Group	Controlled Variable	gbs	hidden	seqlen	heads	pp	layers	dp	mb	tp
b	global batch size	*	4096	128	32	2	32	2	2	2
c	hidden dimension size	512	*	128	32	2	32	2	2	2
d	sequence length	32	128	*	32	2	32	2	2	2
e	heads	512	4096	128	*	2	32	2	2	2
f	pipeline parallelism	512	4096	128	32	*	32	2	2	2
g	layers	512	4096	128	32	4	*	16	8	8
h	data parallelism	512	4096	128	32	4	32	*	8	8
i	micro batches	512	4096	128	32	4	32	16	*	8
j	tensor parallelism	512	4096	128	32	4	32	16	8	*

Table 7. Configurations for variable controlled experiments.

`CP` ❶ to partially counteract the anti-scaling ❷ introduced by weight reducers in data parallelism ❸, ensuring that the final updates reflect per-input-sequence gradients. However, when `calculate-per-token-loss` is enabled, the averaging across the `DP×CP` communication group is skipped and instead replaced by averaging over the total number of trained tokens ❹. In this case, the combination of ❶ and ❹ results in the final gradients being over-scaled by a factor of `CP`.

TrainVerify can eliminate such bugs by comparing data flow of shape-reduced symbolic tensors. While the violation could be detected earlier via $L == L_0$, practical implementations typically do not enforce strict equivalence on distributed losses. Moreover, adapted graphs from manually-parallelized models lack the backward lineage, e.g. $t \overset{\mathcal{L}}{==} (t_0, t_1)$ that is naturally preserved by auto-parallel systems. As a result, TrainVerify detects the problem as soon as it visits a weight tensor in backward pass, (e.g., g_{w0}) by checking that its finalized gradient G_0 is consistent with G.

Such computational issues are subtle, making diagnosis particularly challenging, especially when the code spans multiple modules. The issue post reflects a 10-day effort involving users, developers, and volunteers to reproduce the problem and identify its root cause, amid early misdiagnoses and user concerns. In the year prior to that fix, over 5 issues were filed on the same code snippets, across various training configurations; some were misreported, while others were resolved after extensive discussion. TrainVerify can effectively alleviate such challenges and ensure verified correctness.

C Shape reduction correctness proof

For complete correctness proof of shape reduction, please refer to our external document at https://arxiv.org/abs/2506.15961.

CT: minor: - use bold for tensors; normal for scalar - 0-indexing vs. 1-based indexing (using $\vec{0}$ seems indicating 0-based) - indexing is in $\mathbb{I}$ not $\mathbb{R}$.

YM: consider Natural Numbers $\mathbb{N}$ or Integers $\mathbb{Z}$ rather than $\mathbb{I}$

When a DNN model involves large size tensors, such as popular LLMs, it becomes infeasible for current solvers (e.g., Z3) to verify its parallel execution considering the complexity, as tensors now have symbolic elements. In response, we propose a verification for the same model architecture but with reduced tensor shapes, and prove that: the verification conclusion on the shape-reduced model also applies to the original model with larger tensor shapes.

C.1 Formalization

A DNN model consists of multiple operators, such as `MatMul` and `ReLU`, which essentially are functions with data tensors as input and output. Given input tensor(s), including tensors representing weights, activation and optimizer state, such a DNN function can produce corresponding output tensor(s). Given a DNN function f that executes on a single device, there is an alternative function g (g is different from f) that can execute on either single device sequentially or multiple devices concurrently. Our goal is to verify that regardless of the inputs, f and g can always produce the same results—an equivalence.

Definition 8 (Tensor)**.** *A tensor is an object that generalizes scalars, vectors, and matrices to higher dimensions. Formally, an order-n tensor (also called an n-th order tensor) is an element of the tensor space:*

$$\mathcal{T} \in \mathbb{R}^{d_1 \times d_2 \times \cdots \times d_n} \tag{1}$$

where $\mathbb{R}$ represents real number field and $d_1, d_2, \ldots, d_n$ denote the dimensions along each mode of the tensor.

Tensors are used as primitive data in machine learning.

Definition 9 (Functions)**.** *A general function that operates on multiple tensors can be defined as:*

$$f : (\mathbb{R}^{d_1^a \times d_2^a \times \cdots \times d_n^a}, \mathbb{R}^{d_1^b \times d_2^b \times \cdots \times d_m^b}, \ldots) \rightarrow \mathbb{R}^{d_1^y \times d_2^y \times \cdots \times d_k^y} \tag{2}$$

where f takes one or more tensors as input and outputs a tensor of a potentially different shape.

For simplicity, we assume a single-tensor input/output for function f throughout this proof. The proof can be naturally extended to accommodate multiple input and output tensors.

We use bold symbols (e.g., $\mathbf{x}$, $\mathbf{y}$) to denote tensors and non-bold symbols (e.g., x, y) to denote scalars. We also use zero-based indexing; that is, for a vector $\mathbf{v}$, the first element is "$\mathbf{v}[0]$".

People have long observed that deep learning operators like element-wise operations and convolution are SIMD (Single-Instruction Multiple-Data): the operation consists of repeated, homogeneous computations (the "kernel") over array elements. This SIMD characteristic is the core enabler for our shape reduction mechanism. Below, we formally define what is a SIMD function.

Consider a function $f(\mathbf{x}) \rightarrow \mathbf{y}$, where $\mathbf{x} \in R^{d_1^a \times d_2^a \times \cdots \times d_m^a}$ and $\mathbf{y} \in R^{d_1^b \times d_2^b \times \cdots \times d_n^b}$. So, $rank(\mathbf{x}) = m$ and $rank(\mathbf{y}) = n$.

If f is a SIMD function, a kernel function θ associated with f takes a subtensor from $\mathbf{x}$ and outputs a scalar value. Formally:

Definition 10 (Kernel function)**.** *A kerenel function θ is a function that takes k scalar inputs and produces a single scalar output:*

$$\theta : \mathbb{R}^k \rightarrow \mathbb{R}.$$

Next, we define which input subtensor is associated with each output element. Consider the same function $f(\mathbf{x}) \rightarrow \mathbf{y}$. A dependency mapping τ associated with f is a function that maps each index $\mathbf{i}$ in the output $\mathbf{y}$ to a list of indices in the input $\mathbf{x}$. Formally:

Definition 11 (Dependency mapping)**.** *A dependency mapping τ is an affine transformation that maps a vector of integers (an index of tensor $\mathbf{y}$) to a list of indices in another tensor (i.e., $\mathbf{x}$):*

$$\tau : idx(\mathbf{y}) \in \mathbb{N}^n \rightarrow [idx(\mathbf{x}), \dots] \in \mathbb{N}^{k \times m},$$

where $idx(\cdot)$ is the indexing function of the tensor; n and m are ranks of $\mathbf{x}$ and $\mathbf{y}$; and k is the number of inputs in θ.

With dependency mapping and kernel function, we define SIMD functions.

Definition 12 (SIMD function)**.** *A function $f(\mathbf{x}) \rightarrow \mathbf{y}$ is a SIMD function if, for each $\mathbf{y}[\mathbf{i}], \mathbf{i} \in \mathbb{N}^n$,*

$$\mathbf{y}[\mathbf{i}] = \theta(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k),$$

where θ is the kernel function of f, and

$$\mathbf{x}_j = \mathbf{x}[\tau(\mathbf{i})[j]], \quad 1 \leq j \leq k$$

where τ is the dependency mapping of f.

By fixing the latent representation – kernel function θ and a dependency mapping τ – one can define an SIMD function. We denote an SIMD function f using its θ_f and τ_f as: $\mathbf{y}[\mathbf{i}] = \theta_f(\mathbf{x}[\tau_f(\mathbf{i})])$.

Finally, we introduce another class of operators, reductional opreations, such as `sum`. A reductional function $f : \mathbb{R}^m \rightarrow \mathbb{R}$ returns a single output element from processing a reductional operation among all elements in the input tensor, with the operation satisfying the commutative and associative laws.

Definition 13 (Reductional function)**.** *For an input tensor $\mathbf{x} \in \mathbb{R}^m$, the reductional function $f_\odot$ applies a binary operation $\odot$ to all elements of $\mathbf{x}$ such that:*

$$f_\odot(\mathbf{x}) = \mathbf{x}[0] \odot \mathbf{x}[1] \odot \cdots \odot \mathbf{x}[m-1],$$

and $\odot$ satisfies commutativity ($a \odot b = b \odot a$) and associativity ($(a \odot b) \odot c = a \odot (b \odot c)$).

C.2 Observations: LLM operators are SIMD functions

Deep Neural Network (DNN) computations are characterized by their application to high-dimensional data tensors. A closer examination of commonly used DNN operations reveals that a large number of elements in the output tensor share the same computational logic, differing only in the specific input elements they process. This computational pattern aligns closely with our definition of SIMD functions.

C.2.1 Observation 1: LLM operators have kernel functions. We observe that each computation operator in the transformer architecture is associated with its own kernel function, including Feed Forward layers, Multi-Head Attention layers (without masking), Add & Norm layers, ReLU, Softmax, and Residual Addition.

Consider matrix multiplication (i.e., MatMul) as an example. Given two matrices $\mathbf{A} \in \mathbb{R}^{m \times p}$ and $\mathbf{B} \in \mathbb{R}^{p \times n}$, the resulting matrix $\mathbf{C} \in \mathbb{R}^{m \times n}$ has elements $c_{i,j}$ (short for $\mathbf{C}[i][j]$) computed by: $c_{i,j} = \sum_{k=1}^{p} a_{i,k} \cdot b_{k,j}$. Therefore, MatMul has a kernel function:

$$\theta(a_{i,1}, \dots, a_{i,p}, b_{1,j}, \dots, b_{p,j}) = \sum_{k=1}^{p} a_{i,k} \cdot b_{k,j}$$

YC: we need to revisit the example as Matmul=simd+reduc

C.2.2 Observation 2: dependency mappings in LLM operators share linear components. This property is intuitive, as the "striding" of kernel functions across tensors typically occurs at regular, constant intervals. Consequently, when the input to the dependency mapping—corresponding to the output tensor's index—changes, the resulting input indices change linearly and follow the same pattern. That is, for each input tensor, the mapping takes the affline transformations:

$$\tau(\mathbf{i}) = [\mathbf{M} \cdot \mathbf{i} + \mathbf{b}_1, \ \dots, \ \mathbf{M} \cdot \mathbf{i} + \mathbf{b}_k].$$

For example, in the above MatMul case, the dependency mapping τ^A for the first input matrix $\mathbf{A}$ can be written as affine transformations:

$$\tau^A\left(\begin{bmatrix} i \\ j \end{bmatrix}\right) = [\mathbf{M}_A \begin{bmatrix} i \\ j \end{bmatrix} + \mathbf{b}_{A1}, \dots, \mathbf{M}_A \begin{bmatrix} i \\ j \end{bmatrix} + \mathbf{b}_{Ap}], \text{ where}$$

$$\mathbf{M}_A = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \mathbf{b}_{A1} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \dots, \mathbf{b}_{Ap} = \begin{pmatrix} 0 \\ p \end{pmatrix}$$

Above all, MatMul is a SIMD function because it has

- a kernel function:

$$\theta(a_{i,1}, \dots, a_{i,p}, b_{1,j}, \dots, b_{p,j}) = \sum_{k=1}^{p} a_{i,k} \cdot b_{k,j};$$

- a dependency mapping for each input tensor:

$$\tau^A([i,j]) = [[i,k] | 1 \le k \le p],$$
$$\tau^B([i,j]) = [[k,j] | 1 \le k \le p],$$

  where τ^A and τ^B are dependency mappings for input matrix A and B;
- and MatMul can be expressed as:

$$c_{i,j} = \theta(\mathbf{A}[\tau^A([i,j])] \oplus \mathbf{B}[\tau^B([i,j])]),$$

  where $\oplus$ represents vector concatenation.

YC: we may need a clearer separation/clarification of whether our SIMD include the reductional op (norm)

Fact 5. *We observe that in practice, the dependency mapping $\tau(\cdot)$ does not produce duplicated input indices. Meaning,*

$$\forall \mathbf{i}, \tau(\mathbf{i}) = [\mathbf{j}_1, \mathbf{j}_2, \ldots, \mathbf{j}_k] \in \mathbb{N}^{k \times m},$$
$$\textit{for } 1 \le a \ne b \le k \textit{ in } \tau(\mathbf{i}), \mathbf{j}_a \ne \mathbf{j}_b.$$

C.3 Correctness proof for shape reduction

This section establishes the correctness of TrainVerify's shape reduction by proving the equivalence between two data flow graphs (DFGs) at a reduced scale implies equivalence at the original scale. We denote the original and transformed DFGs—before and after applying parallelization techniques—as functions f and g, respectively.

C.3.1 Prerequisite relations. Before presenting the main theorem, we begin with two equivalent definitions that serve as the foundation for the proof.

Definition 14 (Mapping permutation equivalence)**.** *For two dependency mappings τ_1 and τ_2, we call them mapping permutation equivalence, denoted $\tau_1 \cong_P \tau_2$, if there exists a permutation function P, such that*

$$\forall i, \quad P(\tau_1(i)) = \tau_2(i)$$

Mapping permutation equivalence captures LLM operators with commutative and associative properties, where permuting the inputs does not affect the output. Similarly, we need to define a corresponding equivalence relation for kernel functions.

Definition 15 (Kernel permutation-set equivalence)**.** *For two kernel functions θ_1 and θ_2, we call them kernel permutation-set equivalence, denoted $\theta_1 \cong_Q \theta_2$, if there exists a non-empty set Q of permutation functions, such that*

$$\forall P \in Q, \ \forall \mathbf{x}, \quad \theta_1(\mathbf{x}) = \theta_2(P(\mathbf{x}))$$

Definition 16 (Well-formed kernel function)**.** *We call a kernel function θ well-formed if,*

YC: the "from" and "to" relation of the equation is not clear; the expression is not well aligned with the def semantics

$$\exists \mathbf{x}, \mathbf{x}', \forall i, \mathbf{x}[i] \ne \mathbf{x}'[i] \textit{ and } \forall j \ne i, \mathbf{x}[j] = \mathbf{x}'[j]$$
$$\theta(\mathbf{x}) \ne \theta(\mathbf{x}')$$

Essentially, all elements in the inputs to the well-formed kernel functions contribute to the final output. There is no such input element that does not influcent the output.

C.3.2 Premises from SMT solver. In TrainVerify, we use an SMT solver (Z3) to verify that a shape-reduced model preserves parallelization equivalence. Specifically, if the solver returns `sat`, it proves that for all inputs, the logical dataflow graph of the shape-reduced model is equivalent to that of the parallelized version.

This result yields a premise for each stage (§5.2) in TrainVerify of the form:

$$\forall \mathbf{x}, \forall \mathbf{i} \in \mathcal{I}, f(\mathbf{x})[\mathbf{i}] = g(\mathbf{x})[\mathbf{i}],$$
$$\text{where } \mathcal{I} = \{\sum_{j=0}^{n} a_j \mathbf{e}_j \mid a_j \in \{0,1\} \text{ for all } j\}$$

In the equation, $\mathbf{e}_i$ denotes the standard basis vectors in $\mathbb{R}^n$, defined as:

$$(\mathbf{e}_i)_j = \begin{cases} 1 & \text{if } j = i, \\ 0 & \text{otherwise.} \end{cases}$$

Each $\mathbf{e}_i \in \mathbb{N}^n$ is a column vector with a single 1 in the i-th position and 0 elsewhere, except for e_0 which is all 0s. For example,

$$\mathbf{e}_0 = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}_{n \times 1}, \mathbf{e}_1 = \begin{pmatrix} 1 \\ \vdots \\ 0 \end{pmatrix}_{n \times 1}, \ldots, \mathbf{e}_n = \begin{pmatrix} 0 \\ \vdots \\ 1 \end{pmatrix}_{n \times 1}$$

The above premise holds due to Algorithm 2, line 12, where TrainVerify enforces that, for any output dimension of each operator—excluding those not involved in computation (e.g., batch dimensions, or all dimensions in element-wise operations)—both the logical and parallelized dataflow graphs retain a size of at least two in those dimensions. Meanwhile, the equivalence for abitrary input $\mathbf{x}$ is established by the symbolic computation.

C.3.3 Main proofs. We now present the main proof of shape reduction correctness. The argument proceeds in three steps:

1. We first prove $\theta_f \cong_Q \theta_g$ given the above premise.
2. We then prove $\tau_f \cong_P \tau_g$ based on the premise.
3. Finally, we apply $\theta_f \cong_Q \theta_g$ and $\tau_f \cong_P \tau_g$ to establish the shape reduction theorem.

Next, we consider $f(\mathbf{x}) \to \mathbf{y}$ and $g(\mathbf{x}) \to \mathbf{y}'$, where $\mathbf{x} \in R^{d_1^a \times d_2^a \times \cdots \times d_m^a}$ and $\mathbf{y}, \mathbf{y}' \in R^{d_1^b \times d_2^b \times \cdots \times d_n^b}$. So, $rank(\mathbf{x}) = m$ and $rank(\mathbf{y}) = n$.

We start with a claim that if for all inputs $\mathbf{x}$, f and g give the same output at position $\mathbf{i}$, then the dependency mappings share the same set of indices.

Claim 6. For two well-formed SIMD functions f and g,

$$\forall \mathbf{x}, f(\mathbf{x})[\mathbf{i}] = g(\mathbf{x})[\mathbf{i}] \implies \exists! P, P(\tau_f(\mathbf{i})) = \tau_g(\mathbf{i}).$$

There exists exactly one permutation P between dependency mappings τ_f and τ_g.

Proof. Since f and g are SIMD functions, by Definition 12 and the premise, $\forall \mathbf{x}, \theta_f(\mathbf{x}[\tau_f(\mathbf{i})]) = \theta_g(\mathbf{x}[\tau_f(\mathbf{i})])$.

First, we prove the existence of P by contradition—assume there is no such a P: $set(\tau_f(\mathbf{i})) \neq set(\tau_g(\mathbf{i}))$. Then, there exists some element $\mathbf{j} \in \tau_f(\mathbf{i})$ but $\mathbf{j} \notin \tau_g(\mathbf{i})$. We can construct an input $\hat{\mathbf{x}}$ such that all elements other than j-th are 0; and $\hat{\mathbf{x}}[\mathbf{j}]$ can be an arbitrary number. Note that by premise, $\theta_f(\hat{\mathbf{x}}[\tau_f(\mathbf{i})]) = \theta_g(\hat{\mathbf{x}}[\tau_g(\mathbf{i})])$. By Definition 16, f and g are well-formed, so each input contributes meaningfully. Therefore, $\theta_f([0, \ldots, \hat{\mathbf{x}}[\mathbf{j}], \ldots, 0]) \neq \theta_g(\vec{0})$, a contradiction to the premise. This means $set(\tau_f(\mathbf{i})) = set(\tau_g(\mathbf{i}))$.

Finally, we prove P is the only possible permutation. By Fact 5, all elements in $set(\tau_f(\mathbf{i}))$, and correspondingly in $set(\tau_g(\mathbf{i}))$, are distinct scalars. Therefore, there exists a unique permutation P such that $P(\tau_f(\mathbf{i})) = \tau_g(\mathbf{i})$. □

Lemma 7. *For SIMD functions f and g with well-formed kernel functions:*

$$\forall \mathbf{x}, f(\mathbf{x})[\mathbf{e}_0] = g(\mathbf{x})[\mathbf{e}_0] \implies \theta_f \cong_Q \theta_g.$$

Proof. Recall $\mathbf{e}_0 = \vec{0} \in \mathbb{N}^n$. Then, we have $\forall \mathbf{x}, f(\mathbf{x})[\vec{0}] = g(\mathbf{x})[\vec{0}]$. Because f and g are SIMD functions, $\forall \mathbf{x}, \theta_f(\mathbf{x}[\tau_f(\vec{0})]) = \theta_g(\mathbf{x}[\tau_g(\vec{0})])$. By Claim 6, there exists a permuation, say P_0, such that $P_0(\tau_f(\vec{0})) = \tau_g(\vec{0})$.

We denote $\mathcal{X} = \mathbf{x}[\tau_f(\vec{0})]$. By Fact 5, $\tau_f(\vec{0})$ doesn't have duplicated indices, meaning $\mathcal{X}$ traces back to k unique positions of $\mathbf{x}$. Hence, $\mathcal{X}$ covers all possible inputs of $\mathbb{R}^k$, becaue $\mathbf{x}$ is an arbitrary $\mathbb{R}^{d_1 \times \ldots d_m}$ tensor.

So, we have:

$$\begin{aligned}
&\forall \mathbf{x},\ \theta_f(\mathbf{x}[\tau_f(\vec{0})]) = \theta_g(\mathbf{x}[\tau_g(\vec{0})]) \\
&\quad \Rightarrow \theta_f(\mathbf{x}[\tau_f(\vec{0})]) = \theta_g(\mathbf{x}[P_0(\tau_f(\vec{0}))]) && \text{[Claim 6]} \\
&\quad \Rightarrow \theta_f(\mathbf{x}[\tau_f(\vec{0})]) = \theta_g(P_0(\mathbf{x}[\tau_f(\vec{0})])) && \text{[tensor indexing]} \\
&\quad \Rightarrow \theta_f(\mathcal{X}) = \theta_g(P_0(\mathcal{X})) && [\mathcal{X} = \mathbf{x}[\tau_f(\vec{0})]] \\
&\quad \Rightarrow \theta_f \cong_Q \theta_g && \text{[Definition 15]}
\end{aligned}$$

In addition, P_0 satisfies permutation requirments in Definition 15, hence:

$$P_0 \in Q, \tag{3}$$

where Q is the permutation set in $\theta_f \cong_Q \theta_g$. □

XXX: assume $n > k$?

YC: we can trivially assume $n > k$, else the the kernel func cannot apply

Lemma 8. *For SIMD functions f and g with well-formed kernel functions:*

$$\forall \mathbf{x}, \forall i \in \{0, \ldots, n\}, f(\mathbf{x})[\mathbf{e}_i] = g(\mathbf{x})[\mathbf{e}_i] \implies \tau_f \cong_P \tau_g.$$

Proof. Consider $i = 0$; that is $\mathbf{e}_0 = \vec{0}$.

$$\begin{aligned}
&\forall \mathbf{x}, f(\mathbf{x})[\vec{0}] = g(\mathbf{x})[\vec{0}] \\
&\quad \Rightarrow \theta_f(\mathbf{x}[\tau_f(\vec{0})]) = \theta_g(\mathbf{x}[\tau_g(\vec{0})]) \\
&\qquad\qquad \text{[Definition 12]} \\
&\quad \Rightarrow \theta_f(\mathbf{x}[\mathbf{M}_f \cdot \vec{0} + \mathbf{b}_f]) = \theta_g(\mathbf{x}[\mathbf{M}_g \cdot \vec{0} + \mathbf{b}_g]) \\
&\qquad\qquad \text{[Definition 11, affine transformation]} \\
&\quad \Rightarrow \theta_f(\mathbf{x}[[\mathbf{b}_{f1}, ..., \mathbf{b}_{fk}]) = \theta_g(\mathbf{x}[[\mathbf{b}_{g1}, ..., \mathbf{b}_{gk}]]) \\
&\qquad\qquad \text{[expanding } \mathbf{b}\text{]}
\end{aligned}$$

By Claim 6, there exists a unique permutation, say P_0, such that $P_0([\mathbf{b}_{f1}, \ldots, \mathbf{b}_{fk}]) = [\mathbf{b}_{g1}, \ldots, \mathbf{b}_{gk}]$.

Similarly, consider $i = 1$ for the premise, which gives $\forall \mathbf{x}, f(\mathbf{x})[\mathbf{e}_1] = g(\mathbf{x})[\mathbf{e}_1]$, where $\mathbf{e}_1 = [1, 0, 0, \ldots] \in \mathbb{N}^n$.

$$\begin{aligned}
&\forall \mathbf{x}, f(\mathbf{x})[\mathbf{e}_1] = g(\mathbf{x})[\mathbf{e}_1] \\
&\quad \Rightarrow \theta_f(\mathbf{x}[\tau_f(\mathbf{e}_1)]) = \theta_g(\mathbf{x}[\tau_g(\mathbf{e}_1)]) \\
&\quad \Rightarrow \theta_f(\mathbf{x}[M_f \cdot \mathbf{e}_1 + \mathbf{b}_{f1}], \ldots) = \theta_g(\mathbf{x}[M_g \cdot \mathbf{e}_1 + \mathbf{b}_{g1}], \ldots)
\end{aligned}$$

By Claim 6, there exists a unique permutation, say P_1, such that $P_1([M_f \cdot \mathbf{e}_1 + \mathbf{b}_{f1}, \ldots]) = [M_g \cdot \mathbf{e}_1 + \mathbf{b}_{g1}, \ldots]$.

We repeat this for all $\mathbf{e}_i, i \in [0, n]$. Then, we have:

$$\begin{cases}
P_0([\mathbf{b}_{f1}, \ldots, \mathbf{b}_{fk}]) = [\mathbf{b}_{g1}, \ldots, \mathbf{b}_{gk}] \\
P_1([\mathbf{M}_f \cdot \mathbf{e}_1 + \mathbf{b}_{f1}, \ldots]) = [\mathbf{M}_g \cdot \mathbf{e}_1 + \mathbf{b}_{g1}, \ldots] \\
P_2([\mathbf{M}_f \cdot \mathbf{e}_2 + \mathbf{b}_{f1}, \ldots]) = [\mathbf{M}_g \cdot \mathbf{e}_2 + \mathbf{b}_{g1}, \ldots] \\
\vdots \\
P_n([\mathbf{M}_f \cdot \mathbf{e}_n + \mathbf{b}_{f1}, \ldots]) = [\mathbf{M}_g \cdot \mathbf{e}_n + \mathbf{b}_{g1}, \ldots]
\end{cases}$$

By Claim 9 (which we prove below), all permutations are equivalent, $P_0 = \cdots = P_n$.

Now, we prove $\tau_f \cong_{P_0} \tau_g$. By Definition 14, we need to prove $\forall \mathbf{i} \in \mathbb{N}^n, P_0(\tau_f(\mathbf{i})) = \tau_g(\mathbf{i})$. Notice that $\mathbf{e}_i$'s are standard basis vectors, so any $\mathbf{i}$ is a linear combination of $\mathbf{e}_i$s:

$$\mathbf{i} = a_0\mathbf{e}_0 + a_1\mathbf{e}_1 + a_2\mathbf{e}_2 + \cdots + a_n\mathbf{e}_n$$

where $a_i \in \mathbb{R}$.

$$\begin{aligned}
\tau_g(\mathbf{i}) &= [\sum_{i=1}^{n} a_i M_{g1} \cdot \mathbf{e}_i + \mathbf{b}_{g1}, \sum_{i=1}^{n} a_i M_{g2} \cdot \mathbf{e}_i + \mathbf{b}_{g2}, \ldots] \\
&= P_0([\sum_{i=1}^{n} a_i M_{f1} \cdot \mathbf{e}_i + \mathbf{b}_{f1}, \sum_{i=1}^{n} a_i M_{f2} \cdot \mathbf{e}_i + \mathbf{b}_{f2}, \ldots]) \\
&= P_0(\tau_f(\mathbf{i}))
\end{aligned}$$

Therefore, $\tau_f \cong_{P_0} \tau_g$.

□

Claim 9. *Consider the following $n+1$ equations:*

$$\begin{cases} P_0([\mathbf{b}_{f1},\dots,\mathbf{b}_{fk}]) = [\mathbf{b}_{g1},\dots,\mathbf{b}_{gk}] \\ P_1([\mathbf{M}_f\cdot\mathbf{e}_1+\mathbf{b}_{f1},\dots]) = [\mathbf{M}_g\cdot\mathbf{e}_1+\mathbf{b}_{g1},\dots] \\ P_2([\mathbf{M}_f\cdot\mathbf{e}_2+\mathbf{b}_{f1},\dots]) = [\mathbf{M}_g\cdot\mathbf{e}_2+\mathbf{b}_{g1},\dots] \\ \vdots \\ P_n([\mathbf{M}_f\cdot\mathbf{e}_n+\mathbf{b}_{f1},\dots]) = [\mathbf{M}_g\cdot\mathbf{e}_n+\mathbf{b}_{g1},\dots] \end{cases} \tag{4}$$

We claim that

$$\forall k, \textit{Equation 4} \Rightarrow P_0 = \cdots = P_n.$$

Proof. We prove this claim by contradiction. Without loss of generality, consider P_0 is identity mapping, and assume $P_1 \neq P_0$, meaning

$$\begin{cases} [\mathbf{b}_{f1},\dots,\mathbf{b}_{fk}] = [\mathbf{b}_{g1},\dots,\mathbf{b}_{gk}] \\ P_1([\mathbf{M}_f\cdot\mathbf{e}_1+\mathbf{b}_{f1},\dots]) = [\mathbf{M}_g\cdot\mathbf{e}_1+\mathbf{b}_{g1},\dots] \end{cases}$$

Next, we denote $P_1([\mathbf{M}_f\cdot\mathbf{e}_1+\mathbf{b}_{f1},\dots])$ as $[\mathbf{M}_f\cdot\mathbf{e}_1+\mathbf{b}_{P1(f1)},\dots]$, and replace $\mathbf{b}_{gi}$ with the corresponding $\mathbf{b}_{fi}$, so we have:

$$[\mathbf{M}_f\cdot\mathbf{e}_1+\mathbf{b}_{P1(f1)},\dots] = [\mathbf{M}_g\cdot\mathbf{e}_1+\mathbf{b}_{f1},\dots]$$

By rearraning this, we get:

$$\begin{cases} \mathbf{b}_{f1}-\mathbf{b}_{P1(f1)} = (\mathbf{M}_f-\mathbf{M}_g)\cdot\mathbf{e}_1 \\ \mathbf{b}_{f2}-\mathbf{b}_{P1(f2)} = (\mathbf{M}_f-\mathbf{M}_g)\cdot\mathbf{e}_1 \\ \vdots \\ \mathbf{b}_{fk}-\mathbf{b}_{P1(fk)} = (\mathbf{M}_f-\mathbf{M}_g)\cdot\mathbf{e}_1 \end{cases}$$

Because P_1 is not identity, by Fact 5, $\exists j \in [1,k]$, $\mathbf{b}_{fj}-\mathbf{b}_{P1(fj)} \neq \vec{0} \in \mathbb{R}^m$, therefore

$$\begin{cases} [\mathbf{b}_{f1}[0],\dots]-[\mathbf{b}_{P1(f1)}[0],\dots] \\ = [\mathbf{b}_{f2}[0],\dots]-[\mathbf{b}_{P1(f2)}[0],\dots] \\ \vdots \\ = [\mathbf{b}_{fk}[0],\dots]-[\mathbf{b}_{P1(fk)}[0],\dots] \\ \neq [0,0,\dots] \end{cases}$$

This means at least one dimension, say $i \in [0,m)$, have the following equation:

$$\begin{cases} \mathbf{b}_{f1}[i]-\mathbf{b}_{P1(f1)}[i] \\ = \mathbf{b}_{f2}[i]-\mathbf{b}_{P1(f2)}[i] \\ \vdots \\ = \mathbf{b}_{fk}[i]-\mathbf{b}_{P1(fk)}[i] \neq 0 \end{cases}$$

Consider the value $\mathbf{b}_{f1}[i]-\mathbf{b}_{P1(f1)}[i]$, which must be either positive or negative ($\neq 0$). Without loss of generality, assume it is positive; that is $\mathbf{b}_{f1}[i] > \mathbf{b}_{P1(f1)}[i]$. Since $P1$ is a permuation, one can always locate a corresponding term where $\mathbf{b}_{P1(f_i)}$ appears as the minuend (the left operand of the subtraction). This yields another inequality $\mathbf{b}_{P1(f1)}[i] > \mathbf{b}_{P1(fo)}[i]$. By repeating this reasoning iteratively, we eventually encounter a subtraction in which $\mathbf{b}_{f1}[i]$ appears as the subtrahend (the right opearand). This results in a a contradition of the form $\mathbf{b}_{f_i}[i] > \cdots > \mathbf{b}_{f_i}[i]$. Hence, the contradiction implies that P_1 must be equavalent to P_0.

By applying the above reasoning for all $\mathbf{e}_i$s, we conclude $P_0 = \cdots = P_n$.

□

Next, we prove one of our main theorem below.

Theorem 10. *For SIMD functions f and g with well-formed kernel functions:*

$$\forall \mathbf{x}, \forall i \in \{0,\dots,n\}, f(\mathbf{x})[e_i] = g(\mathbf{x})[e_i] \implies \forall \mathbf{j}, \mathbf{x}, f(\mathbf{x})[\mathbf{j}] = g(\mathbf{x})[\mathbf{j}]$$

Proof. Given the premise:

- By Lemma 7, $\theta_f \cong_Q \theta_g$.
- By Lemma 8, $\tau_f \cong_P \tau_g$.
- By Equation 3, $P \in Q$.

Finally, we prove

$$\theta_f \cong_Q \theta_g \ \wedge\ \tau_f \cong_P \tau_g \ \wedge\ P \in Q \implies f = g$$

$$\begin{aligned} \forall \mathbf{x}, \forall i, f(\mathbf{x})[i] &= \theta_f(\mathbf{x}(\tau_f(i))) && \text{[by Definition 12]} \\ &= \theta_g(P(\mathbf{x}[\tau_f(i)])) && \text{[by } \theta_f \cong_Q \theta_g \wedge P \in Q\text{]} \\ &= \theta_g(\mathbf{x}[P(\tau_f(i))]) && \text{[by tensor indexing rules]} \\ &= \theta_g(\mathbf{x}[\tau_g(i)]) && \text{[by } \tau_f \cong_P \tau_g\text{]} \\ &= g(\mathbf{x})[i] \end{aligned}$$

Because for any input $\mathbf{x}$, $f(\mathbf{x})$ and $g(\mathbf{x})$ produce the same result, therefore $f = g$.

□

In the following, we prove the shape reduction equavalence for reductional operations.

Theorem 11. *Given reductional functions $f_\odot$ and $g_\oplus$,*

$$\forall \mathbf{x} \in \mathbb{R}^2, f_\odot(\mathbf{x}) = g_\oplus(\mathbf{x}) \implies \forall \mathbf{x} \in \mathbb{R}^n, n >= 2, f_\odot(\mathbf{x}) = g_\oplus(\mathbf{x}).$$

Proof. We prove the lemma by mathematical induction.

Base case. Consider the base case, $f(\mathbf{x}) = g(\mathbf{x}) |\ \mathbf{x} \in \mathbb{R}^2$; namely, $\forall \mathbf{x}, \mathbf{x}[0] \odot \mathbf{x}[1] = \mathbf{x}[0] \oplus \mathbf{x}[1]$. This equality holds directly from the given premise.

Inductive step. Assume that $f_\odot(\mathbf{x}) = g_\oplus(\mathbf{x})$ holds for $\forall \mathbf{x} \in \mathbb{R}^k, k >= 2$. Next, we prove that $f_\odot(\mathbf{x}) = g_\oplus(\mathbf{x})$ also holds for $\forall \mathbf{x} \in R^{k+1}$.

We denote $\mathbf{x} \in \mathbb{R}^{k+1}$ as $[\mathbf{x}[0..k-1], \mathbf{x}[k]]$, then:

$$
\begin{aligned}
f_{\odot}(\mathbf{x}) &= f_{\odot}(\mathbf{x}[0..k-1]) \odot \mathbf{x}[k] && \text{[Definition 13]} \\
&= g_{\oplus}(\mathbf{x}[0..k-1]) \odot \mathbf{x}[k] && \text{[Inductive hypothesis]} \\
&= g_{\oplus}(\mathbf{x}[0..k-1]) \oplus \mathbf{x}[k] && \text{[Base case]} \\
&= g_{\oplus}(\mathbf{x}) && \text{[Definition 13]}
\end{aligned}
$$

□

C.4 Connecting theorems to practice

In this section, we prove that TrainVerify's checking algorithm is correct, meaning if our verifier (i.e., Z3) accepts, then the logical DFG is semantically equivalent to the parallelized DFG executed by machines.

The key idea is that LLM operators are either SIMD functions (Definition 12) or reductional functions (Definition 13), or (semantically) combination of the two. TrainVerify verifies equivalence by checking two small sub-tensors from the outputs of the logical DFG and the parallelized DFG. Next, we prove these sub-tensors are sufficient to guarantee that all other corresponding parts of the outputs are identical. For example, by theorem 10 and theorem 11, verifying the equivalence of the operator $\texttt{MatMul}(A, B)$, where $A \in R^{[m,k]}, B \in R^{[k,n]}$ with $m, k, n \in \mathbb{Z}^+$, can be simplified by verifying the case of $m, n, k = 2$.