EinNet: Optimizing Tensor Programs with Derivation-Based Transformations

Liyan Zheng[◇] Haojie Wang Jidong Zhai Muyan Hu Zixuan Ma Tuowei Wang
Shuhong Huang Xupeng Miao[†] Shizhi Tang Kezhao Huang Zhihao Jia[†]
Tsinghua University [†]*Carnegie Mellon University*

Abstract

Boosting the execution performance of deep neural networks (DNNs) is critical due to their wide adoption in real-world applications. However, existing approaches to optimizing the tensor computation of DNNs only consider transformations *representable* by a fixed set of predefined tensor operators, resulting in a highly restricted optimization space. To address this issue, we propose EinNet, a *derivation-based* tensor program optimizer. EinNet optimizes tensor programs by leveraging transformations between *general* tensor algebra expressions and automatically creating new operators desired by transformations, enabling a significantly larger search space that includes those supported by prior works as special cases. Evaluation on seven DNNs shows that EinNet outperforms existing tensor program optimizers by up to 2.72× (1.52× on average) on NVIDIA A100 and up to 2.68× (1.55× on average) on NVIDIA V100. EinNet is publicly available at https://github.com/InfiniTensor/InfiniTensor.

1 Introduction

Fast execution of deep neural networks (DNNs) is critical in a variety of tasks, such as autonomous driving [16, 21, 26], object detection [15, 18], speech recognition [5, 17], and machine translation [37, 39]. A DNN is generally represented as a *tensor program*, which is a directed acyclic graph containing tensor operators (e.g., convolution, matrix multiplication) performed on a set of tensors (i.e., n-dimensional arrays).

To improve the runtime performance of a DNN, existing frameworks (TensorFlow [3], PyTorch [31], and TensorRT [35]) rely on *manually-designed* rules to map an input tensor program to *expert-written* kernel libraries. Although widely used, these approaches require extensive engineering efforts and miss optimization opportunities hard to manually discover. To address these problems, recent works have proposed a variety of *automated* approaches that optimize DNN computation by searching over a set of candidate program transformations or generating high performance kernels on specific hardware. We classify these works into two categories based on their search spaces.

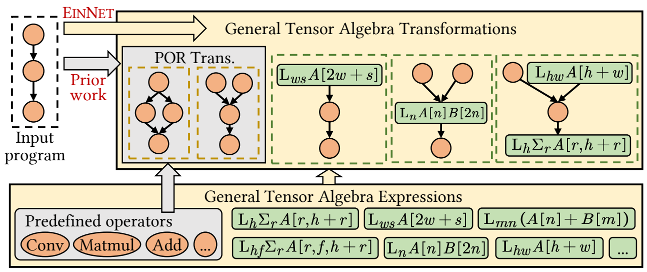


Figure 1: Comparing EinNet's search space with that of prior work. "POR Trans." indicates predefined operator representable transformations.

The first category of work, including TVM [7] and Ansor [40], is motivated by Halide's idea of compute/schedule separation [33] and optimizes tensor programs at the *operator level*. For a given tensor operator, they automatically generate high-performance kernels by searching over *schedules*, each of which specifies an architecture-dependent execution plan on particular hardware. To optimize the graph structure of a tensor program, TVM and Ansor greedily apply a fixed set of expert-designed program transformations.

The second category of work optimizes tensor programs using *graph-level* transformations, which reorganize the DNN computation in more efficient ways. As two representative systems, TASO [20] and PET [38] adopt a superoptimization-based approach to discovering graph transformations. They generate candidate graph transformations by enumerating all possible graphs over a given set of tensor operators up to a fixed size, and search to apply these generated transformations to an input tensor program.

Both operator- and graph-level optimizers only consider program transformations whose nodes are tensor operators predefined by optimizer developers, as shown in the grey box of Figure 1. We call these transformations *predefined operator representable* (POR) transformations. Despite the fact that

[◇]Tsinghua University and BNRist

existing tensor program optimizers only use POR transformations to optimize tensor programs, POR transformations only exhibit limited opportunities for performance optimizations. In this paper, we propose to explore *general* tensor algebra transformations whose nodes are *general* tensor operators [1]. Compared to POR transformations, general tensor algebra transformations constitute a significantly larger optimization space, which includes POR transformations as special cases, as shown in the yellow box of Figure 1.

To discover general tensor algebra transformations, we present EINNET, a *derivation-based* tensor program optimizer. A key difference between EINNET and prior work (e.g., TASO and PET) is that EINNET reveals operator computation semantics in automated graph transformations by applying derivation rules to tensor algebra expressions. By deriving computation at the expression level, EINNET can reorganize computation into arbitrary tensor expressions and map them into both predefined operators with highly optimized implementations and new auto-generated operators desired by derivations. Expression-level derivations allow EINNET to discover a variety of novel program transformations missing in existing frameworks, since these transformations require highly customized tensor operators not predefined in existing optimizers. Example transformations newly discovered by EINNET include: (1) modifying the computation semantics of an operator to improve efficiency, (2) replacing inefficient operators with highly-optimized alternatives and customized tensor operators to bridge the gap, and (3) aggressively reorganizing computation graphs to enable subsequent graph-level optimizations.

EINNET mainly addresses the following three challenges:

The first challenge is automatically discovering transformation opportunities between general expressions. TASO and PET only consider a *fixed* set of predefined operators, but there are infinitely many possible general expressions. Hence, directly applying superoptimization (i.e., enumerating all possible graphs over general expressions) is infeasible. EINNET addresses this challenge by presenting a *derivation-based* mechanism that automatically transforms an expression to equivalent alternatives by applying a collection of derivation rules. Since most derived expressions cannot be simply represented as predefined operators, we introduce *eOperators* (expression as an operator) to represent non-POR computation. eOperators enable EINNET to discover a variety of optimizing transformations between expressions.

The second challenge is converting expressions back to kernels that can be executed on DNN accelerators, a process we term *expression instantiation*. Although existing kernel generators (e.g., TVM and Ansor) can generate kernels for a given expression, doing so is suboptimal since existing vendor-provided libraries (e.g., cuDNN [10] and cuBLAS [11]) offer highly-optimized kernels for a set of predefined operators. EINNET opportunistically matches a part of an expression with predefined operators to take advantage of the highly-optimized kernels from vendor-provided libraries; the remaining part of the expression is lowered to an off-the-shelf kernel generator (i.e., TVM [7]).

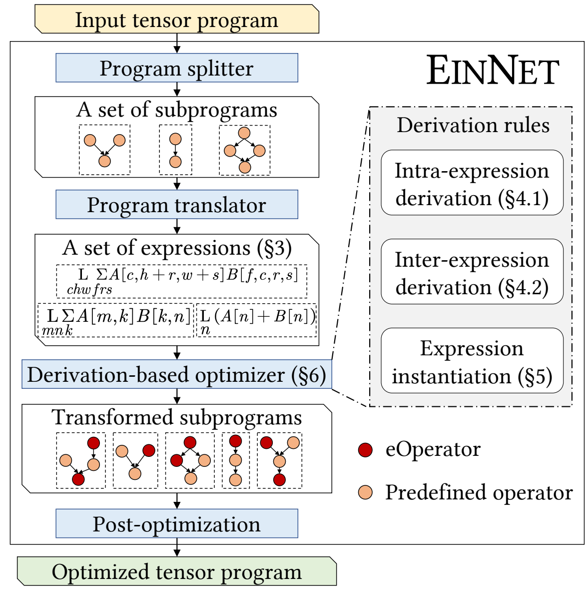


Figure 2: EINNET overview

The third challenge is quickly finding optimizing transformations in the search space of general tensor algebra transformations. In particular, optimizing a tensor program normally requires applying a long sequence of derivation rules (e.g., up to 12 in our evaluation), which cannot be efficiently discovered by a traversal-based search algorithm. To address this challenge, EINNET employs a two-stage search approach to applying derivations, where an *explorative derivation* stage considers applying all possible derivations to the current expression to create a comprehensive collection of expressions, and a *converging derivation* stage uses *expression distance* to guide the search towards promising candidates. This distance-guided approach allows EINNET to discover complex optimizations requiring long sequences of derivations under a reasonable search budget.

We evaluate EINNET on seven real-world DNN models covering a variety of machine learning tasks. We compare EINNET with state-of-the-art frameworks on two GPU platforms, NVIDIA A100 and V100. Evaluation shows that EINNET is up to 2.72× faster than existing tensor program optimizers. The significant performance improvement indicates that EINNET benefits from the new optimization opportunities enabled by derivation-based optimizations.

This paper makes the following contributions:

- We extend the POR optimization space to the general tensor algebra optimization space by combining operator computation semantics and computation graphs with tensor algebra expressions.

[1] An operator is a tensor operator if it can be represented using the tensor algebra expression in Equation (1)

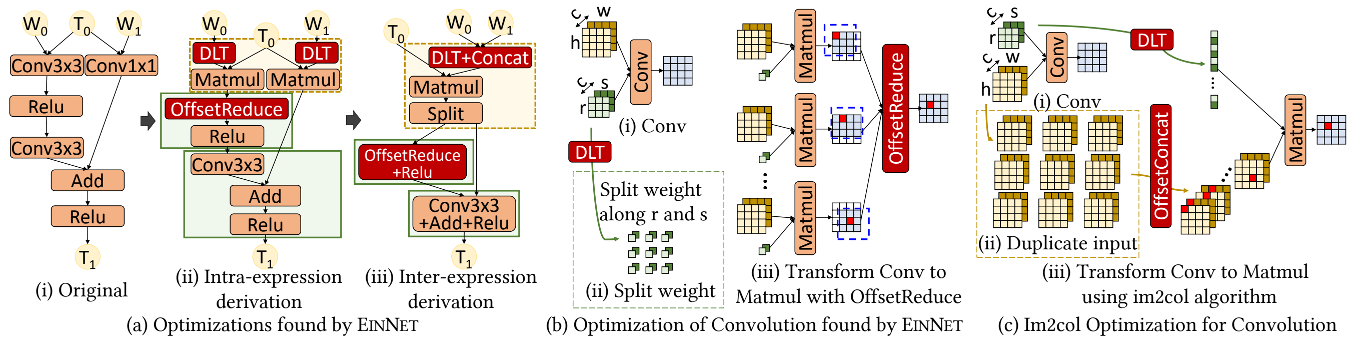


Figure 3: Optimization examples of EINNET. Figure (a) shows the optimization that transforms a `Conv3×3` operator into a `Matmul` and an eOperator `OffsetReduce`, and a `Conv1×1` operator into a `Matmul`. Then, inter-expression derivation is performed to fuse multiple operators into one. Figure (b) shows the optimization details performed by EINNET for the `Conv3×3` operator, which first splits the weight tensor into 9 tensors, then multiplies each tensor with the input, and finally adds the nine results together with certain offsets (illustrated by the dashed boxes and red blocks). The `Matmuls` in Figure (b) are further fused into a single one. As a comparison, Figure (c) shows the typical im2col [36] optimization for `Conv`, which performs a different transformation from that in Figure (b) and can also be automatically found by EINNET.

- We present the first attempt to explore a significantly larger expression search space using a derivation-based mechanism.
- We build EINNET, an implementation of the above techniques with over 23K lines of C++ and Python code, which achieves up to 2.72× speedup over existing tensor program optimizers.

2 Overview and Motivating Example

Figure 2 shows an overview of EINNET, a tensor program optimizer with derivation-based transformations. For an input tensor program, EINNET first splits it into multiple subprograms consisting of predefined operators. Each subprogram is translated to a tensor algebra expression (§3) by a program translator. Then, EINNET's *derivation-based optimizer* uses different derivation rules, including inter- and intra-expression derivation rules (§4) and expression instantiation rules (§5), to generate optimized subprograms for each expression, which consists of both predefined operators and eOperators. Finally, EINNET selects the best discovered transformation for each subprogram and post-optimizes the expressions to construct an efficient tensor program (§6).

Motivating example. As a motivating example, Figure 3(a) shows an optimization found by EINNET. It first performs an intra-expression derivation to transform convolutions into matrix multiplications, and then performs inter-expression derivation to fuse multiple operators into one. The red operators, such as `OffsetReduce`, `DLT` (data layout transformation), and `OffsetReduce+Relu`, are eOperators automatically discovered and generated by EINNET. Figure 3(b) shows the details of the new optimization discovered by EINNET for `Conv3x3` in Figure 3(a). Figure 3(c) illustrates the classic im2col [36] optimization for convolution, which is widely implemented in existing libraries and also covered by the automatic optimization space of EINNET. Different from copying input tensors for the kernel size times in im2col, the newly discovered transformation copies output tensors the same number of times. It can be more efficient when the output size is smaller than the input size, and achieves a 2× speedup compared with cuDNN on the NVIDIA A100 GPU for certain convolutions in ResNet-18 [19] in our evaluation.

Existing tensor program optimizers cannot automatically discover such transformations because: (1) the transformations require eOperators (e.g., adding intermediate tensors with offsets), which are outside of the POR transformation space explored by superoptimization-based frameworks such as TASO [20] and PET [38], and (2) the transformations modify the computation semantics instead of the schedule, and thus cannot be found by schedule-based optimizers like TVM [7] and Ansor [40].

3 Tensor Algebra Expression

EINNET represents a tensor program as *tensor algebra expressions*, which defines how to compute each element of output tensors from input tensors. Figure 4 shows the expression of multiplying three matrices (i.e., $A \times B \times C$). We now describe the components of an expression. For simplicity, we assume an expression has one output. EINNET's expression can be easily generalized to multiple outputs.

Traversal and summation notations. A *traversal notation*, denoted as $\mathbb{L}_{x=x_0}^{x_1}$, consists of an *iterator* x and an *iterating space* $[x_0, x_1)$. The traversal notation corresponds to a dimen-

$$\mathop{\mathrm{L}}_{c=0}^{C}\mathop{\mathrm{L}}_{r=0}^{R}\sum_{k_0=0}^{K}\sum_{k_1=0}^{K} A[c,k_0]B[k_0,k_1]C[k_1,r] \tag{a}$$

$$= \mathop{\mathrm{L}}_{c=0}^{C}\mathop{\mathrm{L}}_{r=0}^{R}\sum_{k_0=0}^{K}\boxed{\left\{\mathop{\mathrm{L}}_{c'=0}^{C}\mathop{\mathrm{L}}_{k_2=0}^{K}\sum_{k_1=0}^{K} A[c',k_1]B[k_1,k_2]\right\}}[c,k_0]C[k_0,r] \tag{b}$$

Traversal notation Summation notation Scope

Figure 4: A tensor algebra expression example for two matrix multiplications $A \times B \times C$. The red box highlights a *scope* that instantiates the intermediate result of $A \times B$.

sion of the output tensor, where the iterating space is the range of the dimension. The order of the traversal notations indicates the layout of the output tensor. For example, in Figure 4, $\mathrm{L}_{c=0}^{C}$ followed by $\mathrm{L}_{r=0}^{R}$ indicates that the expression's output is a two-dimensional tensor with a shape $C \times R$.

A *summation notation*, denoted as $\sum_{x=x_0}^{x_1}$, computes the summation iterating over dimension x with $\{x_0, x_0 + 1, \ldots, x_1 - 1\}$, which is hereinafter represented by a range $[x_0, x_1)$ for brevity. Note that an EINNET expression under different orders of summation notations are considered the same but corresponds to different *schedules* of an expression. Therefore, it is excluded from the expression search space.

Tensors are indexed by an *arithmetic combination* of multiple iterators, including $add(+)$, $sub(-)$, $mul(*)$, $div(/)$ and $mod(\%)$. For simplicity, we may merge multiple iterators into an iterator vector, whose iterating space can be denoted by an integer set or omitted in the expression. For example, $\mathrm{L}_{c=0}^{C}\mathrm{L}_{r=0}^{R}$ can be represented as L_{cr} or $\mathrm{L}_{\vec{x}}^{\mathbb{X}}$, where $\vec{x} = (c, r)$ is the iterator vector, and $\mathbb{X} = \mathbb{C} \times \mathbb{R}$ is the iterating space.

Scope. For a tensor program with multiple operators (e.g., two consecutive matrix multiplications $A \times B \times C$), a common optimization is to instantiate and reuse intermediate results (e.g., caching the output of $A \times B$), which avoids repetitive computation for these results. EINNET introduces *scopes* to represent the instantiation of intermediate results to reuse them later. Formally, a tensor algebra expression is a *scope*, denoted by a surrounding $\{\}$, if the output of the expression is instantiated into a tensor, which allows subsequent computation to refer to this tensor and therefore avoids repeated computation. In Figure 4(b), the expression corresponding to $A \times B$ is a scope, allowing subsequent computation to directly refer to the output of this expression. Many of EINNET's derivation rules are based on transformations between scopes, including generating new scopes from existing ones, transforming a scope to another form, and merging multiple scopes into one (§4). Transformations between scopes are essential to EINNET's optimizations.

Padding. Some computations access an input outside of its region, which we call paddings. E.g., a 3×3 convolution may have paddings. Paddings are set to 0 if not specified.

General format. We represent a one-scope expression as:

Table 1: Derivation rules for tensor algebra expressions.

Rules	Descriptions
Intra-expression derivation §4.1	
Summation splitting	Split summation from one scope into two
Variable substitution	Replace traversal iterators with new ones
Traversal merging	Merge two scopes by merging traversals
Boundary relaxing	Relax the range of iterators
Boundary tightening	Tighten the range of iterators
Inter-expression derivation §4.2	
Expression splitting	Split an expression into independent ones
Expression merging	Merge multiple independent expressions
Expression fusion	Fuse multiple dependent expressions
Expression instantiation §5	
Operator matching	Match a scope with predefined operators
eOperator generation	Generate an eOperator for a scope

$$\mathop{\mathrm{L}}_{\vec{x}}^{\mathbb{X}}\sum_{\vec{y}}^{\mathbb{Y}} f(\mathbf{T}[\tau(\vec{x},\vec{y})]) \tag{1}$$

where $\mathbf{T} = \{T_0, T_1, \ldots\}$ is a list of input tensors, $\tau(\vec{x},\vec{y})$ is the indexing function that computes element indexes for tensors in $\mathbf{T}$ using iterators $\vec{x}$ and $\vec{y}$, and f is the computation taking on the indexed elements of $\mathbf{T}$.

4 Derivation Rules

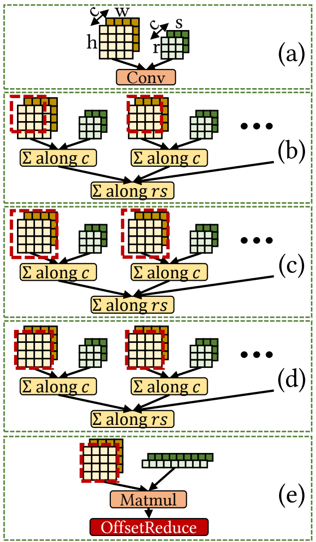
To discover highly-optimized expressions for an input tensor program, EINNET uses *derivation rules* to apply transformations on an input expression. Table 1 summarizes the derivation rules used by EINNET. Note that the *mathematical equivalence* of derivation rules guarantees the equivalence of derived expressions discovered by EINNET.

Different from schedule primitives of kernel generators that are designed to discover optimized schedules of a given expression on specific hardware, EINNET 's derivation rules focus on transform the computation semantics of tensor expressions, such as reorganizing computation into efficient operators.

4.1 Intra-Expression Derivation

Intra-expression derivation rules transform an expression into other functionally equivalent forms, which is essential for constructing a comprehensive search space of expressions for a tensor program. Figure 5 shows the optimization details in Figure 3(b). It splits the expression of `Conv3x3` into two parts, derives one part toward a predefined operator `Matmul`, and then converts the other part to an eOperator. We now describe these intra-expression derivation rules.

Summation splitting divides a summation notation $\sum_{\vec{s}}$ into two separate summations $\sum_{\vec{s}_1}$ and $\sum_{\vec{s}_2}$ and instantiates the result of the inner summation by converting it to a scope:



$$\mathcal{E}_1 \quad \mathop{L}_{hwf}\sum_{crs} A[h+r,w+s,c]K[r,s,f,c]$$

Summation splitting

$$\mathcal{E}_2 = \mathop{L}_{hwf}\sum_{rs}\Big\{\mathop{L}_{rshwf}\sum_{c} A[h+r,w+s,c]K[r,s,f,c]\Big\}[r,s,h,w,f]$$

Variable substitution: $t_1 = h+r, t_2 = w+s$

$$\mathcal{E}_3 = \mathop{L}_{hwf}\sum_{rs}\Big\{\mathop{L}_{rshwf}\Big\{\mathop{L}_{rsf}\mathop{L}_{t_1=r}^{H+r}\mathop{L}_{t_2=s}^{W+s}\sum_{c} A[t_1,t_2,c]K[r,s,f,c]\Big\}[r,s,f,h+r,w+s]\Big\}[r,s,h,w,f]$$

Boundary relaxing

$$\mathcal{E}_4 = \mathop{L}_{hwf}\sum_{rs}\Big\{\mathop{L}_{rshwf}\Big\{\mathop{L}_{rsf}\mathop{L}_{t_1=-1}^{H+1}\mathop{L}_{t_2=-1}^{W+1}\sum_{c} A[t_1,t_2,c]K[r,s,f,c]\Big\}[r,s,f,h+r,w+s]\Big\}[r,s,h,w,f]$$

Traversal merging

$$\mathcal{E}_5 = \mathop{L}_{hwf}\sum_{rs}\Big\{\mathop{L}_{rsf}\mathop{L}_{t_1=-1}^{H+1}\mathop{L}_{t_2=-1}^{W+1}\sum_{c} A[t_1,t_2,c]K[r,s,f,c]\Big\}[r,s,f,h+r,w+s]$$

Boundary tightening

$$\mathcal{E}_6 = \mathop{L}_{hwf}\sum_{rs}\Big\{\mathop{L}_{rsf}\mathop{L}_{t_1=0}^{H}\mathop{L}_{t_2=0}^{W}\sum_{c} A[t_1,t_2,c]K[r,s,f,c]\Big\}[r,s,f,h+r,w+s]$$

Operator matching: $T_1 = K(rsf,c) \times A(c,t_1t_2)$

$$\mathcal{E}_7 = \mathop{L}_{hwf}\sum_{rs} T_1[r,s,f,h+r,w+s]$$

$$\mathcal{E}_8 = T_2$$

eOperator generation: T_2 *is computed by a memory bound operator*

Figure 5: The derivation process of the example in Figure 3(b), which transforms `Conv` with `Matmul` and eOperators

$$\mathop{L}_{\vec{x}}\sum_{\vec{s}_1,\vec{s}_2} f(\mathbf{T}[\tau(\vec{x},\vec{s}_1,\vec{s}_2)]) \Rightarrow \mathop{L}_{\vec{x}}\sum_{\vec{s}_1}\Big\{\mathop{L}_{\vec{x}\vec{s}_1}\sum_{\vec{s}_2} f(\mathbf{T}[\tau(\vec{x},\vec{s}_1,\vec{s}_2)])\Big\}[\vec{x},\vec{s}_1]$$

where τ is a mapping from $(\vec{x},\vec{s}_1,\vec{s}_2)$ to an input position. EINNET divides the iterators of a summation into two disjoint groups, $\vec{s}_1$ and $\vec{s}_2$, which splits the summation into two nested scopes $\mathcal{S}_1$ and $\mathcal{S}_2$, where $\mathcal{S}_1$ is the highlighted part in the above expression and $\mathcal{S}_2 = \mathop{L}_{\vec{x}}\sum_{\vec{s}_1}\mathcal{S}_1[\vec{x},\vec{s}_1]$. Note that in summation splitting, EINNET converts the result of the inner summation into a scope, whose output is reused by the outer summation.

To transform a 3×3 convolution to a batch of nine matrix multiplications, as shown in Figure 5, EINNET first transforms the initial expression $\mathcal{E}_1$ to $\mathcal{E}_2$ by splitting the summation $\sum_{crs}$ into two summations $\sum_{rs}$ and $\sum_{c}$, and instantiating the output of the inner summation (i.e., $\{\mathop{L}_{rshwf}\sum_c A[h+r,w+s,c]K[r,s,f,c]\}$). The inner scope only sums along the c dimension; as a result, an intermediate five-dimensional tensor is instantiated since the summation along the r and s dimensions is not performed but converted to traversal notations. The outer scope computes the remaining summation over the r and s dimensions, which produces a three-dimensional tensor. Figure 5 (a) and (b) show the change in computation graph.

Variable substitution substitutes a set of traversal iterators $\mathop{L}_{\vec{x}}$ with a new set of iterators $\mathop{L}_{\vec{y}}$ by applying a *bijective* function Φ (i.e., $\vec{y} = \Phi(\vec{x})$). This transformation allows the expression to be computed using a different set of traversal iterators. In particular, for an expression $\mathop{L}_{\vec{x}}^{\mathbb{X}} f(\mathbf{T}[\tau(\vec{x})])$, variable substitution introduces an intermediate scope that computes $\mathop{L}_{\vec{y}}^{\mathbb{Y}} f(\mathbf{T}[\tau(\Phi^{-1}(\vec{y}))])$, where Φ is a bijective function that maps the iterating space $\mathbb{X}$ to $\Phi(\mathbb{X})$, and Φ^{-1} is the reverse function of Φ:

$$\mathop{L}_{\vec{x}}^{\mathbb{X}} f(\mathbf{T}[\tau(\vec{x})]) \Rightarrow \mathop{L}_{\vec{x}}^{\mathbb{X}}\{\mathop{L}_{\vec{y}}^{\Phi(\mathbb{X})} f(\mathbf{T}[\tau(\Phi^{-1}(\vec{y}))])\}[\Phi(\vec{x})].$$

A variable substitution constructs an intermediate scope with new traversal iterators. To preserve functional equivalence, the original iterator $\vec{x}$ is used to construct the final result using the output of the intermediate scope.

Although numerous possible variable substitutions exist for an expression, EINNET infers legal ones by analyzing indexing functions in expressions and checking whether they can form bijections. In Figure 5, EINNET applies a variable substitution to transform the expression from $\mathcal{E}_2$ to $\mathcal{E}_3$ using a bijective function Φ that maps $(r,s,f,h+r,w+s)$ to (r,s,f,t_1,t_2). Specifically, $h+r$ and $w+s$ are substituted with t_1 and t_2 in $\mathcal{E}_3$. To automatically identify promising variable substitutions among all alternatives, §6.1 introduces expression distance, a novel technique for efficiently exploring the search space.

Traversal merging combines the traversal notations in two separate scopes into one scope using an indexing function Φ:

$$\mathop{L}_{\vec{x}}^{\mathbb{X}}\sum_{\vec{y}}^{\mathbb{Y}}\{\mathop{L}_{\vec{z}}^{\mathbb{Z}} f(\mathbf{T}[\tau(\vec{z})])\}[\Phi(\vec{x},\vec{y})] \Rightarrow \mathop{L}_{\vec{x}}^{\mathbb{X}}\sum_{\vec{y}}^{\mathbb{Y}} f(\mathbf{T}[\tau(\Phi(\vec{x},\vec{y}))])$$

where indexing function Φ maps the outer scope iterators $\vec{x},\vec{y}$ to the inner scope iterators $\vec{z}$ and satisfies $\Phi(\mathbb{X}\times\mathbb{Y}) \subseteq \mathbb{Z}$.

In the example of Figure 5, EINNET applies traversal merging to transform $\mathcal{E}_4$ to $\mathcal{E}_5$. For this transformation, the outer traversal and summation notations and the inner traversal notation both include five iterators (i.e., $\vec{x} = (h,w,f)$, $\vec{y} = (r,w)$, and $\vec{z} = (r,s,h,w,f)$). Traversal merging is applied with an identity mapping function Φ and an indexing function $\tau(r,s,h,w,f) = (r,s,f,h+r,w+s)$. Traversal merging removes a scope and preserves the same computation graph.

Boundary relaxing and tightening. Boundary tightening inspects whether the computation for some output elements can be avoided if these elements are constants for arbitrary

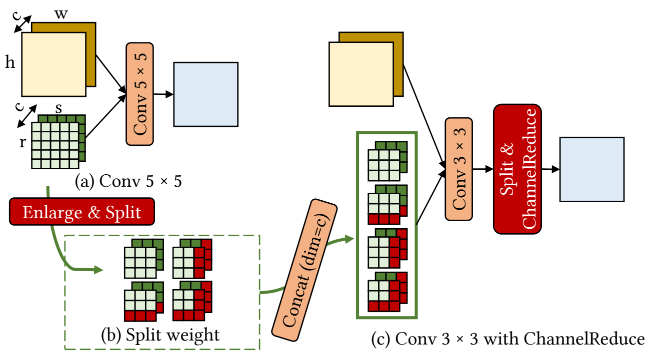


Figure 6: `Conv5×5` to `Conv3×3` transformation

inputs. EINNET executes constant propagation on expressions to deal with constant numbers in expressions and paddings in tensors. If an output region has constant values, EINNET converts it into an attribute of tensors to avoid unnecessary computation. In contrast, boundary relaxing enlarges tensors by adding extra paddings and redundant computations to explore more optimizations. Figure 6 shows the optimization that pads a `Conv5×5` to a `Conv6×6` and then converts it to a `Conv3×3` with quadrupled output channels. The following formula shows how relaxing and tightening are performed:

$$\mathop{\mathrm{L}}_{\vec{x}}^{\mathbb{X}} f(\mathbf{T}[\tau(\vec{x})]) \Longleftrightarrow \mathop{\mathrm{L}}_{\vec{x}}^{\mathbb{X}'} f(\mathbf{T}[\tau(\vec{x})]),$$

where $\mathbb{X} \subset \mathbb{X}'$, and $\mathbf{T}$ has a constant value in $\mathbb{X}' \setminus \mathbb{X}$.

To limit the number of possible candidate parameters for this rule, EINNET relaxes and tightens boundaries to a common constant. In the running example in Figure 5, the formula in $\mathcal{E}_4$ performs boundary relaxing on t_1 and t_2, transforming their ranges from $[r, H+r)$ and $[s, W+s)$ to $[-1, H+1)$ and $[-1, W+1)$, respectively, as the ranges of r and s are $[-1, 1]$ for a 3×3 convolution kernel. After boundary relaxing, the computation graph is transformed from Figure 5 (b) to (c). If multiple plans exist, the most strict one is selected to prevent extra redundant computing. Meanwhile, EINNET is still able to find the transformations introducing more redundancy by applying the rule multiple times.

EINNET performs boundary tightening to transform $\mathcal{E}_5$ into $\mathcal{E}_6$. In $\mathcal{E}_5$, as the computation performed on $t_1 = -1$, $t_1 = H$, $t_2 = -1$ and $t_2 = W$ falls in the paddings of tensor A, the computation result is zero as well. Hence, the ranges of t_1 and t_2 are tightened from $[-1, H+1)$ and $[-1, W+1)$ to $[0, H)$ and $[0, W)$, respectively. After boundary tightening, the computation graph is transformed from Figure 5 (c) to (d).

Derivation search space. The derivation rules allow EINNET to explore a large search space of expressions. Figure 7 illustrates the derivation search space of a 3x3 convolution. By applying different derivation rules, the initial expression is derived into various equivalent expressions, shown as the computation graphs in Figure 7. The motivating example shown in Figure 5 is identified by the derivation path $(a) \to (b) \to (c) \to (d) \to (e)$. The figure also shows many other expressions discovered by EINNET: By deriving the expression in (d) to `Conv1x1` instead of `Matmul`, EINNET discovers a new expression in (f). By merging summation iterators, expression (i) adopts an eOperator to concatenate multiple inputs with offsets for the following `Matmul`, which represents the conventional Im2col optimization [36]. Expression (k) shows a group convolution is equivalent to the original one by duplicating its input. Expressions (n) and (p) show two additional candidate expressions, both of which decompose the 3×3 convolution into smaller convolutions while preserving output using derived eOperators.

4.2 Inter-Expression Derivation

We now introduce the inter-expression derivations rule in EINNET for splitting, merging, and fusing expressions.

Expression splitting divides an expression into two *independent* ones by partitioning the original expression's traversal space $\mathbb{X}$ into two subspaces $\mathbb{X}_1$ and $\mathbb{X}_2$, where $\mathbb{X} \subseteq \mathbb{X}_1 \cup \mathbb{X}_2$:

$$\mathop{\mathrm{L}}_{\vec{x}}^{\mathbb{X}} f(\mathbf{T}[\tau(\vec{x})]) \Longrightarrow \mathop{\mathrm{L}}_{\vec{x}}^{\mathbb{X}_1} f(\mathbf{T}[\tau(\vec{x})]) \sim \mathop{\mathrm{L}}_{\vec{x}}^{\mathbb{X}_2} f(\mathbf{T}[\tau(\vec{x})])$$

where $\sim$ denotes the independence of the two expressions.

Expression merging is the reverse of expression splitting. It merges two *independent* expressions with the same computation by merging their traversal spaces $\mathbb{X}_1 \cup \mathbb{X}_2 \subseteq \mathbb{X}$:

$$\mathop{\mathrm{L}}_{\vec{x}}^{\mathbb{X}_1} f(\mathbf{T}[\tau(\vec{x})]) \sim \mathop{\mathrm{L}}_{\vec{x}}^{\mathbb{X}_2} f(\mathbf{T}[\tau(\vec{x})]) \Longrightarrow \mathop{\mathrm{L}}_{\vec{x}}^{\mathbb{X}} f(\mathbf{T}[\tau(\vec{x})])$$

Expression fusion fuses multiple *dependent* expressions into one using the following rule:

$$\mathop{\mathrm{L}}_{\vec{y}}^{\mathbb{Y}} g(\mathbf{T}'[\pi(\vec{y})]) \circ \mathop{\mathrm{L}}_{\vec{x}}^{\mathbb{X}} f(\mathbf{T}[\tau(\vec{x})]) \Longrightarrow \mathop{\mathrm{L}}_{\vec{y}}^{\mathbb{Y}} g(\{\mathop{\mathrm{L}}_{\vec{x}}^{\mathbb{X}} f(\mathbf{T}[\tau(\vec{x})])\}[\pi(\vec{y})])$$

where $\mathbf{T}'$ is equal to the computation result of the highlighted part in the above expression, and $\mathcal{E}_1 \circ \mathcal{E}_2$ denotes that the result of expression $\mathcal{E}_2$ is fed as inputs to expression $\mathcal{E}_1$.

Figure 3(a) shows a sequence of derivations involving inter-expression derivation. EINNET first applies intra-expression derivation rules to transform `Conv3x3` and `Conv1x1` to two `Matmuls` and an eOperator. Since the two `Matmuls` share the same input and computation pattern, EINNET is able to apply the expression merging rule upon them. As shown in the dashed box, EINNET transposes and concatenates the two weight tensors as the input for `Matmul`. The outputs of `Matmul` are split to get two equivalent outputs. Furthermore, EINNET applies the expression fusion rule to perform vertical operator fusion, an optimization fusing a chain of operators into a single kernel to reduce data movement and kernel launch overhead. In the solid boxes in Figure 3(a), EINNET fuses memory-bound operators (e.g., `OffsetReduce` and `Relu`) into one eOperator by applying expression fusion.

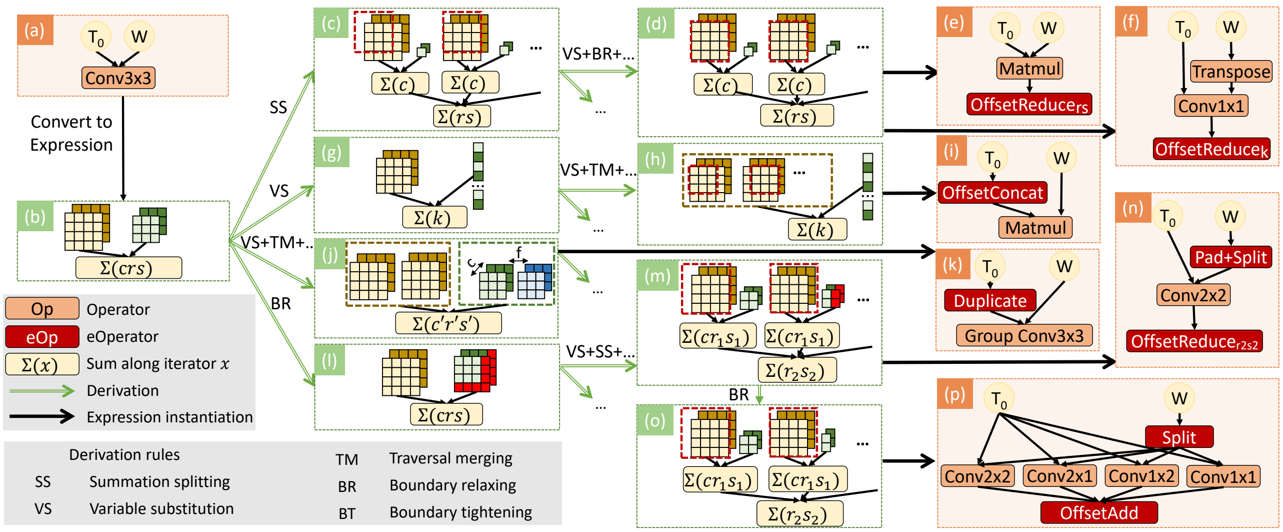


Figure 7: Derivation procedure for a subgraph of a convolution. Data layout transformation operators and intermediate derivation steps are omitted for conciseness. The output channel dimension of convolution kernel is only shown in (j) and denoted by f.

5 Expression Instantiation

Although EINNET can treat all expressions as eOperators and use an off-the-shelf kernel generator (e.g., TVM in our implementation) to generate executable programs, doing so would result in suboptimal performance. This is because existing vendor-provided tensor libraries such as cuDNN [10] and cuBLAS [11] include a collection of highly optimized tensor algebra kernels that outperform their counterparts generated by tensor compilers. The performance and expressiveness trade-off between hand-tuned and auto-generated kernels introduces both challenges and opportunities: we should opportunistically lower some expressions to vendor-provided kernels to realize their performance advantages and use kernel generators to generate executable programs for remaining expressions. We refer to this task as *expression instantiation*.

EINNET considers two derivation rules for expression instantiation: (1) *operator matching* allows EINNET to opportunistically use existing highly optimized kernels (e.g., cuDNN [10] and cuBLAS [11]) to achieve high performance, and (2) *eOperator generation* enables flexible kernel generation for an arbitrary eOperator. After applying these rules, the instantiated scopes are replaced with tensors in the original expression and are separated from the following derivation.

To lower expressions to kernels, EINNET uses a strategy that maps compute-intensive expressions to predefined operators and employs a kernel generator for memory-bound expressions. This strategy allows EINNET to benefit from existing vendor libraries and maintain low compilation time, since memory-bound expressions usually involve a small schedule space in existing code generation frameworks [7]. While a more aggressive utilization of kernel generators has the potential to outperform the opportunistic strategy, it introduces significant kernel tuning overhead for millions of possible expressions during the program optimization. This is due to the difficulty in accurately estimating the performance of a kernel without actually tuning and profiling it.

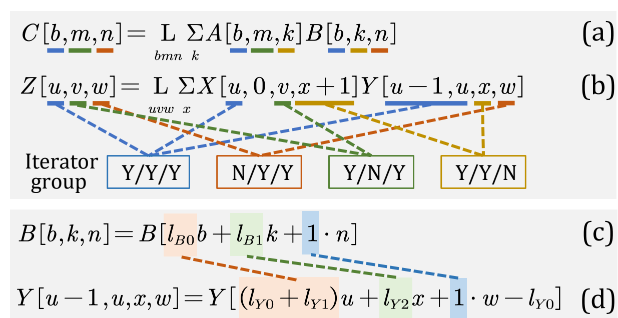


Figure 8: Match an expression to `BatchMatmul`. Expression (a) and (b) show iterator groups and Expression (c) and (d) show matching attributes with flattened expressions.

To determine whether an expression is compute-intensive or memory-bound, EINNET analyzes its arithmetic intensity, calculated as the ratio between its FLOPs and tensor sizes. Expressions with arithmetic intensity lower than a threshold (4 in our evaluation) are considered memory-bound eOperators. EINNET decides whether to perform operator matching or eOperator generation for this expression based on this metric. The following introduces these two instantiation rules.

5.1 Operator Matching

Mapping an expression to a predefined operator is challenging since an operator can be represented in various expressions. For example, while expressions in Figure 8(a-b) have distinct forms, they can both be instantiated as batched matrix multiplication kernels in cuBLAS as it supports tensors with flexible data layouts.

Table 2: Iterator mapping table. Iterators are categorized by where they appear in expressions. For each iterator group, Y and N indicate whether the iterators appear in the index of corresponding tensors.

Operators	Tensor algebra expressions	Iterator groups ($\mathbf{I_0}$/$\mathbf{I_1}$/$\mathbf{O_0}$)			
		Y/Y/Y	Y/N/Y	N/Y/Y	Y/Y/N
`Add`	$\mathbf{O_0}[m,n] = \mathbb{L}_{mn}\mathbf{I_0}[m,n] + \mathbf{I_1}[m,n]$	m,n			
`BatchMatmul`	$\mathbf{O_0}[b,m,n] = \mathbb{L}_{bmn}\sum_k \mathbf{I_0}[b,m,k]\mathbf{I_1}[b,k,n]$	b	m	n	k
`Conv`	$\mathbf{O_0}[n,h,w,f] = \mathbb{L}_{nhwf}\sum_{crs}\mathbf{I_0}[n,h+r,w+s,c]\mathbf{I_1}[r,s,f,c]$		n,h,w	f	c,r,s
`G2BMM`	$\mathbf{O_0}[b,m,w] = \mathbb{L}_{bmw}\sum_k \mathbf{I_0}[b,m,k]\mathbf{I_1}[b,k,m+D*(w-W)]$	b,m		w	k

EINNET uses an *iterator mapping table* to determine if a given expression can be mapped to a predefined operator, where iterators of each operator are grouped based on whether the iterator appears in the operator's input/output tensors. Table 2 shows the iterator mapping table for several operators with two input and one output tensors, including element-wise operators, batched matrix multiplication, convolution, and G2BMM [23] (general to band matrix multiplication). Each row in the table corresponds to an operator, while each column shows an *iterator group*. The iterator mapping table also records the coefficients of iterators in the index of each tensor for operator matching. It can be extended to support operators with an arbitrary number of inputs and outputs.

The iterator mapping table allows EINNET to determine if an expression can be mapped to an operator as follows:

1. **Match tensors**. To map a given expression to an operator, EINNET enumerates all possible one-to-one mappings between the expression and operator's input/output tensors. For example, to map the expression in Figure 8 (b) to `BMM` (i.e., expression in Figure 8 (a)), there exist two possible mappings, $\{A \to X, B \to Y\}$ and $\{A \to Y, B \to X\}$).
2. **Match iterators.** For each tensor mapping, EINNET further enumerates all possible ways to match iterators between the expression and operator using the iterator mapping table described above. For example, assuming a tensor mapping $\{A \to X, B \to Y\}$ in Figure 8, iterators $\{u,v,x,w\}$ in (b) are mapped to iterators $\{b,m,k,n\}$ in (a) based on the iterator mapping table (iterators in the same group are marked in the same color). When there are multiple iterators in the same group, EINNET enumerates all possible mappings between these iterators.
3. **Match operator attributes.** Many predefined operators contain attributes to specify computation. E.g., modern BLAS libraries use *lda* and *ldb* to control the data layouts for input tensors in matrix multiplication. To match these attributes, EINNET flattens the input and output tensors (i.e., reshapes them into one-dimensional tensors) to hide the complexity of tensor shapes. EINNET then matches the operator attributes by examining the variable coefficients of the flattened tensors. Figure 8(c-d) show how EINNET determines the attributes l_{B0} and l_{B1} for a `BMM` operator. It flattens the tensor B in both expressions and compares their coefficients: $l_{B0} = l_{Y0} + l_{Y1}$, $l_{B1} = l_{Y2}$, where l_{Yn} is the stride of n-dimension of tensor Y. The coefficient of w in Expression (d) is also checked to be equal to that of n in Expression (c), as they are a pair of matched iterators.

```
T1 = tvm.te.compute((H, W, F), lambda h, w, f:
       tvm.te.sum(T1[r, s, h+r, w+s, f], axis=[r, s]))
```

Figure 9: Lowering $\mathcal{E}_7$ in Figure 5 to TVM.

5.2 eOperator Generation

For expressions that cannot be mapped to vendor-provided predefined operators, EINNET converts them into eOperators. Since an eOperator precisely defines its computation, EINNET can directly feed it to an off-the-shelf kernel generation framework (e.g., TVM [7]). For example, for expression $\mathcal{E}_7$ in Figure 5, which corresponds to `OffsetReduce` in the transformed computation graph, EINNET feeds it to TVM by converting its iterator space into a tensor and the computation expression into a lambda function. Figure 9 shows the TVM code generated by EINNET for expression $\mathcal{E}_7$, which can be an input program to TVM to generate an executable kernel.

6 Program Optimizer

This section describes EINNET's *program optimizer*, which uses the expression derivation and instantiation techniques described in §4 and §5 to optimize input tensor programs. These derivation rules create a large and complex search space of programs functionally equivalent to the input. EINNET uses a *distance-guided* search algorithm to explore the space (§6.1) and develops a *fingerprinting* technique to prune redundancy (§6.2). Finally, §6.3 describes how EINNET orchestrates these techniques to perform end-to-end optimizations.

6.1 Distance-Guided Search

To explore the search space created by EINNET's derivations, a purely randomized search strategy can only explore either a limited set of paths or small searching depths, leading to suboptimal performance. To address this challenge, EINNET develops a two-stage *distance-guided* search algorithm to apply derivations. It includes an *explorative derivation* stage and a *converging derivation* stage, as shown in Figure 10.

Explorative derivation. During explorative derivation, EINNET iteratively applies *all* derivation rules to the current expression. A hyperparameter *MaxDepth* determines

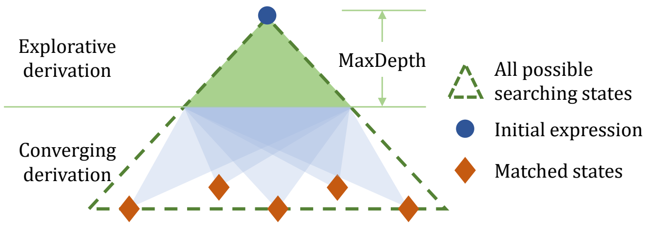


Figure 10: Distance-guided search

the maximum number of derivation rules EINNET applies during explorative derivation. As described in §5, EINNET opportunistically uses vendor-provided kernel libraries to maximize performance. Thus, EINNET leverages converging derivation to quickly derive an expression toward a target operator (e.g., operators in cuDNN and cuBLAS). EINNET automatically generates necessary eOperators to bridge the gap between the current expression and target operator.

Converging derivation. During converging derivation, EINNET first selects a target operator and uses a novel metric, *expression distance*, to guide the applications of derivation rules in this stage. Expression distance measures the difference between a given expression $\mathcal{E}_1$ and the canonical expression of a given operator $\mathcal{E}_2$. To calculate the distance between $\mathcal{E}_1$ and $\mathcal{E}_2$, EINNET first matches all iterators in $\mathcal{E}_1$ and $\mathcal{E}_2$ using the iterator mapping table (see §5.1) and counts the total number of mismatched iterators as their distance.

Specifically, each iterator mismatch between the current expression and target operator indicates that the two expressions have a different number of iterators in an iterator group (see Table 2). EINNET applies derivation rules to fix mismatches, such as variable substitution rules to merge/split iterators, resulting in reduced expression distances. For example, to derive the expression in the inner scope of $\mathcal{E}_6$ in Figure 5 to a `Matmul`, EINNET compares their iterators (Table 2) and obtains the following matches: $t_1, t_2 \rightarrow m; r, s, f \rightarrow n; c \rightarrow k$. To fix mismatches, EINNET applies variable substitutions to merge iterators t_1 and t_2 into m and merge r, s, f into n.

After all iterators are matched, EINNET infers the shape of each input/output tensor according to the target operator and constructs new tensors from existing ones by adding eOperators. For example, the new input tensor $\mathbf{A}'$ and weight tensor $\mathbf{K}'$ for `Matmul` are constructed by the following expressions:

$$\mathbf{A}'[m,k] = \mathbf{A}'[t_1 \times W + t_2, c] = \mathbf{A}[t_1, t_2, c] \quad (2)$$

$$\mathbf{K}'[k,n] = \mathbf{K}'[c, r \times S \times F + s \times F + f] = \mathbf{K}[r,s,f,c], \quad (3)$$

where the mapping functions are $(m,k) = \Phi_A(t_1,t_2,c) = (t_1 \times W + t_2, c)$ and $(k,n) = \Phi_K(r,s,f,c) = (c, r \times S \times F + s \times F + f)$, and W, S, and F are the range of the iterators w, s and f. EINNET automatically generates Expression (2) and (3) to fix the mismatch and reduce the expression distance.

During converging derivation, EINNET only considers derivations that reduce the expression distance of the current expression and target operator, allowing EINNET to prune most derivations and quickly converge to the target operator. By enumerating operators in the iterator mapping table as the target operator, EINNET finds transformations involving different operators.

Delayed code generation. To accelerate the search, EINNET estimates the performance of derived programs to avoid frequent code generation for eOperators. Specifically, the execution time of a predefined operator is measured by profiling its kernel on hardware. Meanwhile, the run time of an eOperator is estimated based on its input/output sizes and hardware memory bandwidth. We observe that this estimation is accurate since eOperators are memory-bound and usually account for a small part of the total execution time.

6.2 Redundancy Pruning

Applying different sequences of derivations may result in the same expression. For example, splitting an iterator into two and then merging them results in the original one. To prune redundancy, EINNET uses a *fingerprint* technique to detect duplicate expressions. A *fingerprint* is a hash of an expression and can eliminate the following sources of redundancy:

- **Summation reordering**: summations can be reordered, e.g., $\sum_{\vec{x}} \sum_{\vec{y}} f(\vec{x},\vec{y})$ is equivalent with $\sum_{\vec{y}} \sum_{\vec{x}} f(\vec{x},\vec{y})$. Note that traversal reordering does not imply equivalence since it involves layout transformations.
- **Operand reordering**: operands of commutative binary operations can be reordered, e.g., $\mathrm{L}_{\vec{x}}(\mathbf{T_1}[\vec{x}] + \mathbf{T_2}[\vec{x}])$ is equal to $\mathrm{L}_{\vec{x}}(\mathbf{T_2}[\vec{x}] + \mathbf{T_1}[\vec{x}])$. Operand reordering should be applied for both iterator computation and tensor computation.
- **Iterator renaming**: iterators should be distinguished by their iterator space instead of names, e.g., $\mathrm{L}_{x=0}^{N} \mathrm{L}_{y=0}^{M} f(x,y)$ and $\mathrm{L}_{y=0}^{N} \mathrm{L}_{z=0}^{M} f(y,z)$ are equivalent, and (x,y) in the former one should be mapped to (y,z) in the latter one.
- **Tensor renaming**: tensors introduced by different scopes may have the same value.

To eliminate the above sources of redundancy, EINNET adopts the following methods to calculate fingerprints. For a traversal iterator, EINNET uses its iterator space and its order relative to all other traversal notations in the current scope as its fingerprint. Since order is considered, fingerprint can differentiate traversal iterators with the same iterator spaces but in different locations of the traversal notations. For a summation iterator, EINNET only uses its iterator space as its fingerprint. Thus expressions under summation reordering have the same fingerprint. To account for operand reordering, EINNET uses the operation type and an *order-independent* hash for commutative operations (e.g., addition) and an *order-dependent* hash for other operations. The fingerprint of a tensor depends on its source. For an input tensor, EINNET calculates its fingerprint by hashing its name. For an intermediate tensor generated by a scope, its fingerprint is identical to that of the expression that produces the tensor.

Algorithm 1 Program-level optimizer.

1: **Input:** An input tensor program $\mathcal{P}$
2: **Output:** An optimized tensor program $\mathcal{P}_{\text{opt}}$
3:
4: $\mathcal{IR}$ = inter-expression rule set
5: $\mathcal{SP}$ = split $\mathcal{P}$ and translate subprograms into expressions
6: $\mathcal{P}_{\text{opt}} = \varnothing$
7: **for** $E_0 \in \mathcal{SP}$ **do**
8: $\quad \mathcal{Q} = \{E_0\}$
9: $\quad$ **for** $E \in \mathcal{Q}$ **do**
10: $\quad\quad$ **for** $r \in \mathcal{IR}$ **do**
11: $\quad\quad\quad \mathcal{Q} = \mathcal{Q} + r(\mathcal{E})$
12: $\quad\quad \mathcal{Q} = \mathcal{Q}$ + DISTANCEGUIDEDSEARCH(E)
13: $\quad$ Add the best transformation in $\mathcal{Q}$ into $\mathcal{P}_{\text{opt}}$
14: POSTOPTIMIZATION($\mathcal{P}_{\text{opt}}$)
15: **return** $\mathcal{P}_{\text{opt}}$

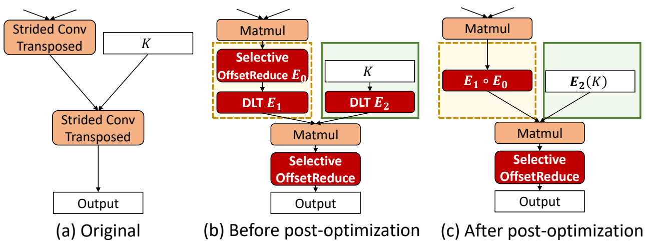


Figure 11: Post-optimization for InfoGAN. Red blocks represent eOperators. DLT means data layout transformation.

6.3 End-to-End Workflow

Algorithm 1 shows EINNET's workflow for optimizing an input tensor program in an end-to-end fashion. For an input program, EINNET first splits it into multiple subprograms using non-linear activation operators as the splitting points. This is because activation operators often do not provide further optimization opportunities other than fusion, as discovered by prior work [38]. For each subprogram, EINNET translates it into expressions using the canonical expression of each operator. Since a subprogram may include multiple operators and thus multiple expressions, EINNET applies inter-expression derivation rules (Line 11) and feeds each expression to the distance-guided search (§6.1) for performing *intra-expression* derivations (Line 12). Instead of integrating intra- and inter-expression optimizations in a unified search space and performing them jointly, the separate search prioritizes the transformations that can map expressions into operators. Thus, EINNET is able to find promising transformations quickly and prune unnecessary search states according to the execution time of transformed results.

Finally, EINNET selects the best-discovered expression of each subprogram, performs post-optimization, and generates an optimized tensor program. Figure 11 shows two types of post-optimization: eOperator fusion and compile-time expression evaluation. EINNET generates eOperators to facilitate optimizing transformations when optimizing a subprogram. During post-optimization, consecutive eOperators are fused into a single eOperator by applying inter-expression derivations. The dashed boxes in Figure 11(b) and (c) show such cases. EINNET also detects compile-time computable expressions to reduce runtime overhead. For example, the data layout transformation E_2 in Figure 11 can be processed during post-optimization.

7 Evaluation

7.1 Experimental Setup

Implementation of EINNET. EINNET is built with over 23K lines of C++ and Python code. We realize the tensor expression derivation system from scratch and implement an execution runtime for tensor programs. Users can both define tensor programs in EINNET directly and load existing ones in the ONNX format [29]. To support an operator in derivation, EINNET requires its tensor expression and operator attribute constraints to automatically convert it between expressions and operators. We set the default maximum search depth of explorative derivation to 7, which is an empirical configuration satisfying both optimization quality and search time.

Platform. We evaluate EINNET on a server with dual Intel Xeon E5-2680 v4 CPUs, NVIDIA A100 40GB and V100 32GB PCIe GPUs. All experiments use CUDA 11.0.2, cuBLAS 11.1.0, and cuDNN 8.0.3.

Workloads. We evaluate EINNET on seven DNN models, spanning various fields and covering both classic and emerging DNNs. InfoGAN [9] is a generative adversarial network learning disentangled representations from objects. DCGAN [32] leverages deep convolution structures to get hierarchical representations. FSRCNN [13] is used for fast image super-resolution. GCN [30] is an image semantic segmentation model. ResNet-18 [19] is a famous image classification network. CSRNet [25] adopts dilated convolution for congested scene analysis. LongFormer [6] is an improved Transformer model dealing with long-sequence language processing. We adopt typical input shapes based on the papers and implementations of models to keep the evaluation meaningful in real scenarios.

7.2 End-to-End Performance

We first compare the end-to-end inference performance of EINNET against today's DNN frameworks, including TensorFlow v2.4 [4], TensorFlow XLA [2], Nimble [22], TVM v0.10 with Ansor [7], TensorRT v8.2 [35], and PET v0.1 [38]. All frameworks use the same version of cuBLAS and cuDNN as their backend and the same data type FP32 in computation for a fair comparison. For the new attention operator in Longformer, we provide an auto-tuned kernel for TVM, TensorRT, PET, and EINNET, and implement it by

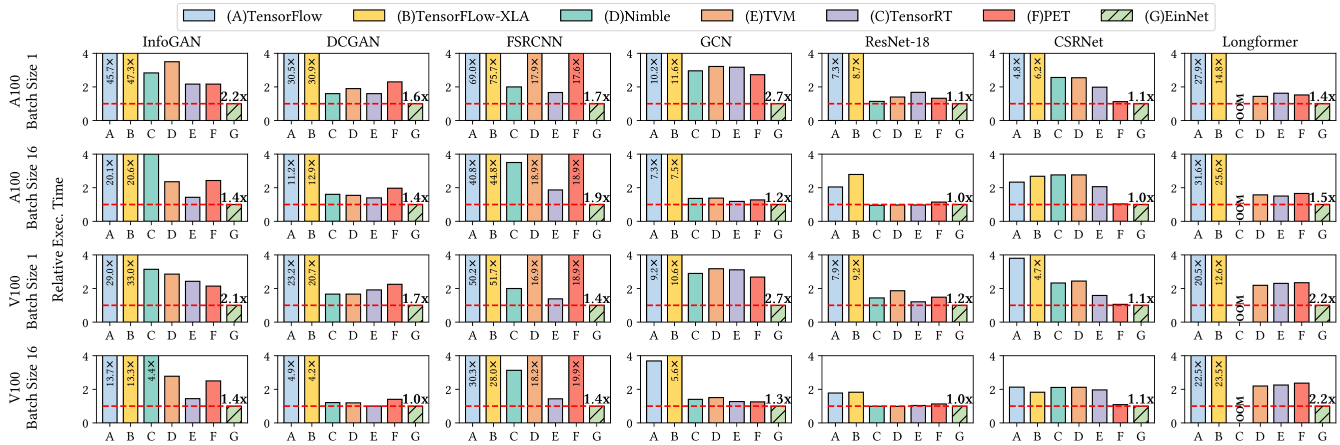


Figure 12: End-to-end performance comparison with other systems on an A100 and a V100 GPU with batch sizes of 1 and 16. OOM means out of memory. Bars over 4× are truncated, and their relative execution times to EINNET are marked on the bars. The numbers above EINNET's bars show EINNET's speedups over the best baseline.

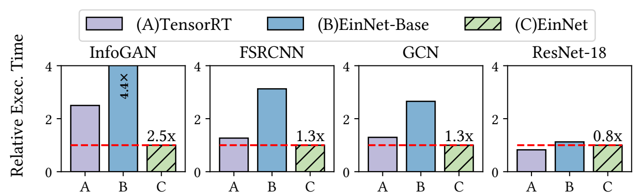


Figure 13: End-to-end performance comparison with TensorRT on an A100 with TF32 and batch sizes of 1. The numbers above EINNET's bars show EINNET's speedups over the best baseline.

einsum in other frameworks. Figure 12 shows the results on NVIDIA A100 and V100 GPUs under batch sizes 1 and 16.

EINNET outperforms the best existing baseline by up to 2.72× on A100 and 2.68× on V100. For both CNNs (e.g., GCN) and language models (e.g., Longformer), EINNET is able to improve their performance by more than 2×. Among the seven models, ResNet-18 has been heavily optimized by existing tensor program frameworks and optimizers; however, EINNET still outperforms existing optimizers by 1.2× on V100, by applying the novel transformations shown in Figure 3. For CSRNet, a typical optimization case of PET, EINNET discovers similar transformations by derivations and eliminates extra introduced transposes, indicating that EINNET's derivation rules can perform PET's optimizations and uncover additional improvements.

Figure 13 shows the speedup with the computation data type of TF32 and Tensor Cores on A100. To show the benefits provided by EINNET, we create a baseline EINNET-Base which executes models in EINNET with derivation optimizations disabled. As shown in Figure 13, while EINNET usually brings significant speedups over EINNET-Base and TensorRT, TensorRT can have better performance in models like ResNet-18. Though TensorRT is not open source, the profiling results

Table 3: Performance studies on the cases in §7.3. The Algo column shows the best convolution algorithm for cuDNN, where `IGEMM`, `FFT`, and `WINO` refer to implicit GEMM, Fast Fourier Transform, and Winograd [24] algorithms. The DRAM and L2 columns show the amount of memory access.

	Input shape		Conv Algo	Time (ms)	DRAM (MB)	L2 (MB)
`Conv3x3` Figure 3 (b)	[1,512,7,7]	**Original**	`WINO`	0.126	56.7	70.6
		Optimized	`N/A`	0.046	10.5	27.5
`Conv-Transpose`	[16,256,2,2]	**Original**	`IGEMM`	0.136	7.74	122
		Optimized	`N/A`	0.032	8.07	27.8
`Conv5x5` Figure 6	[16,32,224,224]	**Original**	`FFT`	0.854	547	579
		Optimized	`WINO`	0.528	368	499
`G2BMM` Figure 14	[8,10000,64]	**Original**	`N/A`	7.14	20.9	19750
		Optimized	`N/A`	1.57	20.6	817

show that it leverages many efficient GPU kernels besides cuBLAS and cuDNN. This can be an important source of its high performance, which is beyond the current search space of EINNET.

7.3 Optimization Analysis

This section analyzes the optimizations discovered by EINNET on these DNNs. To highlight transformations beyond the scope of existing tensor program optimizers, we focus on transformations involving eOperators.

Transforming operator types. EINNET is able to opportunistically substitute an inefficient operator with well-optimized operators of different types. In ResNet-18 and InfoGAN, the transformations from `Conv` and `ConvTranspose` to `Matmul` are profitable. Table 3 shows a detailed performance analysis. As shown in Figure 3(b), EINNET transforms a `Conv3x3` to a `Matmul` and an eOperator (`OffsetReduce`), which significantly reduces GPU DRAM accesses from 56.7 MB to 10.5 MB and achieves a 2.7× speedup. As another

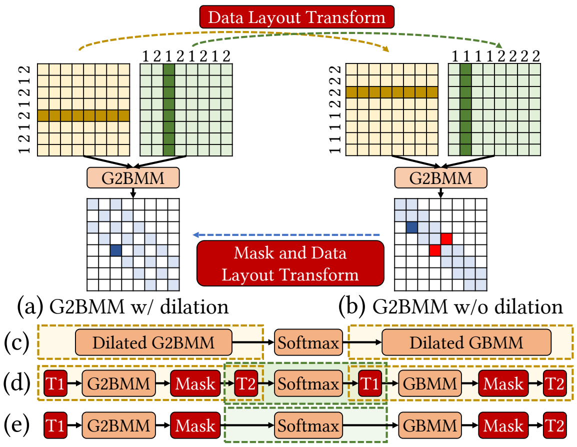


Figure 14: Optimization for Longformer Attention block. `T1` and `T2` are two reciprocal data layout transformations.

example, EINNET also derives a strided `ConvTranspose` to a `Matmul` and another eOperator that selectively aggregates the output of `Matmul` according to the derived expression. This transformation significantly reduces L2 access, a key contribution to performance optimization.

Transforming operator attributes. EINNET can also transform operator attributes by leveraging eOperators. Figure 6 shows such an optimization for convolution, which converts its kernel size from 5×5 to 3×3, allowing EINNET to use more advanced convolution algorithms best suited for 3×3 convolutions. To realize this transformation, an eOperator is added to split the output of `Conv3x3` across the channel dimension and reduce the intermediate results with corresponding offsets. Although padding the convolution kernel introduces additional computation, Table 3 shows it enables using the Winograd algorithm for convolution, which further reduces compute time and memory access.

Transforming tensor layouts. eOperators allow EINNET to accelerate DNN computation by optimizing tensor layouts. Figure 14 shows such a layout optimization for Longformer, which uses a dilated `G2BMM` (general to band matrix multiplication) to compute sparse attention. `G2BMM` has the same computation pattern as `Matmul` and only computes a subset of output. The blue boxes in Figure 14(a) show the output locations with a dilation of 2. EINNET discovers an optimizing layout transformation that reorders the odd and even rows or columns, converting the dilated `G2BMM` to a non-dilated one, as shown in Figure 14(b), which greatly reduces non-contiguous memory accesses at the cost of introducing two redundant elements (marked as red in the figure). As shown in Table 3, this transformation can reduce L2 cache access by 95.9% and execution time by 78.0%.

Transforming graph structures. For the Longformer case shown in Figure 14(d), four data layout transformations are introduced to accelerate dilated `G2BMM`. While they are not predefined operators, EINNET finds that the middle two are reciprocal through expression fusion and eliminates them since they do not affect the `Softmax` computation in between.

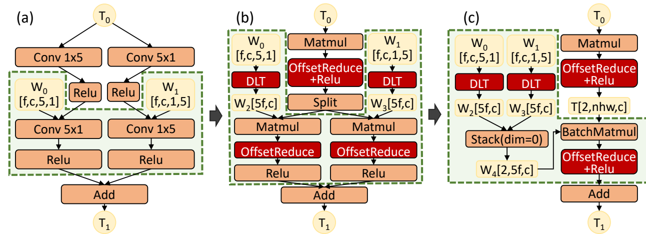


Figure 15: Optimization for the spatial separable convolutions in GCN. c and f are the numbers of input and output channels.

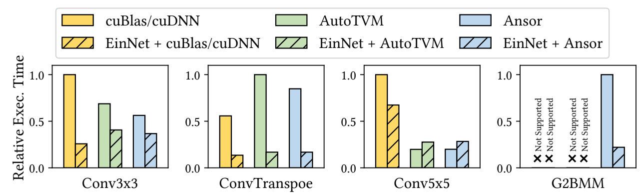


Figure 16: Operator performance before and after optimization (opt) on the math libraries and code generation framework Ansor. The input settings are shown in Table 3.

A more complex example is in GCN, which has a repeated structure of spatially separable convolutions (i.e., sequential `Conv1xK`s and `ConvKx1`s). As shown in Figure 15(b), EINNET first transforms convolutions to `Matmul`s and eOperators, and then merges the first two `Matmul`s into a single `Matmul`. While the two following `Matmul`s do not share common inputs, they have an identical computation pattern and can be merged into a `BatchMatmul` by applying the expression merging and operator matching rules. Note that the left part of Figure 15(c) is computed at compile-time by EINNET since it only involves weight tensors. These transformations optimize subgraph execution time by $4.9\times$ on A100 with batch size of one.

7.4 Integration with Different Backends

Since EINNET searches expression space, it can cooperate with different backends, including math libraries and schedule-based code generation frameworks. To show the improvements of EINNET on these backends, we evaluate EINNET with cuBLAS/cuDNN, AutoTVM [7], and TVM auto-scheduler Ansor [8]. The evaluation is carried out on the same transformations illustrated in §7.3.

Figure 16 shows EINNET can optimize tensor programs on different backends. For the `Conv3x3` in ResNet-18 and the `ConvTranspose` in InfoGAN, transforming them to `Matmul`s and eOperators has significant speedup over all three platforms. While the transformation from `Conv5x5` to `Conv3x3` is beneficial for cuDNN, it does not have perfor-

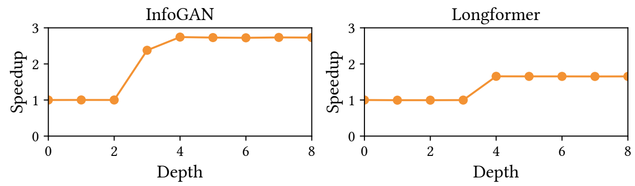


Figure 17: Speedup under different maximum search depths

mance improvement on AutoTVM and Ansor even if efficient algorithms such as Winograd [24] are adopted. This result shows while many transformations are effective on different backends, customizing transformations for each backend is beneficial. For the `G2BMM` operator in Longformer, which is not a predefined operator in current math libraries, the transformation on Ansor brings a speedup of 4.54× since less non-contiguous memory access happens.

7.5 Analysis of Automated Derivation

The search space of EINNET is determined by heuristic parameters, e.g., the maximum search depth for the distance-guided search algorithm (§6.1), which specifies the largest steps of derivation applied to an expression. A larger search depth enables more potential optimizations but introduces larger searching overhead. Figure 17 analyzes the speedup EINNET can achieve with different maximum search depths on InfoGAN and Longformer. On InfoGAN, EINNET has improvement when the searching step increases from 2 to 4, as new transformations are explored with a deeper search. While for Longformer, the major speedup comes from the transformation found in a 4-step derivation. In conclusion, the key takeaway is that EINNET can achieve most of the performance improvement at moderate depth.

To evaluate the proposed techniques for derivation, we evaluate the searching process on the four cases in Table 3 with and without converging derivation and expression fingerprint.

Distance-guided derivation (§6.1) provides a deterministic derivation direction to reduce search overhead. As shown in Figure 18(a), the search time grows exponentially with the maximum depth of explorative derivation (i.e., *MaxDepth* in Figure 10). EINNET adopts converging derivation to reduce the search depth of explorative derivation. Figure 18(b) shows the number of applied explorative derivations in these cases.

In the case of ConvTranspose, the explorative derivation requires a search with $MaxDepth = 12$ to discover the target expression. But with converging derivation, EINNET only requires a search with $MaxDepth = 6$, which means that matching a vendor-provided operator needs a six-step $(12-6)$ search and converging derivation can reduce this unnecessary search. Thus, this optimization leads to a significant reduction of the search time by more than 99.0%.

Expression fingerprint (§6.2) prunes redundant searching states. Figure 19 shows the intermediate states and search time with and without the fingerprint mechanism. During

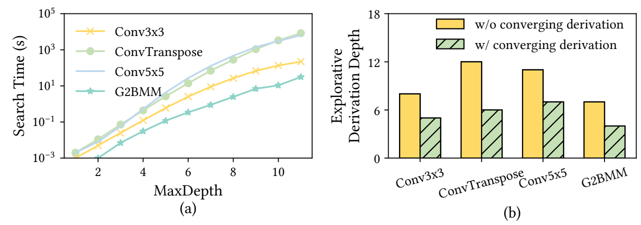


Figure 18: (a) Search time with different *MaxDepth*. (b) The number of explorative derivation steps with and without converging derivation.

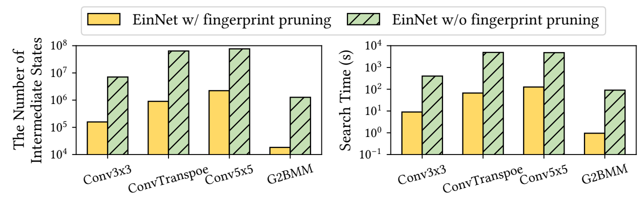


Figure 19: Ablation study of expression fingerprint pruning

the derivation, fingerprint effectively prunes 98.0% of intermediate states by recognizing and eliminating duplicate expressions and reduces 98.2% of search time on average.

With the distance-guided derivation and expression fingerprint, EINNET finishes searching within two minutes for most subprograms and is on par with existing frameworks like TASO and PET. The search spends no more than two hours for most models, which depends on the number of operators contained in models. EINNET is able to be deployed in real production environment since the search cost is one-off for a model and brings persistent benefits.

8 Related Work

Rule-based approaches such as TensorFlow XLA [2], TensorRT [35], and Grappler [1], are widely used and perform optimizations like constant folding and layout optimization. While they can efficiently optimize computation graphs, it requires extensive expert efforts and only performs manually discovered optimizations. For operator fusion, DNNFusion [28] adopts operator-level mathematical-property-based graph rewrite rules, such as associative and commutative properties. However, such rewriting rules are mainly designed for element-wise operators and cannot be easily extended to arbitrary operators since many complex operators, such as convolution and matrix multiplication, do not follow associative and commutative properties. EINNET derives tensor programs at expression level to exploit general program transformations, including splitting, fusing, and reorganizing computation into operators and eOperators. This avoids manually summarizing rules for each operator.

Superoptimization-based approaches. TASO [20] and

PET [38] use superoptimization techniques to generate graph transformations for a given set of operators. TASO adopts formal verification techniques to assure the equivalence of transformations. PET further introduces partially-equivalent transformations and correction mechanism to find more optimizations. Compared with these frameworks, EINNET extends the search space from POR transformations to general expressions by tensor algebra expression derivation.

Schedule-based approaches. Halide [24] decouples a program into computation and schedule and performs schedule space transformations. This idea is widely adopted by code generation frameworks, including TVM [7], FlexTensor [42], and Ansor [40]. Orthogonal to schedule-based optimizers, EINNET focuses on high-level graph transformation space and designs the architecture-independent expression derivation rules to reorganize computation into efficient operators.

Task-based approaches. Task, an abstraction of computation and memory operation workload, is introduced into tensor programs recently to optimize their performance. Rammer [27] divides operators into fine-grained tasks and proposes a sub-operator-level task scheduling method to exploit both intra- and inter-operator parallelism. Hidet [12] leverages task mapping in kernel generation to explore more aggressive optimizations. EINNET is compatible with these optimizers by utilizing them as execution and kernel generator backend.

Tensorization. TensorIR [14], AMOS [41], and Glenside [34] aim to generate tensorized code on tensor accelerators. While computation mapping is stressed in their workflows, these works focus on generating performant native code leveraging special circuits, such as fixed-shape matrix multipliers Intel AMX and NVIDIA TensorCore. In contrast, EINNET adopts expression matching to match arbitrary linear tensor algebra expressions, which are more flexible and contain undetermined parameters in the pattern expressions.

9 Conclusion

We propose EINNET, a derivation-based tensor program optimizer, which extends the optimization space of tensor programs from predefined operator representable transformations to general expressions and can create new operators desired by transformations. EINNET can outperform state-of-the-art frameworks by up to 2.72× on NVIDIA GPUs.

References

[1] Tensorflow graph optimization with grappler; tensorflow core.

[2] Xla: Optimizing compiler for tensorflow. https://www.tensorflow.org/xla, 2017.

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

[4] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[5] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International conference on machine learning*, pages 173–182. PMLR, 2016.

[6] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

[7] 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. TVM: an automated end-to-end optimizing compiler for deep learning. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 578–594. USENIX Association, 2018.

[8] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems 31*, NeurIPS'18. 2018.

[9] Xi Chen, Yan Duan, Rein Houthooft, John Schulman, Ilya Sutskever, and Pieter Abbeel. Infogan: Interpretable representation learning by information maximizing

generative adversarial nets. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, pages 2180–2188, 2016.

[10] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

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

[12] Yaoyao Ding, Cody Hao Yu, Bojian Zheng, Yizhi Liu, Yida Wang, and Gennady Pekhimenko. Hidet: Task-mapping programming paradigm for deep learning tensor programs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 370–384, 2023.

[13] Chao Dong, Chen Change Loy, and Xiaoou Tang. Accelerating the super-resolution convolutional neural network. In *European conference on computer vision*, pages 391–407. Springer, 2016.

[14] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, et al. Tensorir: An abstraction for automatic tensorized program optimization. *arXiv preprint arXiv:2207.04296*, 2022.

[15] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.

[16] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics*, 37(3):362–386, 2020.

[17] Anmol Gulati, James Qin, Chung-Cheng Chiu, Niki Parmar, Yu Zhang, Jiahui Yu, Wei Han, Shibo Wang, Zhengdong Zhang, Yonghui Wu, et al. Conformer: Convolution-augmented transformer for speech recognition. *arXiv preprint arXiv:2005.08100*, 2020.

[18] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR, 2016.

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

[21] B Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A Al Sallab, Senthil Yogamani, and Patrick Pérez. Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 2021.

[22] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. Nimble: Lightweight and parallel gpu task scheduling for deep learning. *Advances in Neural Information Processing Systems*, 33:8343–8354, 2020.

[23] Johannes Langer. *Band Matrices in Recurrent Neural Networks for Long Memory Tasks*. PhD thesis, 2018.

[24] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4013–4021, 2016.

[25] Yuhong Li, Xiaofan Zhang, and Deming Chen. Csrnet: Dilated convolutional neural networks for understanding the highly congested scenes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1091–1100, 2018.

[26] Liangkai Liu, Sidi Lu, Ren Zhong, Baofu Wu, Yongtao Yao, Qingyang Zhang, and Weisong Shi. Computing systems for autonomous driving: State of the art and challenges. *IEEE Internet of Things Journal*, 8(8):6469–6486, 2020.

[27] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rTasks. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 881–897, 2020.

[28] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 883–898, 2021.

[29] Onnx. https://onnx.ai/, 2019.

[30] Chao Peng, Xiangyu Zhang, Gang Yu, Guiming Luo, and Jian Sun. Large kernel matters–improve semantic segmentation by global convolutional network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4353–4361, 2017.

[31] Tensors and Dynamic neural networks in Python with strong GPU acceleration. https://pytorch.org, 2017.

[32] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[33] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, 2013.

[34] Gus Henry Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael Taylor, Luis Ceze, and Zachary Tatlock. Pure tensor program rewriting via access patterns (representation pearl). In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*, MAPS 2021, page 21–31, New York, NY, USA, 2021. Association for Computing Machinery.

[35] NVIDIA TensorRT: Programmable inference accelerator. https://developer.nvidia.com/tensorrt, 2017.

[36] Aravind Vasudevan, Andrew Anderson, and David Gregg. Parallel multi channel convolution using general matrix multiplication, 2017.

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[38] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. Pet: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 37–54, 2021.

[39] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[40] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 863–879, 2020.

[41] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. Amos: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *ISCA*, pages 874–887, 2022.

[42] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 859–873, 2020.

A Artifact Appendix

Abstract

This artifact appendix helps the readers reproduce the main evaluation results of the paper: EINNET: Optimizing Tensor Programs with Derivation-Based Transformations.

Scope

This artifact can be used for evaluating and reproducing the main results of the paper, including the model-level evaluation, operator-level evaluation, and the ablation studies and hyper-parameter studies on search strategies.

Contents

The following evaluation results are contained in the artifact:

E1: An end-to-end performance comparison between EinNet and other frameworks. (Figure 12)

E2: Performance studies on the cases in §7.3. (Table 3)

E3: Operator performance before and after optimization on the math libraries and code generation framework Ansor. (Figure 15)

E4: Speedup under different maximum search depths. (Figure 16)

E5: Search time with different MaxDepth and the number of explorative derivation steps with and without converging derivation. (Figure 17)

E6: Ablation study of expression fingerprint pruning. (Figure 18)

Hosting

The source code of this artifact can be found on GitHub: https://github.com/zhengly123/OSDI23-EinNet-AE, the `main` branch, with the commit ID `26a47d9`.

Requirements

Hardware dependencies

Dual Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz, NVIDIA A100-PCI-40GB GPU, NVIDIA V100-PCIE-32GB GPU.

Software dependencies

The artifact is evaluated on Ubuntu 22.04 LTS (Linux kernel 5.15.0-58). The artifact relies on CUDA 11.0.2 and cuDNN 8.0.3. The following frameworks are required as baselines:

1. TensorFlow 2.4
2. TensorRT 8.0
3. PET 1.0
4. Nimble with the commit ID bac6d10
5. TVM v0.10.0

Experiments workflow

The installation instruction and the following experiments are included in this artifact. All DNN benchmarks use synthetic input data in GPU device memory to remove the side effects of data transfers between CPU and GPU.

End-to-end performance (E1)

This experiment reproduces Figure 12 in the paper. Refer to OSDI23-EinNet-AE/0_model/README.md to prepare the environment and data. The detailed commands for each baseline are provided in separate run.sh and readme files in subdirectories.

Performance studies on the cases in §7.3 (E2)

See README.md and run.sh in OSDI23-EinNet-AE/1_op.

Operator performance (E3)

See README.md and run.sh in OSDI23-EinNet-AE/2_kernel_generator.

Speedup & Depth (E4)

See README.md and evaluate_max_depth.py in OSDI23-EinNet-AE/3_search_depth.

Search Time (E5)

See README.md and run.sh in OSDI23-EinNet-AE/4_search_time.

Ablation Study (E6)

See README.md and run.sh in OSDI23-EinNet-AE/5_fingerprint.