Optimizing Deep Learning Inference via Global Analysis and Tensor Expressions

Chunwei Xia
C.Xia@leeds.ac.uk
SKLP, ICT, CAS
UCAS, China
University of Leed, U. K.
China

Jiacheng Zhao*
zhaojiacheng@ict.ac.cn
SKLP, ICT, CAS
UCAS, China
China

Qianqi Sun
sunqianqi18@mails.ucas.ac.cn
SKLP, ICT, CAS
UCAS, China
China

Zheng Wang
z.wang5@leeds.ac.uk
University of Leeds
U. K.

Yuan Wen
yuan.wen@abdn.ac.uk
University of Aberdeen
U. K.

Teng Yu
sanwan.yu@thewakesystems.com
Thewake Systems, China
China

Xiaobing Feng
fxb@ict.ac.cn
SKLP, ICT, CAS
UCAS
Zhongguancun Laboratory, China
China

Huimin Cui
cuihm@ict.ac.cn
SKLP, ICT, CAS
UCAS, China
China

Abstract

Optimizing deep neural network (DNN) execution is important but becomes increasingly difficult as DNN complexity grows. Existing DNN compilers cannot effectively exploit optimization opportunities across operator boundaries, leaving room for improvement. To address this challenge, we present Souffle, an open-source compiler that optimizes DNN inference across operator boundaries. Souffle creates a global tensor dependency graph using tensor expressions, traces data flow and tensor information, and partitions the computation graph into subprograms based on dataflow analysis and resource constraints. Within a subprogram, Souffle performs local optimization via semantic-preserving transformations, finds an optimized program schedule, and improves instruction-level parallelism and data reuse. We evaluated Souffle using six representative DNN models on an NVIDIA A100 GPU. Experimental results show that Souffle consistently outperforms six state-of-the-art DNN optimizers by delivering a geometric mean speedup of up to 3.7× over TensorRT and 7.8× over Tensorflow XLA.

CCS Concepts: • **Computer systems organization** → **Multicore architectures**; **Single instruction, multiple data**; **Neural networks**; **Heterogeneous (hybrid) systems**; • **Software and its engineering** → **Source code generation**; **Application specific development environments**.

Keywords: Deep Neural Network, Compiler Optimization, Tensor Expression, GPU

*Jiacheng Zhao is the corresponding author.

1 Introduction

No day goes by without hearing about the success of deep neural networks (DNNs). Indeed, advanced DNNs have demonstrated breakthrough effectiveness in solving a wide range of tasks, from drug discovery [11, 16] and self-driving cars [28] to e-commerce [26, 59].

A DNN model is typically expressed as a computational graph using deep learning (DL) programming frameworks like TensorFlow [2] and PyTorch [41]. By separating the expression of the computational graph from the implementation of low-level operators, DL frameworks abstract away the hardware complexity and have become the standard method for writing DNN code. However, using high-level programming abstractions presents significant challenges for low-level performance optimization, especially during model inference when deploying a trained model in a production environment where the response time is crucial [17, 21].

Efforts have been made to perform optimizations across the operator boundaries to increase parallelism, decrease memory access traffic or utilize memory bandwidth more efficiently. One promising approach is *operator/kernel fusions*, which involves merging multiple operators into a single kernel to enable analysis and optimizations across operators. This line of research includes works using hand-crafted rules [37], loop analysis [56], or just-in-time compilation [58] to guide and perform fusions. Typically, these methods use a bottom-up strategy by first performing operator fusion in the graph representation to merge multiple operators into a partition and then generating an optimized kernel for each

partition. However, a key challenge is determining the optimal boundaries of partitions or which operators should be fused together.

Despite the success of bottom-up approaches to operator/kernel fusion, optimization opportunities can still be overlooked. One such issue arises from *separating the operator fusion and code generation stages*. This can result in misplaced operators into different kernels, leading to extra memory access overhead and preventing otherwise possible optimizations. As we will show in the paper, state-of-the-art kernel fusion methods can miss important optimization opportunities, leaving much room for improvement.

We present Souffle, a novel *top-down* approach for optimizing inference across DNN operator boundaries. Unlike bottom-up strategies, Souffle first processes the whole computation graph as a single, merged kernel and then divides the graph into partitions, i.e., subprograms, through a global analysis from the top-level, considering data reuse in shared memory/registers and the generated instructions when determining partitioning boundaries. Each partition would be organized into a kernel. Afterwards, at the bottom level, Souffle performs a series of semantic preserving transformations for each subprogram to simplify the tensor expression and eliminate redundant memory access for the corresponding kernel. To this end, Souffle introduces two new mechanisms: a *tensor-expression-based global analysis* to identify critical partitioning points and a *semantic preserving transformations* approach that uses affine transformation to simplify the tensor expressions of each subprogram. Compared with existing bottom-up fusion approaches, the benefit of our top-down approach is that it globally determines the kernel boundaries by considering the generated code of the kernels.

Tensor-expression-based global analysis. Souffle conducts global dependence analysis on a tensor dependency graph generated from the entire DNN model. It utilizes tensor expressions (TEs) [12] to encode dataflow information of operators and tensors. By mapping higher-level operators to simpler TEs, Souffle performs data-flow analysis and code optimization around these TEs, simplifying the complexity of analysis and optimization and resulting in better code. TEs offer concise semantics, allowing us to translate the task of analyzing and optimizing complex operators to a more manageable problem of analyzing and optimizing simpler mathematical expressions. For instance, a *softmax* operator can be represented by two TEs with simpler data dependence relationships: one is a *one-relies-on-many* TE (reduction), and the other is a *one-relies-on-one* TE (element-wise). Since Souffle's analysis is conducted on the TEs without making any assumptions of low-level library calls, it can optimize across complex operators, even when the operators have complex data dependency like many-to-many, when other methods fail to do so.

Semantic-preserving transformation. After the top-level stage, the computation graph has been divided into multiple subprograms with each subprogram being mapped to a kernel. However, each subprogram contains a large number of TEs which would introduce a large number of redundant memory accesses across these TEs. Therefore, Souffle applies affine transformations to combine multiple TEs to a single TE thus eliminating the redundant memory accesses. This process is performed within the subprograms and relies on the TE-based global analysis. The transformation is fully automated and flexible as the tensor expression precisely describes the mathematical computation of operators in a simple form.

Putting it all together. Souffle first conducts data-flow analysis on the tensor dependency graph of the entire DNN model using TEs. This analysis captures essential information such as tensor shapes and live ranges across operator boundaries, allowing for precise element-wise analysis to infer data dependence. Souffle then partitions the TEs into subprograms using compiler heuristics and conducts local optimization within each subprogram using semantic-preserving mathematical transformations to reduce memory accesses. The optimized subprogram schedule is found by considering the computation characteristics of the subprogram's TEs. With precise dependence information at the TE level, Souffle can optimize memory access latency by reusing tensor buffers and improve instruction-level parallelism by overlapping memory load and arithmetic instructions. Since Souffle's code optimizations are based on subprograms of fused operators rather than individual operators, the optimization boundary of operators is eliminated.

Evaluation. We have implemented a working prototype of Souffle[1] on TVM. We evaluate Souffle on six DNN models with diverse and representative model architectures and compare it against six state-of-the-art DL optimizing and kernel fusion frameworks, including XLA [27], Ansor [57], TensorRT [38], Rammer [33], Apollo [56], and the MLIR-based IREE compiler [1]. Our evaluation, performed on an NVIDIA A100 GPU, shows that Souffle outperforms existing solutions, delivering a geometric mean speedup of up to 3.7× and 7.8× over TensorRT and XLA, respectively. Souffle is highly flexible and can fuse operators where state-of-the-art kernel fusion strategies fail. It is compatible with TensorFlow and ONNX [13] models, and can be integrated with general DL compilers like TVM [12, 57].

Contributions. This paper makes the following contributions:

- It presents a new top-down approach for identifying and exploiting optimization opportunities across operator boundaries (Sec. 5);

[1]The data and code associated with this paper are openly available at: https://github.com/SOUFFLE-AE/SOUFFLE.git.

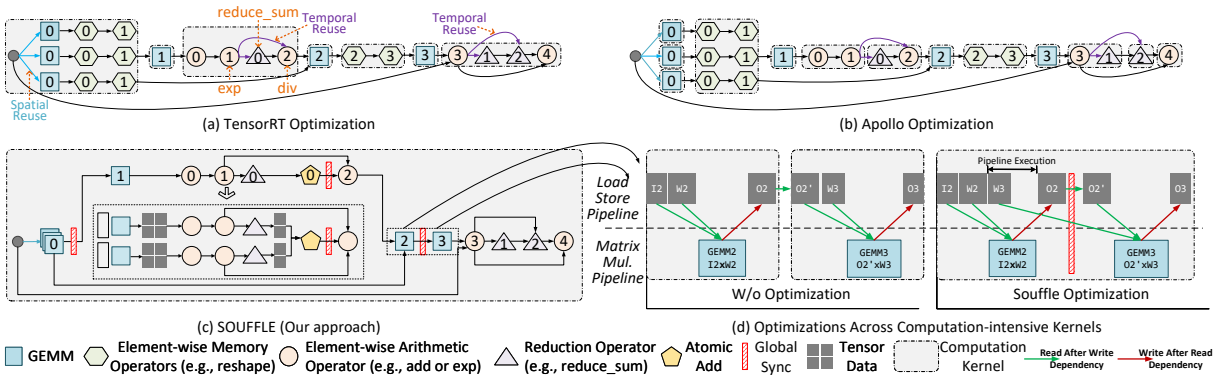

Figure 1. How TensorRT (a), Apollo (b) and SOUFFLE (c) map a BERT computation graph into kernels. The SOUFFLE optimization leads to fewer GPU memory accesses and faster execution time than TensorRT and Apollo.

Table 1. Performance for the generated kernels of Fig. 1.

	TensorRT	Apollo	SOUFFLE
Total execution time(μs)	62.34	179.07	57.73
–Computation-intensive kernels	31.29	61.1	41.77
–Memory-intensive kernels	31.0	117.97	15.96
#Kernels	7	14	1
#Bytes load from global (M)	16.52	27.78	8.87

- It shows how to effectively leverage the global analysis to perform local optimization at the kernel level represented as tensor expressions (Sec. 6);
- It demonstrates how low-level tensor expressions can be employed to perform instruction optimizations beyond operator boundaries (Sec. 6.5).

2 Motivation

2.1 Working Example

As a motivation example, consider optimizing a standard BERT model [14] on an NVIDIA A100 GPU. This model is based on the Transformer architecture [10, 49] and is using FP16 for inference. Fig. 1 depicts how TensorRT and Apollo map operators of a simplified sub-computation graph from BERT into kernels[2]. This subgraph contains representative DNN operators like general matrix multiplication *GEMM*, *reshape*, *permutation*, element-wise arithmetic operators like *add* or *exp*, and reduction operators like *reduce_sum*. The compiler maps these operators to individual kernels, which significantly impacts performance.

2.2 Performance Evaluation

We measure the resulting kernels using the NVIDIA Nsight Compute[39]. Table 1 shows that neither TensorRT nor Apollo can provide an optimal mapping for the evaluated DNN. The subgraph created by TensorRT and Apollo in Fig. 1 loads 16.52MB and 27.78MB of data from global memory, giving an execution time of 62.34μs and 179.1μs, respectively. A better strategy, which is the one chosen by our approach, is to refine and map the subgraph into a single kernel. This strategy reduces the number of bytes loaded from the global memory to 8.87M with an execution time of 57.7μs, translating to 1.1× and 3.1× faster running time than TensorRT and Apollo, respectively. We want to highlight that TensorRT has been specifically tuned for Transformer-based models with close-sourced, hand-optimized low-level operator implementations (like *GEMM*). Therefore, we consider the SOUFFLE improvement over TensorRT on BERT to be significant given that SOUFFLE does not have access to some of the NVIDIA-optimized operator implementations used by TensorRT. Furthermore, as we will show later in Sec. 8, SOUFFLE also significantly outperforms other DNN compilers, including XLA and IREE, on this DNN model.

2.3 Missed Opportunities

After closely examining the profiling data and the kernel fusion outcomes, we identified several optimization opportunities that TensorRT and Apollo miss:

Fail to explore optimization between memory- and compute-intensive kernels. As depicted in Fig. 1, part of BERT requires to perform element-wise memory operators, e.g. *reshape* and *permutation* (Element-wise memory operators 2 and 3 in Fig. 1). TensorRT and Apollo leverage manually crafted rules to fuse adjacent element-wise memory operators together while both of them fail to further perform optimization between the fused operators and their precedent computation operators, e.g. (*GEMM*) operators in Fig. 1. SOUFFLE performs optimization between memory- and compute-intensive kernels, and eventually eliminates all element-wise memory operators. In summary, manually crafted rules cannot cover a diverse set of computation patterns and miss the optimization opportunity in this case.

[2]Layout transformation kernels are omitted from Fig. 1 to aid clarity.

Suboptimal fusion strategy for reduction operators. Fig. 1(a) and (b) show the suboptimal kernel fusion strategy employed by TensorRT and Apollo for reduction operators. Both strategies choose to map the *GEMM* and the reduction operator to separate kernels, which requires storing the entire tensor data that reduction operators rely on to expensive global memory before reduction occurs. Souffle aggressively fuses reduction operators with adjacent computation-intensive operators, such as *R0-2*(R for Reduction Operator) with *GEMM0* and *GEMM1*, as shown in Fig. 1(c). This is achieved through a two-phase reduction approach: performing partial reduction in a per-block fashion and using *atomicAdd* for global reduction. As a result, the entire tensor data can be kept on-chip, with only the partial result stored in global memory. A *global synchronization* (e.g. grid synchronization in CUDA [40]) is inserted to synchronize running blocks, as shown in Fig. 1(c). This optimization applies to all reduction operators in Fig. 1. Moreover, Souffle can cache the output of *arithmetic operator 1* on-chip for reuse in *arithmetic operator 2*.

Poor optimizations across computation-intensive kernels. Like many other DNN frameworks, TensorRT and Apollo try to fuse multiple computation-intensive operators of the same type, but fail to capitalize on the opportunities across different types of operators. Consider Fig. 1(d) that shows how two dependent *GEMM* operators execute asynchronous memory copies and tensor core computations when they are grouped to kernels under two different strategies. The first is to map the *GEMM* operators into two separate kernels, as they do not consider fuse compute-intensive operators. The second is to map them to a single kernel. TensorRT and Apollo use the former, and Souffle uses the latter. By putting two GEMM operators into one single kernel (right part of Fig.1(d)), Souffle allows the pipeline execution of loading *W3* of *GEMM3* while computing *GEMM2*. Souffle is designed to take such cross-operator pipeline optimizations.

2.4 Our Insights

Based on the observations outlined earlier, there is a need to analyze DNN models to fuse operators, perform automatic transformations on tensor operations, and optimize within a fused kernel. A more effective kernel fusion strategy makes extracting crucial tensor information such as live range and tensor data reuse possible. This information can then be used to analyze the fine-grained dependencies at the element-wise level, leading to better kernel-level optimization.

3 Preliminaries

Souffle utilizes TVM's *tensor expression* (TE) [12] as an intermediate representation for analysis and optimization. The TE specifies how output elements are computed from input tensors. In the TE Program shown in Figure 2, *TE0* is an example TE for the *GEMM*, where the *rk* parameter defines the

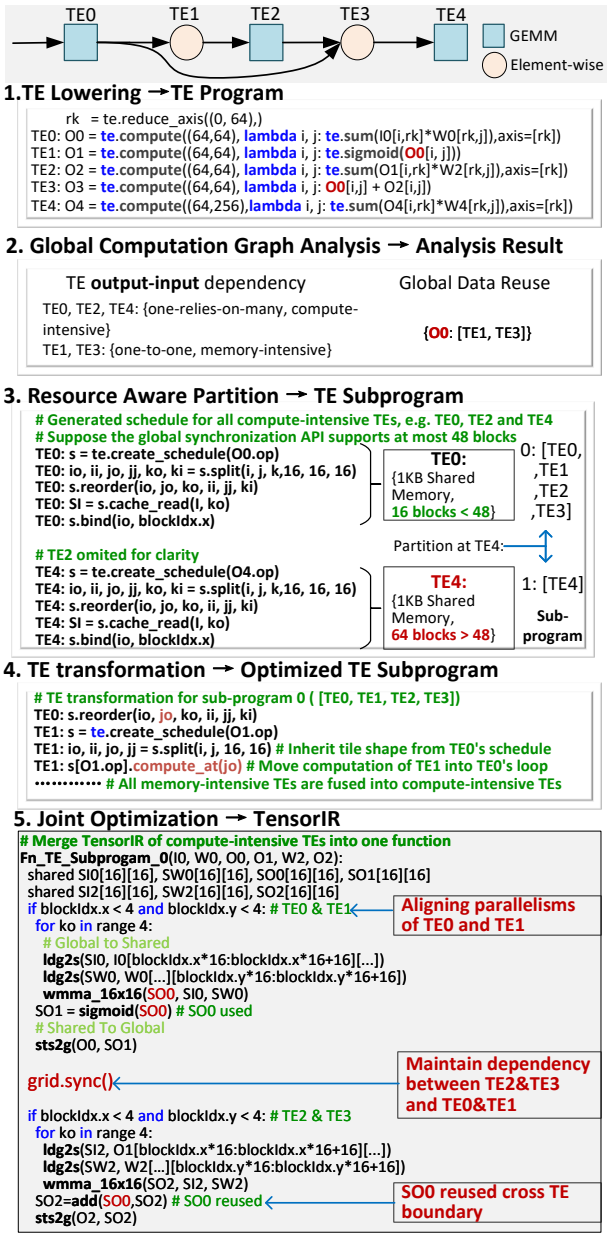

Figure 2. Example of the Souffle work flow.

reduction axis (i.e., on which dimension of a tensor will be traversed to perform the reduction), with a reduction index ranging from 0 to 64. The output tensor *O0* is computed using the *compute* operation, which specifies the computation to be performed on each data element and the output shape. The iteration variables *i* and *j* correspond to the output shape dimensions, and their iteration domains can be inferred naturally. Essentially, TE uses a pure functional language [47] to describe tensor computation, allowing for individual computation of each output tensor element. Note that our optimizations also apply to other DSLs like antares [34] and tensor comprehension [48] with similar concise semantics

and expressiveness. We choose TVM due to its popularity and the established toolchain.

4 Overview of Souffle

Souffle is our open-source framework for DNN code optimization. It is designed to overcome the three limitations of existing DNN inference frameworks identified in Section 2. It enhances data reuse, optimizes reduction operations, and enables cross operator boundary optimization. Currently, it supports TensorFlow models and optimizes DNN inference on a single NVIDIA GPU. But many of our analyses and optimizations can be applied to AMD GPU and other accelerators.

Fig. 2 shows an overview workflow of Souffle, which takes a model as input and uses TVM to lower the model down to TE on which we perform analysis and optimization.

TE lowering. For a DNN model, Souffle first lowers each operator to its corresponding TEs to form a TE program. Fig. 2 shows that the five operators are lowered to five TEs marked with *TE0* to *TE4*.

Global computation graph analysis. The lowered TE program is passed to the Souffle analysis module. Souffle performs a two-level analysis on the TE program. At the tensor level, Souffle extracts important tensor information like the shapes, live range and computation intensity of an TE. At the element-wise level, Souffle analyzes the fine-grained dependencies between the output and input tensors of each TE, as described in Sec.5. Fig. 2 shows the analytical results including element-wise data dependency and computational complexity for the five TEs. At the tensor level, it finds that *O0* is accessed by *TE1* and *TE3*, which reveals the data reuse opportunity across multiple TEs.

Resource aware program partitioning. Souffle analyzes the tensor dependency graph and uses Ansor [57] to schedule compute-intensive TEs. It partitions the input TE program into multiple subprograms based on resource usage and transforms each subprogram into a computation kernel. For example, in Fig. 2, if the number of blocks of *TE4* exceeds the max blocks per wave limit, Souffle partitions the TE program into two subprograms. The first subprogram includes *TE0*, *TE1*, *TE2*, and *TE3*, while the second includes *TE4*.

TE transformation. The subprograms together with the data-flow analysis and tensor information are sent to the TE transformation module to generate semantic preserving but optimized TEs. In Fig. 2, the computation of *TE1* and *TE3* is scheduled to the inner loop of *TE0* and *TE2* respectively. TE schedule and transformation are explained in Sec.6.

Joint optimization and code generation. The transformed TE subprograms are fed to a scheduler optimizer (Ansor [57] in our case) to generate a schedule for the TE subprogram. Next, Souffle composes schedules within a subprogram into one single function represented by TensorIR [15] for joint optimizations of instructions and data reuse within the subprogram. Finally, the optimized subprogram is passed to the back-end code generator to produce CUDA kernels. In Fig. 2, *ldg2s* stands for load from global memory to shared memory, *wmma_16x16* stands for warp matrix multiply-accumulate, and *sts2g* stands for storing shared memory to global. Souffle wraps the TE's corresponding code in *if* statement to match the launch dimensions and inserts global sync primitives (*grid.sync()* in this example) to synchronize data across thread blocks. *SO0* is cached in shared memory and reused across operator boundaries (TE1 and TE3 in this working example). We describe these procedures in Sec. 6.5.

5 Global Computation Graph Analysis

Souffle applies two levels of analysis on the TE's tensor dependency graph. The first identifies data reuse opportunities and the second extracts the element-wise data dependence.

5.1 Identifying data reuse opportunities

Tensors are often reused both in temporal and spatial dimensions, providing opportunities for exploiting data reuse to eliminate expensive memory operations. As discussed in Section 2.3, there are two types of tensor reuse in our working example shown in Fig. 1(a) and Fig. 1(b). First, the three *GEMM0* operators share the same input tensor which can be reused spatially. Fusing the three *GEMM0* operators into one kernel would allow us to load the input once and reuse it three times across *GEMM0* operators. Such a reuse opportunity is common in DNNs, including recurrent neural networks [44], convolution neural networks [29, 45, 55] and Transformer models [31, 49]. Spatial data reuse optimizations apply to tensors that are consumed by operators that have no data dependencies, and the operators will be horizontally transformed as described in Sec. 6.1. The second type of reuse opportunities can manifest in the temporal dimension. Temporal data reuse opportunities apply to tensors that are used more than once by operators that have data dependencies, and guide the tensor reuse optimization which is described in Sec. 6.5. Consider again our working example in Fig. 1, the result of *element-wise arithmetic operator 1* (termed as *A1*) is used by two dependent operators *R1* and *A2*. Once again, accesses to the global memory can be eliminated if we cache the computation output of *A1* on register/shared memory.

Souffle identifies these data reuse opportunities from the TE tensor dependency graph at the tensor level by first traversing the computation graph to gather all the tensors accessed by more than one TE. It records the set of operators, $s(t_i) = \{op_j, ..., op_k\}$, that shares with tensor t_i to enable code optimizations, as described in Sections 6.

5.2 Intra-TE element-wise dependency analysis

Souffle captures element-wise data dependence from output to input tensors within a TE by defining the iteration space as the output shape, and the data space as the domain of iteration and reduction variables for each input tensor. This simplifies the element-wise dataflow from input to output tensors, as we only need to record the relevant input tensor elements for a element of the output tensor. The information also enables reduction operator fusion at the TE transformation stage, which other optimization tools such as TensorRT and Apollo do not support.

Our key observation is that the intra-TE data dependence falls into two categories. Firstly, for TE *without* a reduction axis (see also Section 3), each output tensor element relies on only one input tensor element (termed as *one-relies-on-one*). Secondly, for TE with a reduction axis, each output element relies on all the elements of all the reduction dimensions of input tensors (termed as *one-relies-on-many*). With this observation, we can greatly simplify the dependence analysis process compared to the source code or operator-level analysis that other kernel fusion approaches rely on.

We use the polyhedral model notation [46] to denote element-wise dependencies from output tensor element to input tensor element(s). Each tensor has an associated *Set* $S = [x_0, \ldots, x_n : c_0 \wedge \ldots c_m]$ representing its data space, with x_i as iteration variables and c_j as loop bounds from TEs. *Relation* signifies output elements depending on input tensor elements. A pair of output and input tensors is tied to a relation $R = \{[x_0, \ldots, x_n] \mapsto [y_0, \ldots, y_m] : c_0, \ldots, c_p\}$. For TEs in Fig 2, *TE1* gives $R_1 = \{O1[i, j] \mapsto 0[i, j], 0 \leq i < 64, 0 \leq j < 64\}$ and *TE0* results in $R_0 = \{O0[i, j] \mapsto I0[i, rk], 0 \leq i < 64, 0 \leq j < 64, 0 \leq rk < 64\}$. *TE1* is of type *one-relies-on-one* and *TE0* is of type *one-relies-on-many*.

One-relies-on-one TEs. Souffle adopt quasi-affine maps [7, 36] to represent element-wise dependency for an *one-relies-on-one* TE. The mapping from output to input can be expressed in the form $M\overrightarrow{v} + \overrightarrow{c}$ where $\overrightarrow{v}$ is the indices of output tensor, M is a constant matrix from $\mathbb{Z}^{n \times m}$ and $\overrightarrow{c}$ is a constant vector from $\mathbb{Z}^m$. Here, n is the output tensor's dimension and m is the corresponding input tensor's dimension. Note that multiple indices of the output tensor may rely on the same index of the input tensor. For instance, relation R_1 can be represented as:

$$\begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 & 0 \end{bmatrix}, 0 \leq i < 64, 0 \leq j < 64 \qquad (1)$$

One-relies-on-many TEs. For a one-relies-on-many TE, Souffle extracts the region of input tensor accessed by combining the iteration space and the input tensor's index function. As the iteration domain of reduction axes is a constant value, the mapping can be expressed in the form of $R = \{[x_0, \ldots, x_n] \mapsto \{[y_0, \ldots, y_m], [r_0, \ldots, r_s]\} : c_0, \ldots, c_p\}$, where $[r_0, \ldots, r_s]$ is a set of reduction variables and their ranges. For instance, relation R_0 can be expressed as follows: $R_0 = \{O0[i, j] \mapsto \{I0[i, rk], [0 \leq rk < 64]\}, 0 \leq i < 64, 0 \leq j < 64\}$, where $\{I0[i, rk], [0 \leq rk < 64]\}$ represents a set of elements with rk ranging from 0 to 64. We stress that the element-wise dependency for compute-intensive operators like GEMM and convolution can be easily analyzed as the tensor expression explicitly gives the reduction axes.

TEs with *one-relies-on-one* dependency are then transformed in Sec 6.2, and TEs with *one-relies-on-many* dependency are then scheduled in Sec 6.3 and Sec 6.4.

5.3 TE characterization

Souffle classifies a TE as memory-intensive (e.g., *reduce_sum*) or computation-intensive (e.g., *GEMM*) by estimating the compute-memory ratio for a TE. The ratio is computed by dividing the number of arithmetic instructions by the number of memory accesses. As a result, this ratio unit represents the number of arithmetic operations per tensor element that is both read and written. In this work, the classification threshold is empirically set to 3. A ratio less than the threshold indicates that the TE is memory-intensive.

5.4 TE Program Partitioning

Souffle tries to generate large kernels to maximize data reuse and eliminate extra kernel launches. However, using global synchronization imposes a constraint that the thread block count cannot exceed the maximum block count per wave. If this constraint cannot be satisfied, Souffle partitions the TE program into multiple TE subprograms. In Souffle, a TE subprogram serves as the fundamental unit for high-level TE transformation, middle-end schedule optimization, and back-end code generation. It can include several operators mapped to one GPU kernel.

Selection of partitioning point. We only consider compute-intensive operators as candidate partitioning points. Compute-intensive TEs typically use much more shared memory and registers than memory-intensive TEs. Excessive use of shared memory and registers pushes the occupancy up, thus limiting the max blocks per wave and making it infeasible for global synchronization. Souffle transforms memory-intensive TEs and uses their compute-intensive producer TE's schedule to achieve better data reuse (Sec. 6).

Get required resource. Souffle gets the kernel launch dimension and the register/shared memory occupancy from the TE schedule produced by the schedule optimizer (Ansor in this work).

Partitioning algorithm. Souffle ensures resource constraint being satisfied in TE program partitioning using an analytical model. Given a GPU with a total of C register/shared memory, Souffle extracts the maximal launch dimension max_{grid} and the maximal occupancy of register/shared memory max_{occ} for all compute-intensive TEs in

```
# shape A1:(4,8),B1:(8, 16),A2:(2, 8),B2:(8, 16)
rk = te.reduce_axis((0, 8), name="rk")
C1 = te.compute((4,16), lambda i,j:te.sum(A1[i,rk]*B1[rk,j],axis=[rk]))
C2 = te.compute((2,16), lambda i,j:te.sum(A2[i,rk]*B2[rk,j],axis=[rk]))
```

Horizontal transformation

```
C = te.compute((4+2, 16), lambda i, j:
  te.sum(tir.if_then_else(i<4, A1[i, rk], A2[i, rk]) *
    tir.if_then_else(i<4, B1[rk, j], B2[rk, j]), axis=[rk]))
```

Figure 3. Horizontal transformation for two *GEMM*.

```
A = te.placeholder((4, 8))                                  Before
B = te.compute((4,8),lambda i,j:                            transformation
  tir.if_then_else(A[i,j]>0, A[i,j], 0))  # Relu
C = te.compute((2,4),lambda i,j:B[2*i,j]) # Strided_slice
D = te.compute((4,2), lambda i,j:C[j,i])  # Permute
```

Vertical transformation

```
# Semantic preserving TE                                    After
D = te.compute((4,2), lambda i,j:                           transformation
  tir.if_then_else(A[j, 2*i]>0, A[j,2*i], 0))
```

Figure 4. Example of vertical TE transformation.

the current TE subprogram. It then checks whether the constraint $max_{grid} * max_{occ} < C$ can be satisfied for all selected TEs within a subprogram. Souffle uses a greedy algorithm to partition the TE program, starting with an empty S_j and using Breadth First Search (BFS) to add TE te_i to S_j. If adding te_i to S_j violates the constraint, Souffle creates a new subprogram S_{j+1} by adding this TE to the new sub-program and repeats the process until all TEs have been allocated to a subprogram.

6 Semantic-preserving TE Transformations

After Souffle has collected the reuse and dependence information as described in Section 5, it then looks at opportunities to automatically transform the TEs to improve the performance within each TE subprogram. Souffle supports both horizontal and vertical TE transformations and transforms TEs in the same subprograms. Horizontal fusion fuses branches of operators into a single kernel [33, 52]. Horizontal transformation in Souffle is similar to horizontal fusion and is applied to branches of *independent* TEs. Vertical transformation is similar to vertical fusion [37, 58] and is applied to multiple *consecutive* TEs with a *one-relies-on-one* data dependence. We stress that our horizontal and vertical transformations are more flexible than the fusion strategies used by IREE and Rammer [33], which will not fuse operators with one-relies-on-many operators into one kernel. Furthermore, semantic-preserving transformation ensures the preservation of arithmetic operations(e.g. *add*, *exp*) while satisfying data dependence requirements. In contrast, some transformations used by other DNN optimization approaches may not preserve the semantics. For example, TASO [20] optimizes the DNN graph by subgraph substitution and might replace *add* with a *concat+convolution*.

6.1 Horizontal transformation for independent TEs

Souffle tries to transform multiple independent TEs to a single TE to increase parallelism. Souffle first compares the output tensor's shape for each independent TEs and tries to concatenate them as a single TE. Souffle concatenates output tensors from multiple independent TEs to one as each TE can only produce one output tensor. Souffle adds predicates based on the region of output and rewrites the TE. Subsequently, Souffle then assesses whether these TEs consume the same input tensor. Notably, the opportunity for tensor reuse, as discussed in Sec 5.1, is examined. Consequently, the shared tensor only needs to be loaded once within the generated kernel. Therefore both the number of kernel and global memory transactions can be reduced. Figure 3 gives an example of the horizontal TE transformation, both TEs share the same reduction variable, rk. The output of the first and the second TEs are $(4, 16)$ and $(2, 16)$ and can be concatenated on the second axis to a single tensor with shape $(6, 16)$. If the outputs of independent TEs can not be concatenated, it adds an *if_else* statement to select the corresponding input tensor for concatenated TEs, similar to Rammer [33].

6.2 Vertical transformation for one-relies-on-one TEs

Souffle vertically transforms TEs with *one-relies-on-one* data dependence to one TE to reduce the generated kernels and reuse data on registers. This is enabled by the quasi-affine maps representation (Sec 5.2). To this end, Souffle first transforms all *one-relies-on-one* TEs by applying the index mapping function from the child TEs to their parent TEs. It then applies the index mapping functions and creates a single semantic preserving TE. Assume we have n TEs, te_0, $te_1, \cdots, te_i, \cdots, te_{n-1}$, where the output of te_i is the input of te_{i+1}. The mapping function can be expressed as $f_i(\overrightarrow{v_i})$ for te_i. The transformed TE's mapping function from te_{i+1} to te_i can then be computed using the following function:

$$f_{i+1,i}(\overrightarrow{v_i}) = f_{i+1}(f_i(\overrightarrow{v_i})) = M_{i+1} \times (M_i + \overrightarrow{c_i}) + \overrightarrow{c_{i+1}} \quad (2)$$

For the example in Figure 4, the index mapping function of three TEs, A, B and C, can be converted to a single mathematically equivalent function - effectively reducing the number of TE by 3x - as:

$$\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}\begin{bmatrix}2 & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}i\\ j\end{bmatrix} \Rightarrow \begin{bmatrix}0 & 1\\ 2 & 0\end{bmatrix}\begin{bmatrix}i\\ j\end{bmatrix}$$

Using the method described above, Souffle iteratively refines multiple *one-relies-on-one* TEs from a set of consecutive TEs until no further possible refinements can be found. It then applies a schedule from it's compute-intensive parent TE to attach the memory-intensive *one-relies-on-one* TEs to compute-intensive TE, described in the next subsection. Compared to the hand-crafted transformation rules used by TesorRT, Apollo and Ansor, our semantic-preserving transformation has a better generalization ability.

6.3 Schedule TEs

Souffle uses Ansor to generate optimized schedules for compute- and memory-intensive TEs. Note we have already generated a schedule for compute-intensive TEs during TE program partitioning in Sec. 5.4. It propagates the compute-intensive producer's schedule for memory-intensive TEs to maximize data reuse. For *one-relies-on-one* TEs, Souffle first schedules them based on their compute-intensive TEs tile size, then safely inlines them with their producer's compute statement. For *one-relies-on-many* TEs, Souffle reduces locally to reuse the producer's data on shared memory/register, then uses atomic primitives to reduce across thread blocks.

6.4 Merging TEs Schedule

After scheduling TEs, Souffle merges the schedules of TEs within a subprogram to create a holistic function using TensorIR [15]. It adds predicates if the launch dimension of TEs differs and inserts global sync primitives between TEs with *one-relies-on-many* dependency. Finally, it performs several optimizations described in the next section.

6.5 Optimizations within a Subprogram

Souffle supports two types of optimizations within a TE subprogram. The first is instruction-level optimization aiming to overlap asynchronous GPU memory loads with arithmetic instructions. Note that this pipeline execution is scheduled across TEs and without global data dependency analysis the optimization can not be done. The second is to reuse tensor buffers across TEs (and potentially across multiple original operators). Souffle performs subprogram-level optimization after TE schedules within a subprogram have been generated by the underlying scheduler (Ansor in this work). It utilizes the global computation dependency analysis results (Sec. 5) to apply the two optimizations.

Instruction-level optimization. Souffle regroups instructions within a fused subprogram containing multiple original operators to execute memory and arithmetic instructions in parallel. This is accomplished by the scheduling load/store and computation instructions for pipeline execution across operator boundaries. For instance, in the BERT model discussed in Section 2.1 and Figure 1(d), the Souffle-generated schedule issues NVIDIA instructions *LDGSTS.E.BYPASS.128* and *HMMA.16818.F16* in parallel, where the former instruction copies 128 bits from the GPU global memory to shared memory, and the latter computes GEMM on NVIDIA tensorcores.

Tensor reuse optimization. Souffle maximizes tensor buffer reuse across TEs with a simple software-managed cache, using a Least Recently Used (LRU) policy to replace tensor buffers (e.g., matrices and vectors) from shared memory at runtime. It scans instructions linearly until shared memory is exhausted, spilling the shared memory to global memory and adding a memory barrier if the shared memory is exhausted.

Algorithm 1: Semantic-preserving TE transformation

Input: TE program P, analysis results OR, MR, MI, CI, SR, TR
Output: Generated Schedule sch

```
1  sch = {}; SP = ∅;
2  for e in BFS(P) do
3      if e in CI then
4          sch[e] = auto_schedule(e);
5          if resource(sch[e])>C then
6              (SP_i, P) = split(P, e); SP = SP ∪ SP_i ;
7          end
8      end
9  end
10 for SP_i in SP do
11     SP_i = horizon_trans(e, SP_i) for e in SP_i and e in
           ST;
12     SP_i = verti_trans(e, SP_i) for e in SP_i and e in OR;
13     for e in SP_i and e in CI do
14         for e_i in dominated_by(e) and e_i in MI do
15             sch[(e, e_i)] =
                 propagate_sch(e_i, sch[e], SP_i);
16             mark e_i has been scheduled;
17         end
18         sch[SP_i] = sch[SP_i] ∪sch[(e, e_i)];
19     end
20     for e in TR do
21         sch[SP_i] = optimize_tensor_reuse(e,
             sch[SP_i]);
22     end
23 end
```

6.6 Put it all together

Algorithm 1 outlines TE transformation. It takes TE program and the corresponding analysis results: OR (one-relies-one-one TEs), MR (one-relies-on-many TEs), MI (memory-intensive TEs), CI (compute-intensive TEs), TR (temporal reuse tensor and TE tuples), and SR (spatial reuse tensor tuples). It partitions TEs based on compute-intensive TEs' schedule against resource limits (lines 2-9). Then, it horizontally transforms and optimizes spatial reuse through vertical transformation within each partition (lines 11-12). It propagates compute-intensive TEs' schedules to memory-intensive TEs and merges schedules within each sub-program (lines 13-18). Finally, it optimizes tensor reuse through temporal data reuse and across original operator boundaries (lines 19-21).

Table 2. DNN models and datasets used in our evaluation.

Model (Dataset)	Model parameters
ResNeXt (ImageNet)	#layers:101, bottleneck width: 64d
EfficientNet (ImageNet)	Efficient-b0 from the source publication
Swin-Trans. (ImageNet)	Base version, patch: 4 and window size: 7
BERT (SQuAD)	Base version with 12 layers from TensorRT
LSTM (synthetic)	input length: 100, hidden size: 256, layer: 10
MMoE (synthetic)	We use the base model from [32]

Table 3. End-to-end model runtime (*ms*) - lower is better.

Model	XLA	Ansor	TRT	Rammer	Apollo	IREE	Ours
BERT	2.55	2.31	1.30	2.19	3.29	2.22	**1.22**
ResNeXt	8.91	20.50	24.82	11.69	22.80	314.8	**4.43**
LSTM	10.57	6.78	6.30	1.72	Failed	16.0	**0.80**
EfficientNet	2.96	0.91	1.21	Failed	2.3	12.33	**0.66**
SwinTrans.	6.43	5.81	1.74	Failed	10.78	18.1	**1.55**
MMoE	0.29	0.034	0.070	Failed	0.049	0.088	**0.014**

6.7 Implementation Details

We implemented Souffle with 10K lines of C++ and 1K lines of Python code. We integrated Souffle with TVM [12] using Ansor [57] as its schedule optimizer. However, Souffle can work with other schedulers compatible with TEs. Souffle supports element-wise operators, broadcasts, reductions (including *reduce_sum*, *GEMM* and *Conv*), reorganized operators like *reshape* and shuffle operators like *transpose*. Souffle does not support non linear algebra operators like *TopK* or *Conditional*. We use direct convolution which is the default implementation of Ansor.

7 Experimental Setup

7.1 Evaluation Platform and Workloads

Platform. Our evaluation platform is a GPU server with a dual-socket 20-core, 2.50 GHz Xeon Gold 6248 CPU, 768GB of DDR4 RAM, and a 40GB NVIDIA A100 GPU. The server runs Ubuntu 18.04.5 with Linux kernel 5.4.55. We use CUDA version 11.7 with "-O3" as the compiler option.

DNN workloads. We evaluated Souffle on representative and diverse DNN workloads in Table 2. These include natural language processing (BERT [14]), computer vision (Swin-transformer [31] - Swin-trans. for short) and knowledge discovery (MMoE [32]) that implements the latest mixture-of-expert DNN. These also include classic convolutional and recurrent networks like ResNeXt [55] and LSTM [18]. We use single-precision floating-point (FP32) for all operators, except for GEMM for which we use half-precision floating-point (FP16) to use the tensor cores. we target model inference and set the batch size to one.

7.2 Competing Baselines

We compare Souffle against six strong baselines:

XLA (Tensorflow v2.10). The TensorFlow XLA compiler can fuse DNN operators like point-wise and reduction operators and performs optimizations on the fused operator. Unlike Souffle that performs analysis and optimizations on TEs, XLA performs analysis on its high-level operators(HLO).

TensorRT (v8.2). This GPU-vendor-specific framework optimizes the inference of DNNs on NVIDIA GPUs [38].

Ansor(TVM v0.8). This state-of-the-art DNN optimizer builds upon TVM. It uses auto-tuning techniques to find good tensor schedules from hand-crafted templates.

Rammer (v0.4). This DNN compiler is also known as NNFusion [33]. It generates a spatial-temporal schedule at compile time to minimize scheduling overhead and exploit hardware parallelism through inter- and intra-operator co-scheduling.

Apollo. This represents the state-of-the-art fusion framework for inference optimization [56]. Apollo considers both memory- and compute-bound tensor operators for kernel fusions and uses hand-crafted rules to exploit parallelism between independent tensor operators.

IREE(released on 30 Dec. 2022). The intermediate representation execution environment (IREE) builds upon the LLVM MLIR project [1, 30]. IREE is designed to lower DNN models to MLIR dialects to optimize model inference. IREE utilizes the *linalg* dialect to perform the operator fusion, which supports loop affine fusion optimization and global analysis.

7.3 Performance Report

We use NVIDIA Nsight Compute to profile DNN model's computation latency and record performance metrics. We found that the variance across different runs is less than 2% and only reports the *geometric mean*.

8 Experimental Results

In this section, we first present the overall results of Souffle and the competing approaches, showing that Souffle outperforms all other schemes across evaluated DNN models (Section 8.1). We then quantify the contribution of individual optimizations to the performance improvement for each DNN workload (Section 8.2 and 8.3), compare Souffle with alternative schemes on selected workloads (Section 8.4), and discuss the negligible compilation overhead introduced by Souffle (Section 8.5).

8.1 Overall Performance

Table 3 gives the end-to-end execution time (in ms) of each DNN model running on an A100 GPU. Note that some compilers failed to compile and execute certain DNNs. Overall, Souffle outperforms competing methods across all DNNs. Souffle builds upon TVM's Ansor, but it can significantly

boost the performance of the native TVM + Ansor implementation, giving an average speedup of 3.9× (up to 8.5×) over Ansor. Furthermore, it improves NVIDIA-tuned TensorRT by 3.7× on average (up to 7.9×), with a similar performance improvement over Rammer, Apollo, and IREE. The results demonstrate that Souffle delivers consistent and robust performance improvement across DNN workloads.

Kernel or operator fusion techniques such as XLA, Rammer, and Apollo can surpass Ansor in certain scenarios, highlighting the importance of kernel fusion. However, these approaches can only merge a limited set of operators and lack efficient instruction-level optimizations across some operators, resulting in redundant computations.

Rammer relies on hand-crafted rules for operator fusion and can only merge sibling operators in the computation graph. It does not perform element-wise data dependence analysis or reuse tensor buffers, limiting its ability to optimize operators with shared input-output buffers. Similarly, XLA maps some computation-intensive operators (e.g., GEMM) to a BLAS library call and cannot merge such operators with others. XLA relies on hand-crafted rules for operator fusion and cannot optimize some computation patterns, such as merging two consecutive reduction operators in the BERT model.

Apollo also relies on loop fusion rules and can only merge two reductions with the same tile size. Moreover, it does not support schedules with global synchronization, further restricting a scheduler's optimization. IREE only fuses producer-consumer types of fusions with parametric tile-and-fuse optimizations. By contrast, the horizontal and vertical transformations supported by Souffle are more flexible and can fuse operator patterns unsupported by IREE. As such, IREE cannot fuse computation-intensive operators (e.g., *batch_matmul*) to reduce GPU global memory accesses.

Compared to other kernel fusion techniques, Souffle can identify more data reuse opportunities by operating on TEs, which have simple and well-defined semantics and do not rely on inflexible fusion rules. Souffle can utilize data reuse across operators with different-shaped buffers and perform instruction-level optimizations for unfused operators. Souffle outperforms competing baselines due to these advantages.

8.2 Performance Breakdown

We conducted a series of experiments to evaluate the performance benefits of Souffle's optimizations. We gradually activated our optimizations, starting from the TVM + Ansor generated code (V0) and then adding our *TE horizontal trans.* (V1), *TE vertical trans.* (V2), *global sync* with global synchronization API (V3), and *subprogram-level optimization* (V4), as described in Sec. 6.1, 6.2, 6.4, and 6.5, respectively.

Table 4 reports the impact of individual optimizations on inference time reduction for each DNN model. Our horizontal and vertical TE transformation schemes benefit all DNN workloads, increasing SIMD parallelism and reducing memory accesses. Transformer-based BERT and Swin-Trans. also benefit from global sync and subprogram-level optimization, which enable overlapping load and tensor core's arithmetic instructions and tensor buffer reuse.

Table 4. Execution time (*ms*) with Souffle individual optimizations

Model	V0	V1	V2	V3	V4
BERT	3.1	2.12	1.53	1.41	1.22
ResNeXt	29.0	5.90	4.43	4.43	4.43
LSTM	6.78	1.60	1.21	0.8	0.8
EfficientNet	4.2	0.91	0.72	0.63	0.63
Swin-Trans.	5.81	4.88	2.09	1.78	1.55
MMoE	0.05	0.019	0.016	0.014	0.014

Table 5. The number of GPU kernel calls and global memory data transfer size (*M* bytes) of the resulting code.

Model	# of kernel calls				Memory transfer size		
	TRT	Apollo	XLA	Ours	TRT	Apollo	Ours
BERT	120	240	216	24	361.8	880.5	226.8
ResNeXt	2406	1226	526	105	622.2	436.1	470.2
LSTM	662	Failed	3363	1	126.8	Failed	10.6
Efficient.	187	273	332	66	96.4	127.4	86.6
Swin-Tran.	716	1014	3188	53	831.5	1309.0	282.9
MMoE	20	10	7	1	0.061	0.063	0.058

(a) Unfused **(b)** Fused **(c)** Global-sync **(d)** Data-reuse

Figure 5. Example of fusion results for sub-module EfficientNet.

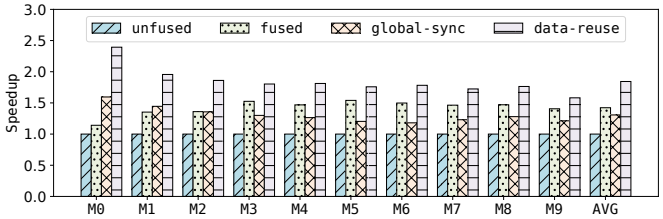

Figure 6. EfficientNet sub-module latency breakdown.

8.3 Analysis of Performance Advantages

We identified two reasons for Souffle's improved performance over TensorRT and Apollo: reduced GPU kernel calls and reduced GPU memory data transfers. We use a micro-benchmark taken from EfficientNet to illustrate the performance contribution of the two optimizations. The sub-module is the building block of EfficientNet and repeats many times with different input sizes (marked with M0 to M9). The pattern of this sub-module is common in many DNN models

and existing DNN frameworks fail to optimize it optimally. Fig. 5 shows four versions: 5a unfused with generating each TE to one kernel, 5b fused with Ansor's fusion, 5c SOUFFLE's global-sync with generating the whole sub-module to one kernel but without any data reuse; 5d with SOUFFLE's data reuse. Fig. 6 shows the normalized speedup of the four versions with the horizontal axis being the different sub-modules. Global sync can achieve 1.31× speedup on average compared with Ansor's unfused, with performance improvements coming from kernel calls reduction and lightweight CUDA grid sync. Enabling data reuse further improves the speedup from 1.31× to 1.84× on average. SOUFFLE's reduced GPU kernel calls and increased data reuse can both significantly improve the performance. However, it's non-trivial to separate the performance contribution for end-to-end models, as TE transformation and global synchronization may both reduce kernel calls and reduce memory access. We report the reduced GPU kernel call and GPU memory data transfers in the following.

Reduce GPU kernel call. GPU kernel calls can be expensive, and it takes around 2 *us* to launch a kernel on an NVIDIA A100 GPU. Table 5 compares the number of kernel calls from TensorRT, Apollo, XLA and SOUFFLE. SOUFFLE can create large subprograms that result in fewer kernels because of resource-aware TE program partitioning. This optimization reduces the kernel launch overhead. For example, in BERT, SOUFFLE reduces the number of kernels from 120 and 240 (generated by TensorRT and Apollo, respectively) to 24. Similar kernel call reductions are observed in other DNN workloads. Operator fusion is one of the key features of XLA. Nonetheless, XLA leverages libraries such as cuBLAS to execute compute-intensive operators. Consequently, it faces limitations in fusing compute-intensive operators with memory-intensive counterparts, thereby hindering the potential reduction in kernel count. For instance, XLA generates 6 custom calls to invoke cuBLAS to run the GEMM operators for one BERT layer. While SOUFFLE seamlessly propagates the schedule of compute-intensive TEs to memory-intensive TEs and generates one kernel.

Reduce GPU memory data transfers. GPU global memory data transfer is known to be expensive and it is desired to reduce the amount of data transfers from the global memory. To do so, SOUFFLE maximizes tensor buffer reuse through TE program partitioning (Sec. 5.4) and TE transformation (Sec. 6). Table 5 also compares the amount of GPU global memory data transfers measured by Nsight Compute for TensorRT, Apollo, and SOUFFLE. SOUFFLE-generated code incurs significantly fewer data transfers compared to TensorRT and Apollo. For example, in BERT, SOUFFLE reduces the memory transaction from 361.8M and 880.5M bytes (loaded by TensorRT and Apollo, respectively) to 226.8M bytes.

Consider the performance of TensorRT and SOUFFLE again when optimizing BERT. Like Sec. 2, we classify the computation kernels in BERT into compute- (like GEMM) and memory-intensive kernels (like *softmax*). We then measure the execution latency (in clock cycles) of each kernel. SOUFFLE is more flexible in fusing operators, which reduces the number of kernels and kernel invocation overhead compared to TensorRT. For example, TensorRT maps a BERT layer to 10 kernels, while SOUFFLE can partition one layer into two kernels and perform instruction-level optimization. SOUFFLE reduces the memory-intensive kernel latency from 31.0 *us* (in TensorRT) to 25.5 *us* by buffering intermediate results in fast memory and GPU registers for BERT one layer.

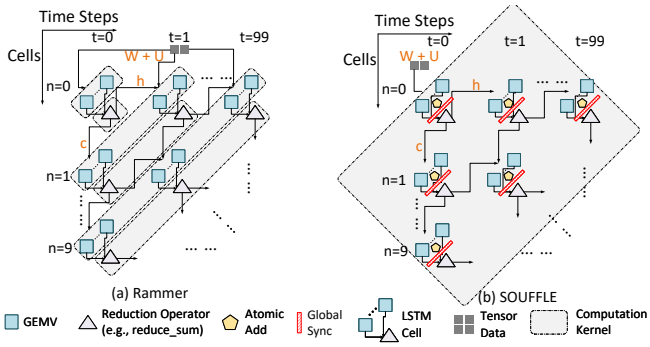

Figure 7. How Rammer (a) and SOUFFLE (b) map a LSTM graph into computation kernels.

We also examine IREE's fusion performance on BERT. IREE misses two optimization opportunities for BERT: it does not fuse GEMM and *softmax* operators and several GEMM operators. IREE launches 180 kernels and takes 2.22 ms for execution. In comparison, SOUFFLE launches 24 kernels and takes 1.22 ms.

8.4 Case Study on LSTM

Following the discussion in Sec. 8.3, we conducted studies on the LSTM model to reveal new optimization opportunities offered by SOUFFLE, which achieved a performance improvement of 4.3× over TensorRT and 2.2× over Rammer. We compared SOUFFLE with Rammer, the most performant baseline, as discussed in Sec. 8.3. Fig. 7 shows the fusion strategy used by Rammer and SOUFFLE for an LSTM with 10 cells (listed vertically in Fig. 7). Each cell has its dedicated weight tensors (marked as W and U in Fig. 7), hidden states (h) and output (c). In each time step t, the n-th cell performs general matrix-vector multiplication (*GEMV* for short) using its weight tensors (W_n and U_n), hidden state (h_n) and output (c_{n-1}) from $(n-1)$-th cell, updates its hidden state (h_n) and generates output (c_n) for the current time step. Fig. 7 shows the fully unrolled time step loop. The LSTM operators alongside the diagonal line are independent, i.e. no data dependency exists. Both SOUFFLE and Rammer exploit such optimization opportunity, i.e. the wavefront parallelism, and fuse the *GEMV* computation to different *blocks* of a kernel.

With the TE-based global analysis, SOUFFLE discovers that the weight tensors (W and U) of each LSTM cell are reused

Table 6. GPU performance counter values for LSTM optimized by Rammer and Souffle.

Metrics	Rammer	Souffle
GPU global memory trans. (in bytes)	1911.0MB	21.11MB
Pipeline Utilization (LSU)	20.2%	35.4%
Pipeline Utilization (FMA)	8.0%	19.0%

across all time steps (temporal reuse). It utilizes the global synchronization and generates one kernel for the entire model, as shown in the right part of Fig. 7. On the other hand, the Rammer version needs to load the weight tensors in every wavefront, resulting in a longer execution time compared to Souffle. We measured GPU global memory data transfer and pipeline utilization for the optimized LSTM. As Table 6 show, Souffle-optimized code reduces memory loads by orders of magnitude compared to Rammer's version ($21MB$ vs $1911MB$) and increases pipeline utilization for both the load store unit (LSU) and fused multiply-add unit (FMA).

8.5 Compilation Overhead

Souffle employs Ansor and TVM for schedule search and generation. The compilation overhead of Souffle + Ansor is mainly from the time required for searching the program schedule using native Ansor implementation. The additional overhead introduced by Souffle involves two-level dependence analysis, model splitting, schedule tuning, and global optimization. Our measurements on six evaluated models indicate that Souffle adds up to $63s$ overhead on top of Ansor, which is negligible compared to the hours Ansor requires for schedule search. This overhead can be reduced by using faster optimizer like Roller [60], which is orthogonal of Souffle.

9 Discussion

Expression power of TE. Souffle relies on the expression power of tensor expressions, which currently does not support all DNN operators, e.g., it does not support *resize*. Souffle maps these TE-unsupported operators to a computation kernel and uses the back-end operator library implementation but without fusing them with other operators. Given the active developer community of TVM, we expect this limitation to be addressed by future TVM releases.

Cost model for TE program partitioning. Souffle extracts tensor information by compiling the raw TE program. This can be improved by building a cost model [53] to estimate occupancy from the TE program.

Reusing dynamic-shaped tensors. Certain DNN operators have unknown tensor shapes at compile time. Our current implementation does not support reusing tensors of dynamic shapes. To address this, we can generate multiple versions of a kernel and choose the appropriate one based on shape information available at execution time.

Fusion in DL training. DL compilers like TensorFlow XLA also enable operator fusion in training (forward inference and backward parameter updates). Our TE-based transformation can be integrated into DL compilers to accelerate forward and backward passes during training. However, intermediate tensors must be kept in global memory in DL training for backward gradient-based optimization like Adam [24], restricting operator fusion chances. Our main focus is optimizing model inference after DNN training. Support for TE transformation in DL training is left for future work.

Slowdown. Performance slowdown can occur when Souffle extends the schedule from compute-intensive TEs to memory-intensive reduction TEs (discussed in Sec. 6.3). This introduces synchronization between blocks, potentially hampering parallelism for reduction TEs. A potential remedy is to create a cost model to decide whether fusing these TEs is beneficial.

10 Related work

Loop and kernel fusion. Loop fusion is commonly used to improve the performance of CPU programs [3, 8, 9, 23]. Recent research has also utilized kernel fusion to optimize GPU programs by reducing data traffic to/from off-chip memory[42, 50]. Various domain-specific kernel fusion policies have been proposed for workloads like data center applications [54], mesh computing [6], machine learning workloads [4] and image processing [35]. Souffle leverages loop fusion to optimize DNN inference through compiler transformations, building on these previous research efforts.

Operator fusion in DNN compilers. Operator fusions can enhance performance by improving data reuse and reducing on-chip data traffic. To seek out fusion opportunities, DNNFusion classifies operators and defines rules based on their classification [37]. Astitch [58] and Rammer [33] fuse independent operators to leverage inter-operator parallelism, while Deepcuts [22] uses rules to fuse kernels based on GPU hardware parameters. Apollo [56] adopts a partition-based approach to search for fusion opportunities within sub-graphs. However, these approaches rely on hand-crafted rules with extensive engineering efforts and may miss optimization opportunities, as discussed in Sec. 2. Jeong et al [19] proposed a dynamic programming algorithm to decide whether to pin or fuse an activation map on DNN accelerators with a global buffer. DNNFusion [37] classifies operators based on the element-wise mappings from input to output, but it can not fuse many-to-many with many-to-many operators (like GEMM and Softmax), while Souffle can further reduce the overhead of kernel launch. Furthermore, DNNFusion lacks global analysis of tensor reuse opportunities and may miss the temporal and spatial data reuse opportunities,

which are critical to improve the performance as shown in Sec 8.2. Souffle improves upon previous operator fusion techniques by utilizing control and data-flow analysis on the tensor dependency graph to partition TEs into subprograms. TEs have clear and simple relations, and they can be combined to represent numerous DNN operators. Souffle leverages the well-defined semantics in TEs to perform precise data dependence analysis for instruction scheduling and data reuse optimization. Additionally, Souffle applies semantic preserving transformations to refine TEs. Its optimization capabilities have better generalization ability as TEs can be combined to represent more complex operators.

Global analysis and fusion optimization. TensorFlow XLA [27] and MLIR [43] also conduct global analysis on the input program graph. XLA utilizes profitability analysis on its high-level operations intermediate representation before deciding on tiling and fusion. However, XLA relies on handcrafted heuristics to fuse operators, which can be challenging for high-level operators. For instance, XLA's fusion heuristic cannot fuse two consecutive reduction operators in the BERT model. Moreover, as XLA operates at the operator level and some operators are mapped to low-level library calls, it cannot optimize across libraries. In contrast, Souffle takes a different approach by lowering high-level operators into lower-level tensor expressions (TEs), which have concise semantics. Operating on TEs rather than assuming high-level operators enables Souffle to optimize flexibly across operator boundaries. Unlike XLA, Souffle can merge GEMM and Softmax operators and optimize across reduction operators.

The MLIR *-affine-loop-fusion* pass utilizes a slicing-based method to identify producer-consumer and sibling fusion opportunities. Souffle implements a lightweight, specialized global analysis on TEs, which can be easily integrated into DNN inference engines. Moreover, Souffle offers more optimization opportunities than just fusion. For example, it enables joint optimizations across multiple compute-intensive TEs in a TE subprogram and facilitates horizontal and vertical transformations for sibling fusion.

Optimizing individual operators. Numerous compiler-based approaches exist to optimize individual operators, including TVM [12, 51], XLA [27], Tiramisu [5], and TACO [25]. These compilers often represent operators in high-level forms such as TEs or linear algebra, enabling aggressive optimization without complex analysis through domain-specific knowledge. Souffle is orthogonal to these techniques.

11 Conclusion

We have presented Souffle, a top-down compiler-based approach for improving DNN inference. Souffle identifies optimization opportunities across DNN operators by performing data-flow analysis on the entire tensor dependence graph built from tensor expressions. It groups tensor expressions into subprograms and performs local optimization through semantics-preserving transformations, instruction scheduling, and tensor buffer reuse. We evaluated Souffle on six DNN models using an NVIDIA A100 GPU and compared it to six state-of-the-art DNN optimizing frameworks. Souffle outperformed them with a speedup of up to 7.9× over TensorRT.

Acknowledgments

We thank our shepherd, Vinod Grover, and the anonymous reviewers for their constructive feedback. This work was supported in part by the National Key R&D Program of China under grant agreement 2021ZD0110101, the National Natural Science Foundation of China (NSFC) under grant agreements T2222026, 22003073, 62232015, and 62090024, the Innovation Funding of ICT CAS under grant agreement E361010, a Beijing Nova Program, and the UK Engineering and Physical Sciences Research Council (EPSRC) under grant agreement EP/X018202/1. For the purpose of open access, the authors have applied a Creative Commons Attribution (CCBY) license to any Author Accepted Manuscript version arising from this submission.

References

[1] [n. d.]. IREE: Intermediate Representation Execution Environment. https://github.com/iree-org/iree.

[2] Martín Abadi, Paul Barham, Jianmin Chen, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[3] Aravind Acharya, Uday Bondhugula, and Albert Cohen. 2018. Polyhedral auto-transformation with no integer linear programming. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 529–542.

[4] Arash Ashari, Shirish Tatikonda, Matthias Boehm, Berthold Reinwald, Keith Campbell, John Keenleyside, and P Sadayappan. 2015. On optimizing machine learning workloads via kernel fusion. *ACM SIGPLAN Notices* 50, 8 (2015), 173–182.

[5] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 193–205. https://doi.org/10.1109/CGO.2019.8661197

[6] Carlo Bertolli, Adam Betts, Paul HJ Kelly, Gihan R Mudalige, and Mike B Giles. 2012. Mesh independent loop fusion for unstructured mesh applications. In *Proceedings of the 9th conference on Computing Frontiers*. 43–52.

[7] U. Bondhugula, A. Acharya, and A Cohen. 2016. The Pluto+ Algorithm: A Practical Approach for Parallelization and Locality Optimization of Affine Loop Nests. In *ACM Transactions on Programming Languages and Systems*.

[8] Uday Bondhugula, Oktay Gunluk, Sanjeeb Dash, and Lakshminarayanan Renganarayanan. 2010. A model for fusion and code motion in an automatic parallelizing compiler. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. 343–352.

[9] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 101–113.

[10] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[11] Hongming Chen, Ola Engkvist, Yinhai Wang, Marcus Olivecrona, and Thomas Blaschke. 2018. The rise of deep learning in drug discovery. *Drug Discovery Today* 23, 6 (2018), 1241–1250. https://doi.org/10.1016/j.drudis.2018.01.039

[12] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[13] ONNX Runtime developers. 2021. ONNX Runtime. https://onnxruntime.ai/. Version: x.y.z.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[15] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, and Tianqi Chen. 2023. TensorIR: An Abstraction for Automatic Tensorized Program Optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 804–817. https://doi.org/10.1145/3575693.3576933

[16] Tianfan Fu, Cao Xiao, Cheng Qian, Lucas M. Glass, and Jimeng Sun. 2021. Probabilistic and Dynamic Molecule-Disease Interaction Modeling for Drug Discovery. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining* (Virtual Event, Singapore). 404–414. https://doi.org/10.1145/3447548.3467286

[17] Kim Hazelwood, Sarah Bird, et al. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 620–629. https://doi.org/10.1109/HPCA.2018.00059

[18] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[19] Hyuk-Jin Jeong, JiHwan Yeo, Cheongyo Bahk, and JongHyun Park. 2023. Pin or Fuse? Exploiting Scratchpad Memory to Reduce Off-Chip Data Transfer in DNN Accelerators. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization (CGO 2023)*. Association for Computing Machinery, 224–235. https://doi.org/10.1145/3579990.3580017

[20] Zhihao Jia, Oded Padon, et al. 2019. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. 47–62. https://doi.org/10.1145/3341301.3359630

[21] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. 1–12. https://doi.org/10.1145/3079856.3080246

[22] Wookeun Jung, Thanh Tuan Dao, and Jaejin Lee. 2021. DeepCuts: a deep learning optimization framework for versatile GPU workloads. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 190–205.

[23] Ken Kennedy and Kathryn S McKinley. 1993. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 301–320.

[24] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1412.6980

[25] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. 2017. Taco: A tool to generate tensor algebra kernels. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 943–948.

[26] Anna Larionova, Polina Kazakova, and Nikita Nikitinsky. 2019. Deep Structured Semantic Model for Recommendations in E-commerce. In *Hybrid Artificial Intelligent Systems - 14th International Conference, HAIS 2019, León, Spain, September 4-6, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11734)*. Springer, 85–96. https://doi.org/10.1007/978-3-030-29859-3_8

[27] Chris Leary and Todd Wang. 2017. XLA: TensorFlow, compiled. TesorFlow Dev Summit.

[28] Shih-Chieh Lin, Yunqi Zhang, et al. 2018. The Architectural Implications of Autonomous Driving: Constraints and Acceleration. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018*. ACM, 751–766. https://doi.org/10.1145/3173162.3173191

[29] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. 2018. Progressive Neural Architecture Search. In *Computer Vision – ECCV 2018*. Springer International Publishing, 19–35.

[30] Hsin-I Cindy Liu, Marius Brehler, Mahesh Ravishankar, Nicolas Vasilache, Ben Vanik, and Stella Laurenzo. 2022. TinyIREE: An ML Execution Environment for Embedded Systems From Compilation to Deployment. *IEEE Micro* 42, 5 (sep 2022), 9–16. https://doi.org/10.1109/MM.2022.3178068

[31] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. 2021. Swin Transformer: Hierarchical Vision Transformer using Shifted Windows. *CoRR* abs/2103.14030 (2021). arXiv:2103.14030 https://arxiv.org/abs/2103.14030

[32] Jiaqi Ma, Zhe Zhao, Xinyang Yi, Jilin Chen, Lichan Hong, and Ed H. Chi. 2018. Modeling Task Relationships in Multi-Task Learning with Multi-Gate Mixture-of-Experts. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '18)*. 1930–1939. https://doi.org/10.1145/3219819.3220007

[33] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. [n. d.]. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 881–897. https://www.usenix.org/conference/osdi20/presentation/ma

[34] Microsoft. 2022. Antares. https://github.com/microsoft/antares/tree/latest.

[35] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically Scheduling Halide Image Processing Pipelines. *ACM Trans. Graph.* 35, 4, Article 83 (jul 2016), 11 pages. https://doi.org/10.1145/2897824.2925952

[36] Multi-Level IR Compiler Framework committee. 2022. *'affine' Dialect.*

[37] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. 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 (PLDI 2021)*. 883–898. https://doi.org/10.1145/3453483.3454083

[38] NVIDIA Corporation. 2021. TensorRT. https://developer.nvidia.com/tensorrt.

[39] NVIDIA Corporation. 2022. *NVIDIA Nsight Compute*.

[40] NVIDIA Corporation. 2023. CUDA Grid Synchronization. https://docs.nvidia.com/cuda/cuda-c-programming-guide/#grid-synchronization.

[41] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *CoRR* abs/1912.01703 (2019). arXiv:1912.01703 http://arxiv.org/abs/1912.01703

[42] Bo Qiao, Oliver Reiche, Frank Hannig, and Jïrgen Teich. 2019. From loop fusion to kernel fusion: A domain-specific approach to locality optimization. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 242–253.

[43] Tatiana Shpeisman and Chris Lattner. 2019. Mlir: Multi-level intermediate representation for compiler infrastructure.

[44] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. [n. d.]. Sequence to Sequence Learning with Neural Networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2* (Montreal, Canada) *(NIPS'14)*. MIT Press, 3104–3112.

[45] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A. Alemi. 2017. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*. 4278–4284.

[46] Sanket Tavarageri, Alexander Heinecke, Sasikanth Avancha, Bharat Kaul, Gagandeep Goyal, and Ramakrishna Upadrasta. 2021. PolyDL: Polyhedral Optimizations for Creation of High-performance DL Primitives. *ACM Trans. Archit. Code Optim.* 18, 1 (2021), 11:1–11:27. https://doi.org/10.1145/3433103

[47] Tianqi Chen. 2022. Working with Operators Using Tensor Expression. https://tvm.apache.org/docs/tutorial/tensor_expr_get_started.html.

[48] Nicolas Vasilache, Oleksandr Zinenko, et al. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018). arXiv:1802.04730 http://arxiv.org/abs/1802.04730

[49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *CoRR* abs/1706.03762 (2017). arXiv:1706.03762 http://arxiv.org/abs/1706.03762

[50] Mohamed Wahib and Naoya Maruyama. 2014. Scalable kernel fusion for memory-bound GPU applications. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 191–202.

[51] Huanting Wang, Zhanyong Tang, et al. 2022. Automating Reinforcement Learning Architecture Design for Code Optimization. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction* (Seoul, South Korea) *(CC 2022)*. Association for Computing Machinery, New York, NY, USA, 129–143. https://doi.org/10.1145/3497776.3517769

[52] Shang Wang, Peiming Yang, Yuxuan Zheng, Xin Li, and Gennady Pekhimenko. 2021. Horizontally Fused Training Array: An Effective Hardware Utilization Squeezer for Training Novel Deep Learning Models. In *Proceedings of Machine Learning and Systems 2021*. mlsys.org. https://proceedings.mlsys.org/paper/2021/hash/a97da629b098b75c294dffdc3e463904-Abstract.html

[53] Zheng Wang and Michael O'Boyle. 2018. Machine learning in compiler optimization. *Proc. IEEE* 106, 11 (2018), 1879–1901.

[54] Haicheng Wu, Gregory Diamos, Jin Wang, Srihari Cadambi, Sudhakar Yalamanchili, and Srimat Chakradhar. 2012. Optimizing data warehousing applications for GPUs using kernel fusion/fission. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE, 2433–2442.

[55] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. 2017. Aggregated Residual Transformations for Deep Neural Networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 5987–5995. https://doi.org/10.1109/CVPR.2017.634

[56] Jie Zhao, Xiong Gao, et al. 2022. Apollo: Automatic Partition-based Operator Fusion through Layer by Layer Optimization. In *Proceedings of Machine Learning and Systems 2022*. https://proceedings.mlsys.org/paper/2022/hash/069059b7ef840f0c74a814ec9237b6ec-Abstract.html

[57] Lianmin Zheng, Chengfan Jia, et al. [n. d.]. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation, 2020*. 863–879. https://www.usenix.org/conference/osdi20/presentation/zheng

[58] Zhen Zheng, Xuanda Yang, et al. 2022. AStitch: enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 359–373.

[59] Guorui Zhou, Na Mou, Ying Fan, Qi Pi, Weijie Bian, Chang Zhou, Xiaoqiang Zhu, and Kun Gai. 2018. Deep Interest Evolution Network for Click-Through Rate Prediction. https://doi.org/10.48550/ARXIV.1809.03672

[60] Hongyu Zhu, Ruofan Wu, et al. 2022. ROLLER: Fast and Efficient Tensor Compilation for Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, 233–248. https://www.usenix.org/conference/osdi22/presentation/zhu