

FractalTensor is implemented using Python, Rust, and C++. The programming interface and parser are implemented based on PyTorch, with 3,000 lines of Python code. PyTorch is used to wrap the FractalTensor and its functional operators. The system extracts the ETDG from the PyTorch code and performs necessary optimizations (§5.1, §5.2). A dataflow analysis component with 2,000 lines of Rust code is implemented to analyze the optimized ETDG to convert it into C++ format. Finally, a tile library to compile the ETDG to CUDA codes is implemented using 15,000 lines of C++ code. The tile library is a collection of CUDA device functions to support the ETDG execution efficiently (§5.3).

Next, we elaborate three key steps that transform ETDG into efficient implementation.

5.1 ETDG coarsening

An ETDG represents nested data parallelism, which incurs nested control overheads. A FractalTensor compiler coarsens an ETDG to reduce both the depth and the dimension of the

ETDG, thus simplifying the nested control structures and improving the data parallelism. The coarsening consists of width-wise, which decreases ETDG's depth and the number of block nodes in it, and depth-wise, which reduces ETDG's dimension.

Width-wise ETDG coarsening merging block nodes whenever possible during operation node lowering, both horizontally and vertically.

Table 3. Composition rules for merging array compute operators.

o	map	reduce	scanr	scanl	scan
map	map	reduce	scanr	scanl	scan
reduce	reduce	reduce	scanr	scanl	scan
scanr	scanr	scanr	scanr	✗	scanr
scanl	scanl	scanl	scanl	scanl	✗
scan	scan	scan	scanr	scanl	scan

Operation node lowering: For ease of programming, FractalTensor allows user-defined math functions. These functions are represented as operation nodes in an ETDG. As the functions involve tensor algebra with no side effects, FractalTensor decomposes each operation node into finer-grained block nodes associated with compute operators working on individual tensor elements. The resulting ETDG removes the obscure semantics boundaries due to user defined functions, thus creating more opportunities for node merging.

Horizontally mergeable block nodes: Two block nodes, $\Gamma_{d_1}^1$ and $\Gamma_{d_2}^2$, at the same ETDG depth can be horizontally merged into a single block node if and only if: 1) there is no directed edge between them; 2) $d_1 = d_2$ and $\vec{p}_{d_1} \equiv \vec{p}_{d_2}$. In this case, the merged block node is represented as $\Gamma_{d_1} \cup \Gamma_{d_2} = (\vec{i}_{d_1}, \mathcal{P}_{d_1} \cup \mathcal{P}_{d_2}, G_T = (V_1 \cup V_2, E_1 \cup E_2), \vec{p}_{d_1})$.

Vertically mergeable block nodes: When two block nodes, denoted as $\Gamma_{d_1}^1$ and $\Gamma_{d_2}^2$, are at the same ETDG depth and are connected by a directed edge, they exhibit a producer-consumer relationship. To merge them vertically, it is essential to preserve all the iteration-level data dependencies in the original executions of $G_{T_1}^1$ followed by $G_{T_2}^2$.

To evaluate the feasibility of a vertical merge, we iteratively align the dimensions of \mathcal{P}_{d_1} and \mathcal{P}_{d_2} from the outermost to the innermost dimension. If the i -th dimension has the same length, we merge it into a single dimension if the associated array compute operators can be merged without conflicts (e.g., preserving the same data dependencies). Table 3 illustrates the composition table for merging two array compute operators into one.

When dimension length does not match, we create child block nodes, denoted as $\Gamma_{d_1-i}^1$ and $\Gamma_{d_2-i}^2$, to represent the unaligned iteration space. These child nodes are then made children of the block node for the aligned iteration space, ensuring they uphold the producer-consumer relationship in the same manner as $\Gamma_{d_1}^1$ and $\Gamma_{d_2}^2$.

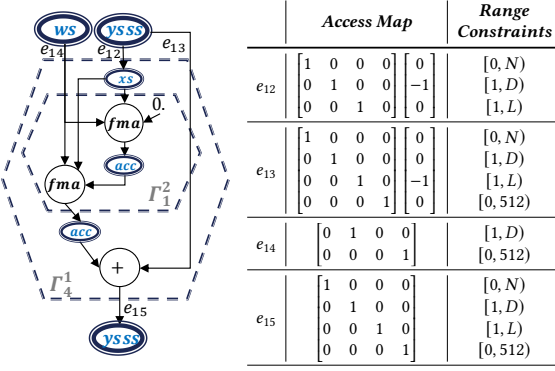
Figure 5. Width-wise coarsening for `region3` in Figure 4.

Figure 5 shows the ETDG for `region3` in Figure 4 after width-coarsening. The user-defined math function $y = x@w + s$ is decomposed and merged at the finest granularity based on shape, data dependence, and access operator analysis. The original ETDG is transformed into a two-depth graph, with the depth-2 graph executed on a four-dimensional iteration vector and the depth-1 graph executed on a one-dimensional iteration vector. Since the computation is implementation-independent at this stage, our analysis is constrained only by data dependence to preserve program semantics. As a result, it is highly flexible and can aggressively identify opportunities for fusing computations.

Depth-wise ETDG coarsening is applied to further minimize the dimensions of an ETDG after width-wise coarsening. This involves iteratively merging two dimensions of a d -dimensional block node. A dimension can interact with a buffer node in one of three ways: through an access relation, where the dimension accesses the buffer node; through an invariant relation, where the data in the buffer node remains unchanged when iterate over this dimension; or through a non-access relation, where the buffer node is not needed at this particular dimension. To be merged, two dimensions must interact with buffer nodes through access and/or invariant relations, and have the same array compute and access operator for accessing buffer nodes. Merging two dimensions expands the iteration space by taking the union of the original two iteration spaces. The *access map* is also updated to reflect the merged dimensions. When an invariant relation is merged with an access relation, the access function changes from contiguously linear to constantly strided, while other situations remain unchanged.

Depth-wise coarsening is similar to “axis fusion” proposed in Roller [46], but the former is hardware agnostic.

Access map fusion. The single assignment property requires copying a buffer node when the buffer node is mutated multiple times (e.g., BigBird in Listing 4 in Appendix B).

Table 4. Dependence approximation for array compute operators.

	dependence distance vector
map_i	$\vec{0}$ (fully parallelizable)
reduce_i	$\vec{d}, d_i = 1, \forall j \neq i, j \in [D, 0)$
foldl_i	$\vec{d}, d_i = 1, \forall j \neq i, j \in [D, 0)$
scanl_i	$\vec{d}, d_i = 1, \forall j \neq i, j \in [D, 0)$

1. op_i : the array compute operator op is nested at the i -th depth
2. D is the dimension of the block node, and $D > 1$.

This incurs additional memory overheads. Access map fusion reduces such overheads by merging directly-connected buffer nodes in an ETDG by composing the access matrices, offset vectors, and union range constraints.

5.2 Access reordering

After ETDG coarsening, FractalTensor reorders the computation to improve parallelism and data locality.

Reordering framework. Given a block node Λ_d accessing a buffer node Λ_m through the access map $\mathcal{A}(\vec{t}) = M_{m \times d} \vec{t}_d + \vec{o}_m$, the goal is to find a unimodular transformation matrix $\mathcal{T}_{d \times d}$ [19, 36] that: 1) preserves the computation semantics; 2) maximizes the number of fully parallelizable dimensions of Γ_d (e.g., a dimension associated with a `map` operator is fully parallelizable); 3) ensures that dimensions of a block node carrying data reuse are interchanged as inner dimensions to enhance data locality.

$\mathcal{T}_{d \times d}$ reorders the iterations of Γ_d into $\vec{j} = \mathcal{T} \vec{t}$ and accesses the buffer node by $\mathcal{A}(\vec{j}) = (M\mathcal{T}^{-1}) \vec{j} + \vec{o}$. Once \mathcal{T} is established, it is integrated into the access map with new range constraints to access buffer nodes derived using Fourier-Motzkin elimination [3].

To solve $\mathcal{T}_{d \times d}$, first note that in FractalTensor, only aggregate operators like `scanl` or `foldl` introduce dependencies. For example, iteration $i - 1$ of `scanl` writes to a destination buffer node where iteration i of `scanl` reads. This introduces a dependency distance of 1. A dependence distance vector [13, 36] can then approximate the iteration-level data dependency carried by aggregate operators.

Table 4 shows the dependence distance vector induced by an array compute operator when nested at depth- i (with the default continuously linear access operator). While array access operators do not introduce dependencies, they can modify the dependence distance. For instance, a strided linear access operator with a stride of 4, when combined with a `scan`, adjusts the dependence distance to 4.

Given a nest of compute operators, FractalTensor exhibits a critical property: any nest of compute operators in FractalTensor can be transformed into a *fully permutable* loop nest. This is because the nest satisfies the following conditions [36]: 1) The computation is regulated by the high-level

functional operators discussed in §4.2. 2) The buffer nodes involved in the computation follow the single assignment property. 3) The value in the dependence distance vector is constant, i.e., a dependence distance is always fixed. A fully permutable loop nest can be transformed to contain at most one sequential loop [36]. This implies that when multiple dimensions of a block node carry data dependencies, a single transformed dimension of the block node is sufficient to carry *all* the data dependencies, while the other dimensions are free to permute to enhance access locality.

Constructing the transformation matrix. With the fully permutable property, the transformation matrix \mathcal{T} can be decomposed into two parts: the first row, which satisfies all the data dependencies, and the remaining rows, which denote dimensions that can be reordered for data reuse.

Let the semantic-preserving schedule that occupies the first row of \mathcal{T} be denoted as $\pi(\vec{t}) = a_n \vec{t}_n + \dots + a_1 \vec{t}_1$, where a_i are non-negative integers to be determined. Given that FractalTensor's dependence distance vectors contain only *constant* dependence distances, the hyperplane method [19] is used to select the appropriate a_i . Subsequently, block nodes whose dimensions involve data reuse are interchanged into inner dimensions, with a minimized number of interchanges to avoid unnecessary data movement.

$$\mathcal{T} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} t_5 \\ t_4 \\ t_3 \\ t_2 \end{bmatrix} \begin{bmatrix} \text{map} \\ \text{scanl} \\ \text{scanl} \\ \text{map} \end{bmatrix} \begin{bmatrix} t_4 + t_3 = j_5 \text{ seq} \\ t_4 = j_4 \text{ map} \\ t_5 = j_3 \text{ map} \\ t_2 = j_2 \text{ map} \end{bmatrix}$$

Figure 6. The transformation matrix and its meaning.

To identify data reuse, let \vec{t}_1 and \vec{t}_2 represent two iterations that refer to the same buffer node location: when $M\vec{t}_1 + \vec{o} = M\vec{t}_2 + \vec{o}$, the two iterations are referring to the same place, thus data reuse occurs. The set of all solutions to the equation $M(\vec{t}_1 - \vec{t}_2) = \vec{o}$ is known as the *null space* of M , a concept in linear algebra. The null space can be represented by its basis vectors, which indicate that a specific dimension of the block node changes, but the data accessed by it remains unchanged. This presents a unique opportunity to exploit data locality [3].

Considering the block node Γ_4^1 , associated with $\vec{p}_4 = [\text{map}, \text{foldl}, \text{scanl}, \text{map}]$ in Figure 5, as an example. The aggregate operators `foldl` and `scanl` are applied to the 3rd and 2nd dimensions of Γ_4^1 , respectively. Two dependence distance vectors are then extracted: $\vec{d}_1 = [0, 1, 0, 0]$ and $\vec{d}_2 = [0, 0, 1, 0]$. Let M_i represent the access matrix of the access map annotated to dataflow edge e_i . The null space of M_{12} is spanned by the basis vector $[0, 0, 0, 1]$, indicating that the last dimension of block node Γ_4^1 carries data reuse during its execution. Similarly, the null space of M_{14} is spanned by the basis vectors $[1, 0, 0, 0]$ and $[0, 0, 0, 1]$, indicating that the first and last

dimensions carry data reuse during the execution of Γ_4^1 . The null spaces of M_{13} and M_{15} are empty, indicating that accessing the corresponding buffer node does not result in easily exploitable data reuse. To exploit locality, the first dimension of Γ_4^1 is interchanged with the second-to-last dimension, while the remaining dimensions remain unchanged. This is achieved by, for the third row of \mathcal{T} , setting $\mathcal{T}_{3,1} = 1$, and for the other rows, setting the diagonal element to 1. Figure 6 illustrates the transformation matrix constructed for Γ_4^1 . Table 5 presents the updated access map and range constraints for region_3 after applying \mathcal{T} .

Table 5. Transformed access map for region_3 in Figure 5.

	Access Map	Range Constraints
e_{12}	$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix}$	$\begin{bmatrix} [2, L + D - 1] \\ [\max(1, j_4 - L + 1), \min(j_4, D)) \\ [0, N) \end{bmatrix}$
e_{13}	$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -1 \\ 0 \end{bmatrix}$	$\begin{bmatrix} [2, L + D - 1] \\ [\max(1, j_4 - L + 1), \min(j_4, D)) \\ [0, N) \\ [0, 512) \end{bmatrix}$
e_{14}	$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} [2, L + D - 1] \\ [\max(1, j_4 - L + 1), \min(j_4, D)) \end{bmatrix}$
e_{15}	$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} [2, L + D - 1] \\ [\max(1, j_4 - L + 1), \min(j_4, D)) \\ [0, N) \\ [0, 512) \end{bmatrix}$

5.3 Access materialization

In the final hardware-dependent phase, we generate device code by implementing a tile computation library that elevates CUDA C's SIMT programming model to tile processing. The tile library handles data tile loading, computing, and storing with a base tile whose size is aligned with a TensorCore's instruction shape [44, 46]. The base tile serves as a basis to compose larger tiles aligned with various cache levels. As a result, the tile library enables a code emitter to traverse the ETDG and translate its elements into corresponding C++ implementations. Buffer nodes in the ETDG are decomposed into smaller tiles based on the base tile. And access maps, after being fused properly, are finally materialized efficiently into load and store tiles between memory hierarchies. During the final mapping step, memory requirements are calculated at the register level. The library selects predefined tile shapes that optimize cache utilization while maintaining a good SM occupancy. Subsequently, the data transfer from DRAM to SRAM and finally to registers is inserted. The process effectively manages each level of the GPU's memory hierarchy. In general, the practice is consistent with state-of-the-art tile-based DNN compilers, such as Welder [29] and PIT [44].

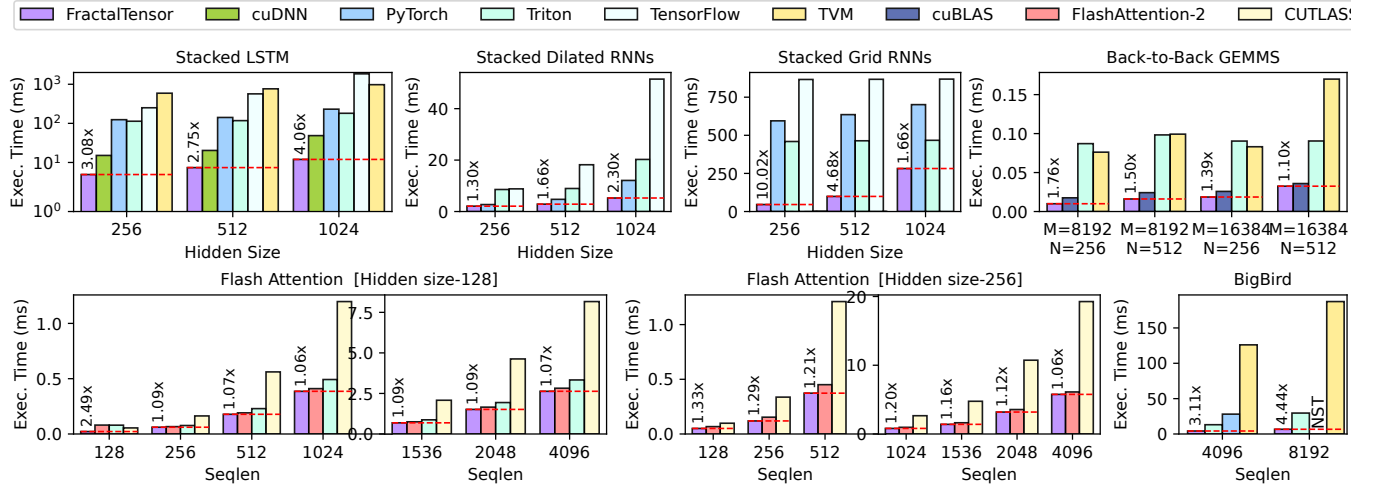


Figure 7. End-to-end execution time (ms) for each DNN workload at different shapes. NST means the compiler or framework does not support some workloads or shapes.

6 Evaluation

In this section, we evaluate FractalTensor by addressing two key questions. First, we assess the capability of the FractalTensor programming interface to express various innovative DNN algorithms. Second, we examine whether increased programmability facilitates the global analysis of these DNN algorithms, thereby making optimizations generally available for them.

6.1 Experimental setup

DNN workloads. To evaluate the effectiveness of the FractalTensor programming framework, we tested it on six representative DNN workloads. Table 6 shows the DNNs and the hyperparameters used in our evaluation. Our selection of algorithms includes one popular algorithm that has been widely adopted in vector-specific libraries. Additionally, we chose several of its algorithmic variants that modify the original algorithm slightly. This approach allows us to determine whether existing methods can exploit optimization opportunities that benefit all the algorithms.

Table 6. Benchmark specification

DNN algorithm	Shape
Stacked LSTM[5]	batch size: 256, depth: 32.
Stacked Dilated RNNs[9]	batch size: 256, dilation: [1 ~ 32].
Stacked Grid RNNs[18]	batch size: 256, depth: 32.
Back-to-Back GEMMs	K: 64, P: 64.
Flash Attention[12]	consistent with official implementation.
BigBird[38]	consistent with official implementation.

Baselines: We compare the selected workloads with baseline solutions in three categories: ① DL frameworks’ DAG

approach with compiler-supported analysis, including PyTorch 2.2.1 JIT and Tensorflow 2.2.0; ② DL compilers’ code-generation techniques, including TVM[10] commit ef46f4e (Mar 24, 2024) and Triton[33] 2.2.0; and ③ state-of-the-art manual optimization libraries, including cuDNN 8.9.7.29-12, cuBLAS[23], and CUTLASS commit f9ece1b (Mar 28, 2024). Additionally, we refer to the official implementation as “FlashAttention-2” for the Flash Attention benchmark.

Platform. We evaluate FractalTensor and all the baselines on servers equipped with NVIDIA GeForce A100 GPU.

6.2 Overall performance

Figure 7 displays the end-to-end execution time (in milliseconds) for the selected DNN workloads after warm-up. FractalTensor achieves speedups of up to 3.75x and 1.21x for the stacked LSTM and back-to-back GEMMs compared to the best baselines cuDNN and cuBLAS respectively. For stacked dilated RNNs and stacked grid RNNs, 2.35x and 5.44x speedups are achieved relative to the best baseline provided by PyTorch JIT and Triton, respectively. In the case of FlashAttention, a 1.07x speedup is observed compared to the official implementation. Despite the irregular and complex memory access patterns inherent in the BigBird workload, our approach still achieves a speedup of up to 1.36x against the best baseline, Triton. Overall, our proposed method improves the performance of various workloads, yielding an average speedup of 1.97x over the best baseline.

None of our evaluated workloads are naturally expressible as a DAG of tensor operators. Thus handcrafted implementations, if available, remain the optimal baselines. Innovative but less popular algorithms, such as dilated RNN, grid RNN, and BigBird, lack the necessary optimizations due to their

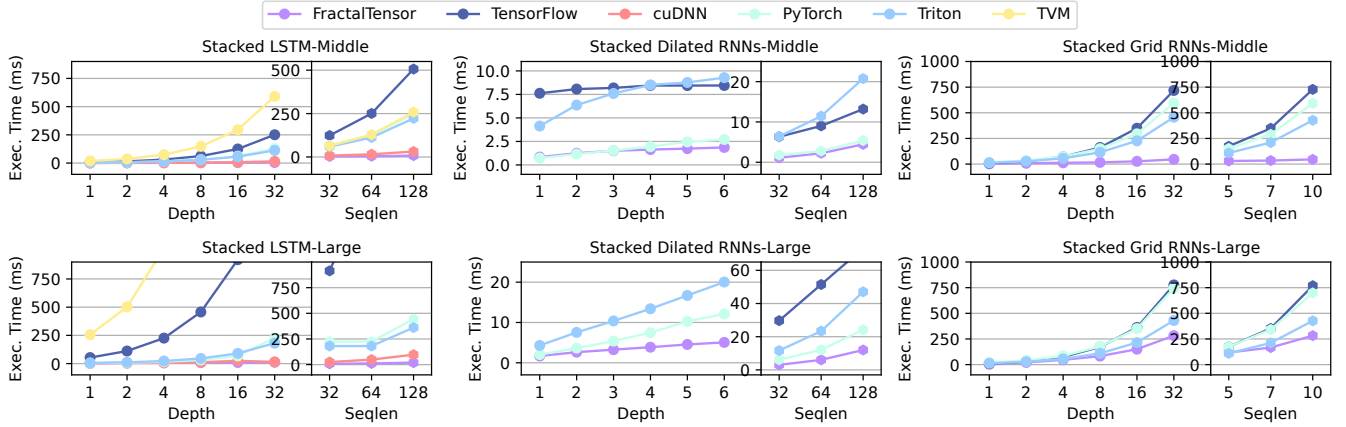


Figure 8. The performance of RNN with increased depth and sequence length (Middle model means the shape is batchsize=256, hidden=256. Large model means the shape is batchsize=256, hidden=1024).

limited adoption. Triton is the only existing solution capable of providing reasonable performance for all the selected workloads. This is because Triton elevates the abstraction level of CUDA’s SIMT programming model from the thread level to the block level, offering a compromise between DAG programming and low-level hardware specific programming. Users can implement the kernel using a simplified CUDA programming model, thus providing reasonable solutions for these innovative DNN algorithmic ideas. In contrast, our proposed techniques offer a unified solution to all these models, addressing diverse optimization considerations.

6.3 Parallelism exploitation

In this section, we conduct a case study examining the impact of array compute operators on the exploitation of parallelism.

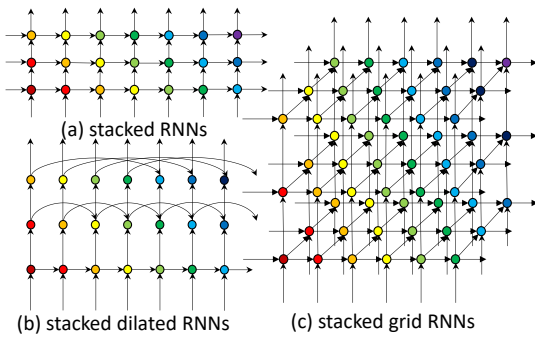


Figure 9. The dataflow structure of RNN variants.

Figure 8 presents a performance comparison of the FractalTensor framework for three RNN models and their scaling with key hyperparameters. First, we vary the hidden dimension and batch size to change the data parallelism within the local scope of the cell function, which acts as a pure dataflow

task. Second, we investigate the impact of sequence length on performance, increasing it to 32, 64, and 128. Third, we test the performance with increased depth. For stacked RNN and grid RNN, we increase the depth from 1 to 32 and report the performance at every 4 depths. For the dilated RNN, due to the significant increase in the dilation rate, the algorithmic design does not support extensive stacking; therefore, we increase the stacking layers from 1 to 6.

As the RNN layers become deeper, more parallelism opportunities occur. In Figure 9, cells of the same color can be executed in parallel. Our hypothesis is that if an automatic optimization technique can identify maximal exploitable data parallelism, the performance of the stacked RNN network should not scale linearly with depth. To test this hypothesis, we conducted four scaling experiments for each RNN algorithm. As shown in Figure 8, only the implementation of the stacked LSTM neural network using cuDNN satisfies this hypothesis. This implementation treats the entire stacked RNN network as a single operator and manually exploits wave-front execution, resulting in significantly better performance than other baselines.

Innovative algorithmic ideas such as dilated and grid RNNs have yet to be adopted by hardware vendors, as they fail to scale using existing methods. While Triton and PyTorch can exploit data parallelism within the local scope, computations within the local scope of the cell function cannot fully utilize high-end GPUs. In this situation, the depth and length dimensions that carry data parallelism across cells are crucial for enhancing performance. However, the DAG approach lacks an efficient way to handle the complex control structures underlying all RNN networks, with the exception of the cuDNN implementation. This implementation incorporates nested loops within a single low-level programming

model, thereby enabling the exploitation of parallel execution opportunities across loops. All other existing solutions leave loops in two separate contexts, which hinders the full exploitation of parallelism.

FractalTensor captures the computational patterns underlying all stacked RNN networks through nested `scan`. The semantics of the aggregate pattern indicate that the first step differs from subsequent steps, leading to complicated data access behaviors. When `scan` are nested, the parser directly transforms the combinatorial conditional branching into separate data-parallel tasks. After parsing, the stacked LSTM is represented by 4 block nodes, while the stacked Grid RNN is represented by 8 block nodes. Reordering analysis is then applied to each block node if it involves complex data dependencies. Since array compute operators have explicit data parallelism semantics, all these transformations rely on identifying how parallel operators are nested, which can be read from code rather than complicated reasoning. Therefore, these transformations can scale to algorithm variants that share the same array compute operators.

6.4 Memory performance optimization

In this section, we present a case study that examines how access operators impact memory performance, with a particular focus on the attention mechanism and their variations. Attention and its variants exhibit memory-bound behaviors. Identifying data reuse and staging reusable data on high-speed memory offers a unique opportunity to improve memory performance. To validate the effectiveness of FractalTensor, we conduct experiments comparing the total bytes of data traffic across the GPU's three memory hierarchies. The results are presented in Table 7. For FlashAttention, FractalTensor's memory usage is comparable to the handcrafted implementation. In the blocked sparse attention BigBird, FractalTensor reduces memory traffic significantly, resulting in reductions of DRAM to 43.8%, L1 cache to 47.2%, and L2 cache to 43.5% compared to the best baseline.

Table 7. Profiling results for ① FlashAttention ② BigBird algorithm on A100. The metrics refer to the total bytes of access to GPU DRAM, L1 cache, and L2 cache, respectively.

	Methodology	DRAM Access(GB)	L1 Access (GB)	L2 Access (GB)
①	FractalTensor	3.99	18.03	23.64
	Triton	4.01	18.22	24.09
	FlashAttention-2	4.13	22.21	27.07
	CUTLASS	3.98	73.01	78.96
②	FractalTensor	12.36	17.69	28.85
	Triton	28.21	37.45	66.37
	PyTorch	47.49	66.46	114.4
	TVM	108.9	456.8	622.8

We conduct a further investigation into the source of the performance improvement. By replacing the softmax with its online version, FlashAttention can compute the entire multi-head attention (MHA) tile by tile, enabling the sharing of intermediate results on high-speed memory. We view FlashAttention as a parallel algorithm for MHA rather than a specific DNN architecture. This shift in perspective from describing the connectivities of processing units to inventing parallel algorithms to process blocked data (i.e., a tile) is significant. To express this algorithm, a nested map operator with reduce is necessary. The FlashAttention algorithm is inherently unfit for single-level data parallelism. All existing methods fail to express the algorithm, resulting in the state-of-the-art implementation being a handcrafted CUDA kernel. However, composing the FlashAttention algorithm using array compute operator nesting is straightforward in FractalTensor, as demonstrated in the Appendix B. The key source of performance improvement comes from the fact that if users can naturally describe this algorithm, almost a direct translation that partitions the buffer node further over the memory hierarchy can help achieve the performance benefits intended by FlashAttention.

Compared to the standard MHA, the blocked sparse attention BigBird [38] has three attention components. The key performance improvement arises from the fact that the main computationally intensive task in BigBird is the windowed attention. FractalTensor maintains a logical form of data access using access map. Data movement can be deferred until the batched matrix multiplication kernel computes on a higher memory hierarchy, thereby reducing the overhead of redundant data movement. Table 7 shows that TVM and PyTorch, dictated by the DAG approach, require algorithm researchers to organize data as large tensors, and the resulting operators directly materialize the data movements. Tensors are scanned back and forth, with intermediate results all materialized on GPU DRAM. This introduces significant data access overhead and memory consumption. Profiling reveals that in PyTorch's DAG approach, the operators that do not compute but merely move data contribute 20% to 40% of the total execution time under different shapes. Triton, on the other hand, has expressive power almost equivalent to CUDA's SIMT programming. If users carefully design their parallel implementations, the data traffic over the memory hierarchy can be reduced through implicit implementation.

7 Discussion

FractalTensor is designed to support a wide range of deep neural networks, including RNN, CNN, Transformer, and their variants. As DNNs are still fast evolving, new model architectures like Mamba [14], RWKV [25], and RetNet [30] are emerging. Existing optimizations are difficult to be applied to these models directly. FractalTensor is well-positioned to

support these new designs in an efficient way. In the future, we plan to apply FractalTensor to the new models.

Although FractalTensor can support CNN, we do not implement CNN since the mainstream CNNs have already been well-optimized through low-level kernel optimizations (e.g., CUDA). Thus a FractalTensor-based implementation is less likely to yield additional gains. We leave it as future work.

In addition, the current access patterns of FractalTensor are determined at compile-time and are either affine or quasi-affine. Some dynamic sparse attention models, e.g., changing attending positions on runtime values, require dynamic access patterns. This necessitates a more sophisticated access map, which complicates program analysis. In such cases, FractalTensor needs to rely on complimentary optimization techniques, e.g., PIT [44].

The programming model of FractalTensor focuses on the logic-level expression of DNNs. The programming code is hardware independent. Currently FractalTensor's underlying compiler only supports the generation of NVIDIA CUDA codes. It is possible to extend the compiler support AMD GPUs or even new types of accelerators like Cerebras [8].

Finally, it is also possible to extend FractalTensor to a distributed setting to support typical parallel patterns like model parallelism and pipeline parallelism. This requires the compiler to extract the ETDG, optimize the ETDG up to the architecture-independent phase, and finally generate a flattened single-level DAG. Existing parallel deep learning frameworks like Alpa [42] or nnScaler [20] can then use the generated DAG to scale out the model.

8 Related Works

As discussed in previous sections extensively, there exists a long-standing conflict between expressiveness and efficiency for DNN programming.

To achieve expressiveness, deep learning frameworks like TensorFlow [2] and PyTorch [24] use a host language like Python to develop DNNs. They all rely on an extensive set of operators to achieve efficiency. With the introduction of operators, TensorFlow and PyTorch codes can be compiled into a DAG through solutions like TensorFlow AutoGraph[24] or PyTorch JIT[22], which produce efficient codes. The expressiveness of this method depends on the type of operators, while developing efficient operators are costly. Thus the support of operators is limited, which inevitably sacrifices expressiveness. To alleviate the issue, some incorporate control-flow operators to support dynamic DNNs [37]. Nonetheless, such single-level DAGs buries the inherently nested control loops and the sophisticated data access patterns, making the global analysis difficult, if not impossible.

Domain-specific languages (DSLs) are popular methods striving to offer a better trade-off between expressiveness

and efficiency [6, 10, 33, 34]. The solutions like Tensor Comprehension [10, 34], TVM [10, 27], Ansor [41], and EinNet [43] employ an Einstein summation-like tensor expression language to express numerical computations across tensors. However, the end-to-end program is difficult to be implemented entirely by the tensor expression. Thus the scope of these approaches are often constrained within one or several adjacent tensor operators [29, 41, 45]. Moreover, during compilation, the tensor expression is often directly transformed into low-level imperative loop nests, losing critical parallel and data dependency information. These imperative loops often incur a combinatorial complexity for polyhedral-like techniques to find a reasoning scheduling. Triton [33] is another DSL that simplifies the SIMT programming model of CUDA, a DSL dedicated for NVIDIA devices. Like TVM, the scope of Triton is still at the granularity of tensor operators.

9 Conclusions

The current DNN software stack commonly relies on the abstraction of a single-level DAG of tensor operators to achieve efficient computation while preserving expressiveness. However, the operator abstraction introduces arbitrary boundaries with ad hoc semantics, impeding the analysis and optimizations that fully exploit the data parallelism and data locality originated from DNN algorithms globally. To address the inherent limitation of the operator abstraction based on single-level DAG, we propose FractalTensor, a high-level functional programming interface based on a list-based nested abstract tensor structure. In response to the multi-dimensional nature of DNN computation, FractalTensor organizes the DNN computation following the inherent pattern of nested data parallelism, which can be naturally transformed into an Extended Task Dependency Graph (ETDG), an intermediate representation of hierarchical multidimensional dataflow graph. ETDG preserves the holistic view of data dependency across different nested level without forcing a direct (and inefficient) materialization of computation and data movement. We show that ETDG can be effectively lowered to efficient implementations through graph coarsening, data reordering, and access materialization, achieving superior performance compared implementations relied on single-level DAG of operators.

Acknowledgments

We thank all the anonymous reviewers for their insightful feedback and our shepherd for the guidance during the preparation of our camera-ready submission. Ying Cao is the corresponding author.

References

- [1] 2024. *FractalTensor*. <https://github.com/microsoft/FractalTensor>

- [2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. USENIX Association, 265–283. <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi>
- [3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley. <https://www.worldcat.org/oclc/12285707>
- [4] Randy Allen and Ken Kennedy. 2001. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann.
- [5] Jeremy Appleyard, Tomás Kociský, and Phil Blunsom. 2016. Optimizing Performance of Recurrent Neural Networks on GPUs. *CoRR* abs/1604.01946 (2016). arXiv:1604.01946 <http://arxiv.org/abs/1604.01946>
- [6] 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*. IEEE, 193–205. <https://doi.org/10.1109/CGO.2019.8661197>
- [7] Paul Barham and Michael Isard. 2019. Machine Learning Systems are Stuck in a Rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019, Bertinoro, Italy, May 13-15, 2019*. ACM, 177–183. <https://doi.org/10.1145/3317550.3321441>
- [8] Cerebras. 2021. *Cerebras Systems: Achieving Industry Best AI Performance Through A Systems Approach*. White Paper. Cerebras. <https://8968533.fs1.hubspotusercontent-na1.net/hubfs/8968533/Whitepapers/Cerebras-CS-2-Whitepaper.pdf>
- [9] Shiyu Chang, Yang Zhang, Wei Han, Mo Yu, Xiaoxiao Guo, Wei Tan, Xiaodong Cui, Michael Witbrock, Mark A. Hasegawa-Johnson, and Thomas S. Huang. 2017. Dilated Recurrent Neural Networks. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. 77–87. <https://proceedings.neurips.cc/paper/2017/hash/32bb90e8976aab5298d5da10fe66f21d-Abstract.html>
- [10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. USENIX Association, 578–594. <https://www.usenix.org/conference/osdi18/presentation/chen>
- [11] 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). arXiv:1410.0759 <http://arxiv.org/abs/1410.0759>
- [12] Tri Dao. 2024. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. <https://openreview.net/forum?id=mZn2Xyh9Ec>
- [13] Alain Darte, Yves Robert, and Frédéric Vivien. 2000. *Scheduling and automatic parallelization*. Birkhäuser.
- [14] Albert Gu and Tri Dao. 2023. Mamba: Linear-Time Sequence Modeling with Selective State Spaces. *CoRR* abs/2312.00752 (2023). <https://doi.org/10.48550/ARXIV.2312.00752> arXiv:2312.00752
- [15] Troels Henriksen and Cosmin Eugen Oancea. 2013. A T2 graph-reduction approach to fusion. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing, Boston, MA, USA, FHPC@ICFP 2013, September 25-27, 2013*. ACM, 47–58. <https://doi.org/10.1145/2502323.2502328>
- [16] Troels Henriksen, Niels G. W. Serup, Martin Elsmann, Fritz Henklein, and Cosmin E. Oancea. 2017. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. ACM, 556–571. <https://doi.org/10.1145/3062341.3062354>
- [17] Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David R. Kaeli. 2011. Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures. *IEEE Trans. Parallel Distributed Syst.* 22, 1 (2011), 105–118. <https://doi.org/10.1109/TPDS.2010.107>
- [18] Nal Kalchbrenner, Ivo Danihelka, and Alex Graves. 2016. Grid Long Short-Term Memory. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. <http://arxiv.org/abs/1507.01526>
- [19] Leslie Lamport. 1974. The Parallel Execution of DO Loops. *Commun. ACM* 17, 2 (1974), 83–93. <https://doi.org/10.1145/360827.360844>
- [20] Zhiqi Lin, Youshan Miao, Quanlu Zhang, Fan Yang, Yi Zhu, Cheng Li, Saeed Maleki, Xu Cao, Ning Shang, Yilei Yang, Weijiang Xu, Mao Yang, Lintao Zhang, and Lidong Zhou. 2024. nnScaler: Constraint-Guided Parallelization Plan Generation for Deep Learning Training. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*. USENIX Association, 347–363. <https://www.usenix.org/conference/osdi24/presentation/lin-zhiqi>
- [21] 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>
- [22] Dan Moldovan, James M. Decker, Fei Wang, Andrew A. Johnson, Brian K. Lee, Zachary Nado, D. Sculley, Tiark Rompf, and Alexander B. Wiltschko. 2019. AutoGraph: Imperative-style Coding with Graph-based Performance. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*. mlsys.org. <https://proceedings.mlsys.org/book/272.pdf>
- [23] NVIDIA Corporation. 2022. cuBLAS. <https://developer.nvidia.com/cublas>
- [24] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. 8024–8035. <https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html>
- [25] Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Acadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Leon Derczynski, Xingjian Du, Matteo Grella, Kranthi Kiran GV, Xuzheng He, Haowen Hou, Przemyslaw Kazienko, Jan Kocon, Jiaming Kong, Bartłomiej Koptyra, Hayden Lau, Jiaju Lin, Krishna Sri Ipsit Mantri, Ferdinand Mom, Atsushi Saito, Guangyu Song, Xiangru Tang, Johan S. Wind, Stanislaw Wozniak, Zhenyuan Zhang,

- Qinghua Zhou, Jian Zhu, and Rui-Jie Zhu. 2023. RWKV: Reinventing RNNs for the Transformer Era. In *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6–10, 2023*. Association for Computational Linguistics, 14048–14077. <https://doi.org/10.18653/V1/2023.FINDINGS-EMNLP.936>
- [26] William W. 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>
- [27] Jared Roesch, Steven Lyubomirsky, Marisa Kirisame, Josh Pollock, Logan Weber, Ziheng Jiang, Tianqi Chen, Thierry Moreau, and Zachary Tatlock. 2019. Relay: A High-Level IR for Deep Learning. *CoRR* abs/1904.08368 (2019). arXiv:1904.08368 <http://arxiv.org/abs/1904.08368>
- [28] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision. *CoRR* abs/2407.08608 (2024). <https://doi.org/10.48550/ARXIV.2407.08608> arXiv:2407.08608
- [29] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. 2023. Welder: Scheduling Deep Learning Memory Access via Tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10–12, 2023*. USENIX Association, 701–718. <https://www.usenix.org/conference/osdi23/presentation/shi>
- [30] Yutao Sun, Li Dong, Shaohan Huang, Shuming Ma, Yuqing Xia, Jilong Xue, Jianyong Wang, and Furu Wei. 2023. Retentive Network: A Successor to Transformer for Large Language Models. *CoRR* abs/2307.08621 (2023). <https://doi.org/10.48550/arXiv.2307.08621> arXiv:2307.08621
- [31] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8–13 2014, Montreal, Quebec, Canada*. 3104–3112. <https://proceedings.neurips.cc/paper/2014/hash/a14ac55a4f27472c5d894ec1c3c743d2-Abstract.html>
- [32] 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 *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 – 17, 2022*. ACM, 872–887. <https://doi.org/10.1145/3519939.3523448>
- [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*. ACM, 10–19. <https://doi.org/10.1145/3315508.3329973>
- [34] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2020. The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically. *ACM Trans. Archit. Optim.* 16, 4 (2020), 38:1–38:26. <https://doi.org/10.1145/3355606>
- [35] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4–9, 2017, Long Beach, CA, USA*. 5998–6008. <https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fb0d53c1c4a845aa-Abstract.html>
- [36] Michael E. Wolf and Monica S. Lam. 1991. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Trans. Parallel Distributed Syst.* 2, 4 (1991), 452–471. <https://doi.org/10.1109/71.97902>
- [37] Yuan Yu, Martin Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, Michael Isard, Manjunath Kudlur, Rajat Monga, Derek Gordon Murray, and Xiaoqiang Zheng. 2018. Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23–26, 2018*. ACM, 18:1–18:15. <https://doi.org/10.1145/3190508.3190551>
- [38] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontañón, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. 2020. Big Bird: Transformers for Longer Sequences. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6–12, 2020, virtual*. <https://proceedings.neurips.cc/paper/2020/hash/c8512d142a2d849725f31a9a7a361ab9-Abstract.html>
- [39] Jie Zhao, Siyuan Feng, Xiaoqiang Dan, Fei Liu, Chengke Wang, Sheng Yuan, Wenyan Lv, and Qikai Xie. 2023. Effectively Scheduling Computational Graphs of Deep Neural Networks toward Their Domain-Specific Accelerators. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10–12, 2023*. USENIX Association, 719–737. <https://www.usenix.org/conference/osdi23/presentation/zhao>
- [40] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, Peng Di, Kun Zhang, and Xuefeng Jin. 2021. AKG: automatic kernel generation for neural processing units using polyhedral transformations. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20–25, 2021*. ACM, 1233–1248. <https://doi.org/10.1145/3453483.3454106>
- [41] 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. (2020), 863–879. <https://www.usenix.org/conference/osdi20/presentation/zheng>
- [42] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11–13, 2022*. USENIX Association, 559–578. <https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin>
- [43] Liyan Zheng, Haojie Wang, Jidong Zhai, Muyan Hu, Zixuan Ma, Tuowei Wang, Shuhong Huang, Xupeng Miao, Shizhi Tang, Kezhao Huang, and Zhihao Jia. 2023. EINNET: Optimizing Tensor Programs with Derivation-Based Transformations. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10–12, 2023*. USENIX Association, 739–755. <https://www.usenix.org/conference/osdi23/presentation/zheng>
- [44] Ningxin Zheng, Huiqiang Jiang, Quanlu Zhang, Zhenhua Han, Lingxiao Ma, Yuqing Yang, Fan Yang, Chengruidong Zhang, Lili Qiu, Mao Yang, and Lidong Zhou. 2023. PIT: Optimization of Dynamic Sparse Deep Learning Models via Permutation Invariant Transformation. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23–26, 2023*. ACM, 331–347. <https://doi.org/10.1145/3600006.3613139>
- [45] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, Shuaiwen Leon Song, and Wei Lin. 2022. AStitch: enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures. In *ASPLOS '22: 27th ACM*

International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022. ACM, 359–373. <https://doi.org/10.1145/3503222.3507723>

- [46] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. 2022. ROLLER: Fast and Efficient Tensor Compilation for Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*. USENIX Association, 233–248. <https://www.usenix.org/conference/osdi22/presentation/zhu>

Appendices

The following appendices contain supplemental material to the main paper. However, the material in these appendices has not received the peer review as the main paper.

Appendix A The abstract syntax

We present the abstract syntax for the FractalTensor programming interface. This interface incorporates constructs such as list declaration, lambda expression, function evaluation, and array indexing, which is permitted exclusively on the right-hand side of an expression. The syntax can be implemented in any programming language to program a nested dataflow graph as a single function decomposed into multiple primitive functions.

```

<CompUnit>      ::= (<VarDecl> | <FuncDef>)
<Shape>         ::= ' { N {', N } ' '
<BaseType>      ::= <Int> | <Float>
<Int>           ::= 'Int' ' ('N')
<Float>         ::= 'Float' ' ('N')
<Tensor>        ::= 'Tensor' ' ('<BaseType>', '<Shape>')
<FractalTensor> ::= 'FractalTensor' ' ('<Tensor>')
                | 'FractalTensor' ' ('<FractalTensor>')
<Type>          ::= <BaseType>
                | <Tensor>
                | <FractalTensor>
<VarDecl>       ::= 'Ident' ':' '<Type>'
<Lambda>        ::= '{Ident}' '=>' ' { {<Expr> } } '
<FuncDef>       ::= 'Ident' ' (' {<VarDecl> } ) ' -> ' (<Type> {':<Type> } ) ' '
                {<Expr> } ' '
<UnaryOp>       ::= '-' | 'σ' | 'tanh' | 'exp' | ...
<BinaryOp>      ::= '+' | '-' | '*' | '/' | '@' | ...
<Expr>          ::= <VarDecl>
                | 'Ident' ' [ { N {', N } ' ] ' (* Array Indexing *)
                | <UnaryOp>
                | <BinaryOp>
                | 'map' ' ('<Ident> [':<FractalTensor>']', '<Lambda>')
                | 'reduce' ' ('<Ident> [':<FractalTensor>'], <Ident> [':<Type>']
                ', '<Lambda>')
                | 'fold' ' ('<Ident> [':<FractalTensor>'], <Ident> [':<Type>']
                ', '<Lambda>')
                | 'scan' ' ('<Ident> [':<FractalTensor>'], <Ident> [':<Type>']
                ', '<Lambda>')

```

Appendix B More exemplary applications

Listings 2, 3, and 4 outline the stacked LSTM, flash attention, and blocked sparse attention algorithm BigBird (with the random attention component omitted for simplicity), respectively, as array compute and access operator nesting. ETGD serves as a graphical representation of this syntax form.

Listing 2. The stacked LSTM network.

```

xss: [32, 128] float32 [1, 512] = ...
wss: [5, 4] float32 [512, 512] = ...
uss: [5, 4] float32 [512, 512] = ...
bss: [5, 4] float32 [1, 512] = ...

hsss, csss = xss.map xs =>
  zip(uss, wss, bss).foldl xs, (ss, (ws, us, bs)) =>
    ss.scanl (0, 0), ((c, h), x) =>
      gs = zip(ws, us, bs).map (w, u, b) =>
        x@w + h@u + b
      gs = gs[0:3].map x => sigmoid(x)
      c = gs[1] * c + gs[0] * tanh(gs[3])
      h = gs[2] * tanh(c)

```

Listing 3. The flash attention algorithm.

```

qsss:[32, 16, 2048] float32 [32, 128] = ...
ksss:[16, 16, 4096] float32 [32, 128] = ...
vsss:[16, 16, 4096] float32 [32, 128] = ...

ooss = zip(qsss, ksss, vsss).map (qss, kss, vss) =>
  zip(qss, kss, vss).map (qs, ks, vs) =>
    qs.map q =>
      zip(ks, vs).reduce (-∞, 0, 0), ((m, s, o), k, v) =>
        t1 = q@k
        t2 = max(t1)
        t3 = exp(t1 - t2)
        t4 = t3@v
        t5 = sum(t3, dim = 1)
        mt = max(t2, m)
        t6 = exp(mt - m)
        t7 = exp(t2 - mt)
        st = t6 * s + t7 * t5
        ot = (o * s * t6 + t7 * t4) / s

```

Listing 4. The blocked sparse attention algorithm: BigBird

```

// blocked queries, keys, values
qss:[16, 64] float32 [32, 512] = ...
kss:[16, 64] float32 [32, 512] = ...
vss:[16, 64] float32 [32, 512] = ...

oos = zip(qss, kss, vss).map qs, ks, vs =>
  // windowed attention
  wks, wvs = zip(ks, vs).shifted_slide(window=3)
  wqk = zip(qs[2:-2], wks[2:-2]).map q, k => q@k.T

  // left and right global attention
  gqk1 = qs[2:-2].map q => q@ks[0].T
  gqk2 = qs[2:-2].map q => q@ks[-1].T

  // user-defined norm function
  scores = zip(gqk1, wqk, gqk2).map gl, w, gr =>
    softmax([gl:w:gr])

  // weighted values
  go1 = scores.map v => v[0] @ vs[0]
  go2 = scores.map v => v[-1] @ vs[-1]
  wo = zip(scores, wqk[2:-2]).map s, v => s[1:-1] @ v
  os = zip(gy1, gy2, oy).map v1, v2, v3 => v1 + v2 + v3

```