2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)

Chimera: An Analytical Optimizing Framework for Effective Compute-intensive Operators Fusion

Size Zheng[1, †], Siyuan Chen[1, †], Peidi Song[1], Renze Chen[1], Xiuhong Li[2, 4], Shengen Yan[2], Dahua Lin[3, 4], Jingwen Leng[5], Yun Liang[1, 6, *]
[1]*Peking University,* [2]*Sensetime Research,* [3]*The Chinese University of Hong Kong,*
[4] *Shanghai AI Lab,* [5]*Shanghai Jiao Tong University,* [6]*Beijing Advanced Innovation Center for Integrated Circuits*
{zhengsz, chensiyuan, ppppaidy, crz, ericlyun}@pku.edu.cn, {lixiuhong, yanshengen}@sensetime.com
dhlin@ie.cuhk.edu.hk, leng-jw@sjtu.edu.cn

***Abstract*—Machine learning models with various tensor operators are becoming ubiquitous in recent years. There are two types of operators in machine learning: compute-intensive operators (e.g., GEMM and convolution) and memory-intensive operators (e.g., ReLU and softmax). In emerging machine learning models, compute-intensive operators are usually organized in a chain structure. With the continual specialization of hardware, the gap between computing performance and memory bandwidth has become more prominent. Consequently, the implementations of many compute-intensive operator chains are bounded by memory bandwidth, and generating fused kernels to improve locality for these compute-intensive operators becomes necessary. But in existing machine learning compilers, there lack both precise analysis and efficient optimization for compute-intensive operator chains on different accelerators. As a result, they usually produce sub-optimal performance for these operator chains.**

In this paper, we propose Chimera, an optimizing framework that can efficiently improve the locality of compute-intensive operator chains on different hardware accelerators. In Chimera, each compute-intensive operator is composed of a series of computation blocks. To generate efficient fused kernels for the operator chains, optimizations for both inter-block and intra-block are required. For inter-block optimization, Chimera decides the optimized block execution order by minimizing the data movement volume among blocks using an analytical model. For intra-block optimization, Chimera uses unified replaceable micro kernels to apply hardware-specific optimizations for different accelerators. Finally, Chimera generates fused kernels for compute-intensive operator chains. Evaluation of batch GEMM chains and convolution chains on CPU, GPU, and NPU shows that Chimera achieves up to $2.87\times$, $2.29\times$, and $2.39\times$ speedups to hand-tuned libraries. Compared to state-of-the-art compilers, the speedups are up to $2.29\times$, $1.64\times$, and $1.14\times$ for CPU, GPU, and NPU.

I. Introduction

Machine learning models that are composed of various tensor operators are becoming ubiquitous [13], [16], [17], [19], [24], [40], [42], [48]. There are two types of tensor operators in current machine learning models: compute-intensive operators (such as GEMM and convolution) that account for most of the computations and memory-intensive operators (such as ReLU and softmax) that are used to connect compute-intensive operators. Many previous libraries [1]–[3], [5]–[7], [28], [38] and compilers [12], [14], [23], [36], [43], [45], [47], [49], [54], [56], [57], [63] are proposed to optimize these operators.

[†]Both authors contribute equally to this work.
[*]Corresponding author.

TABLE I
The compute/memory breakdown of ML models and the performance of different accelerators.

ML Model Breakdown			
Name	**%MI**	**%CI**	**%BMM**
Transformer	19.45%	40.51%	40.04%
Bert-Base	30.56%	42.79%	26.65%
ViT-Huge	15.63%	50.85%	33.52%
Compute and Memory Characteristics of Accelerators			
Device	**Xeon Gold**	**A100**	**Ascend 910**
Dedicated Unit	AVX-512	Tensor Core	Cube Unit
Peak Perf.	12 TFlops	312 TFlops	320 TFlops
Memory BW.	131 GB/s	1555 GB/s	1200 GB/s
Peak Perf/BW	92 Flop/byte	200 Flop/byte	267 Flop/byte

With the continual progress of hardware specialization, the disparity of speed between dedicated compute cores and memory outside the chips becomes increasingly prominent. As a result, many compute-intensive operators are bounded by memory bandwidth. In Table I we show the FP16 peak compute performance and memory bandwidth of several hardware accelerators: Xeon Gold AVX-512 CPU, A100 Tensor Core GPU [4], and Ascend 910 NPU [30]. The high ratio of the peak performance to the memory bandwidth of these accelerators indicates that they require high arithmetic intensity to achieve high performance. For example, to unleash the computing power of Xeon Gold CPU, at least 92 float operations are expected for per byte loaded.

Moreover, the gap between compute performance and memory bandwidth is expected to continue to grow. The memory-bound implementations of many compute-intensive operators (e.g., batch GEMM in Transformer) are becoming a performance bottleneck. We show the execution time breakdown of some emerging models in Table I (sequence length is set to 512). The column *%MI* refers to the ratio of execution time that all the memory-intensive operators account for; the column *%CI* refers to the ratio of compute-intensive operators except for the batch GEMMs in attention layers; and the column *%BMM* refers to the ratio that the batch GEMMs (whose implementations are memory-bound) account for. As shown in the Table, the memory-bound batch GEMMs occupy a large proportion of execution time (26.65% to 40.04%), which exceeds that of other memory-intensive operators.

978-1-6654-7652-2/23/$31.00 ©2023 IEEE

Therefore, optimizations to these memory-bound compute-intensive operators to improve locality and reduce the pressure on memory bandwidth are necessary.

Kernel fusion is an effective optimization for memory-bound operators. However, the compute-intensive operators in emerging models often form a chain structure with strict data dependency and thus generating efficient fused kernels for the compute-intensive operator chains is difficult. First, it is hard to decide the execution order of the computations of compute-intensive operator chains. Each operator in the chain can be decomposed into a series of computation blocks as pointed in previous works [58], [65]. Different execution order of these computation blocks can result in different data movement volume among the blocks and thus the performance will also change drastically. Existing works [23], [36], [57], [63], [65] fail to optimize the execution order for compute-intensive operator chains because they lack a precise performance model to evaluate the data movement volume of different ordering choices of operators. Second, optimizing the computations within each block using hardware-specific features is challenging. There lacks a unified approach for extensible and flexible micro kernel generation for different hardware accelerators. Previous works [36], [54], [63] use fixed micro kernels so they can hardly be generalized to other hardware accelerators.

In this paper, we present *Chimera*, an optimization framework for machine learning models that generates fused kernels for compute-intensive operator chains for high performance. Chimera decomposes compute-intensive operator chains into computation blocks and its optimizations then fall into two aspects: inter-block and intra-block optimization. For inter-block optimization, Chimera optimizes the block execution order by minimizing the data movement volume (maximizing data locality). In detail, Chimera enumerates different block execution orders and analytically estimates the input/output data movement volume among blocks. After that, Chimera selects the execution order that gives the minimal data movement volume so that the optimal data locality is achieved.

Different from previous works [26], [65] that only optimize the block orders within one compute-intensive operator, Chimera's optimization is applicable to multiple compute-intensive operators by considering intermediate result reuse and interleaving of blocks from different operators. For intra-block optimization, Chimera applies hardware-specific optimizations. To handle hardware diversity, Chimera uses a unified replaceable micro kernel as a high-level abstraction and generates low-level micro kernel implementations for different hardware architectures during code generation. Finally, the computation blocks from different compute-intensive operators are interleaved according to the block execution order and low-level device code is generated by using hardware-specific micro kernels. In summary, this paper makes the following contributions:

1) It proposes an analytical model to evaluate the data movement volume of memory-bound compute-intensive operator chains.

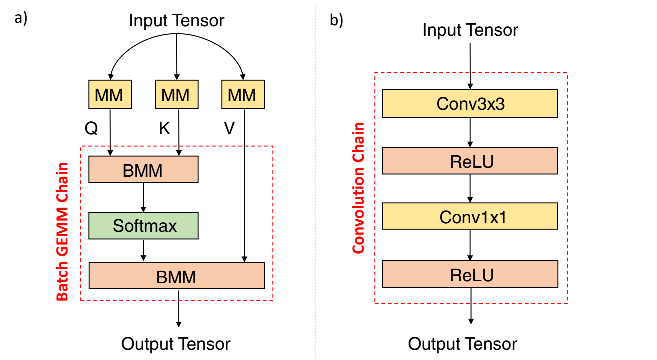


Fig. 1. Typical tensor operators in machine learning. a) batch GEMM chains from Transformers. b) convolution chains from CNNs.

2) It proposes to use replaceable micro kernels for different accelerators and uses an analytical approach to optimize the micro kernels.
3) It achieves better performance than state-of-the-art compilers for different compute-intensive operator chains.

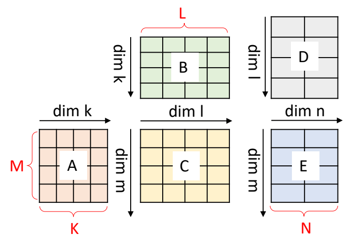
Evaluation of batch GEMM chains and convolution chains on CPU, GPU, and NPU shows that Chimera achieves up to $2.87\times$, $2.29\times$, and $2.39\times$ speedups to hand-tuned libraries [1], [6], [8]. Compared to state-of-the-art compilers [23], [43], [54], [56], [57], the speedups are up to $2.29\times$, $1.64\times$, and $1.14\times$ for CPU, GPU, and NPU.

II. Background and Challenges

In this section, we first introduce several typical tensor operators in machine learning models including GEMM chains from Transformers [48] and convolution chains from CNNs [19], [40]. Then, we explain the major challenges of generating fused kernels for these operator chains.

A. Tensor Operators in Machine Learning

Current machine learning models are usually constructed with multiple tensor operators. Unlike previous models that wrap some memory-intensive element-wise or reduce operators around one compute-intensive operator such as GEMM and convolution, recent models tend to assemble multiple compute-intensive operators together. In Figure 1, we show two typical examples. In part a) there's a self-attention layer that is widely used in Transformer-based models such as Bert [16] and ViT [17]. The main component of this layer includes a batch GEMM chain that is composed of two batch GEMMs and one softmax layer. As shown in Table I in Section I, the batch GEMM chain occupies a substantial part of the whole execution time (26.65% to 40.04%). In part b) there's a convolution chain that is composed of one 3×3 convolution, one 1×1 convolution, and two ReLU layers. The convolution chain is common in CNNs [19], [40]. The convolutions can also become memory-bound under certain input shapes (discussed in Section VI).



		Reuse Dimension			
No.	order	A	B	D	E
1	mnkl	l	-	k	k,l
2	mnlk	-	-	k	k,l
3	mknl	n,l	-	k	k,l
4	mkln	l,n	n	k	k
5	mlnk	-	-	k	k
6	mlkn	n	n	k	k
⋮	⋮	⋮	⋮	⋮	⋮
24	lknm	-	n,m	m,k	m,k

Fig. 2. Different block execution orders result in different data reuse (use GEMM chain as an example).

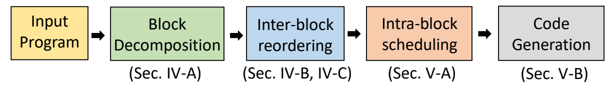


Fig. 3. The overall workflow of Chimera.

B. Major Challenges

Here, we present two major challenges in generating efficient fused kernels for compute-intensive operator chains and explain why previous work can't address these challenges. We summarize the comparison of related work in Table II.

1) The execution order of inter-blocks: Compute-intensive operators can be represented by a series of computation blocks as pointed out in previous works [58], [65] and the block execution order is critical to locality optimization. When fusing chains of compute-intensive operators, the main optimization objective is to select the optimal execution order that maximizes data reuse. We use the GEMM chain example in Figure 2 ($C = A \times B, E = C \times D$) to illustrate the effect of different execution orders. There are four different dimensions (m, n, k, l) in the GEMM chain and the two GEMMs are tiled into multiple computation blocks. The block execution order can be represented by the ordering of the four dimensions as shown in the Table in Figure 2. The order *mnkl* (the first row) indicates that we execute the blocks along dimension l first, then dimension k, then dimension n, and finally, dimension m. Under this execution order, matrix A is reused along dimension l; matrix B is not reused because when we traverse the blocks along dimension l, different data blocks of matrix B are accessed. Matrix C is an intermediate result and is stored in on-chip memory, so we don't show its reuse dimensions. Matrices D and E are always reused along k dimension because k is private to the first GEMM, which will not iterate on the computations of the second GEMM. In addition, the size of each computation block also affects the final data movement volume. As a result, the optimization problem should model block decomposition strategy and block ordering choices together.

Previous works [21], [23], [27], [36], [54], [56]–[58], [63], [65] only partially solve the problem as shown in Table II. Ansor [57], TASO [23], DNNFusion [36], MOpt [26], AStitch [63], and Roller [65] only optimize the block orders within one compute-intensive operator at a time and fix the inter-operator order by using expert rules. DNNFusion [36] classifies compute-intensive operators as the Many-to-Many mapping type and fails to fuse two or more Many-to-May operators together because its code generator cannot predict the benefit of such complex fusion. CoSA [21] uses mixed-integer programming (MIP) to optimize the total execution cycles without considering the inter-block memory access. HASCO [53] uses reinforcement learning and Bayesian optimization to explore the hardware-software desgin space. But operator fusion is not in the design space. AKG [56] uses polyhedral models to improve locality. But the polyhedral model is a general approach and its optimization space is too large to explore. As a result, it relies on heuristics to find solutions, which often gives sub-optimal performance in practice. Atomic [58] only considers inter-engine data reuse (e.g., the reuse of matrix C in Figure 2). But the data reuse from input/output data access also has a great impact on performance, which is not optimized in Atomic.

2) The intra-block code generation: Scheduling the computation within one block is the core to high-performance kernel implementation. The instructions within one block should be scheduled to hide the latency of memory access and maximize the utilization of the computation pipeline. Previous works adopt different approaches for intra-block optimization as shown in Table II. TASO [23], CoSA [21], and Atomic [58] don't generate low-level code and they have no intra-block optimizations. AKG [56] and Ansor [57] use loop transformations along with tuning methods to generate micro kernels. But they rely on a general instruction selection logic (in TVM [14] and LLVM [25]) without utilizing hardware-specific instructions such as AVX-512 and Tensor Core as pointed out in previous work [60]. In addition, the tuning process is expensive because it requires hundreds of hardware profiling steps to obtain a good performance. DNNFusion [36], Astitch [63], and BOLT [54] use hand-tuned micro kernels to optimize a fixed computation pipeline. However, their micro kernels are tightly coupled with the inter-block optimizations, making it hard to support new operators or new accelerators.

III. Overview of Chimera

In this section, we present the overall workflow of Chimera. As shown in Figure 3, Chimera is composed of four parts: block decomposition, inter-block reordering, intra-block scheduling, and code generation. The input of Chimera is a compute DAG in machine learning (described by domain-specific language). Each operator in the DAG is firstly decomposed into a series of computation blocks (Section IV-A). Then, an optimized block execution order is selected by resorting to an analytical model (Section IV-B). The analytical model relies only on the analysis of loop nests of the dense tensors. Therefore, it is general for different model topology structures (e.g. different number of tensors or operators). The original dependencies among the blocks are all preserved so that all the block orderings selected by Chimera are valid. After that, intra-block optimization is applied by using replaceable micro kernels. Chimera supports different hardware

TABLE II

THE COMPARISON OF PREVIOUS REPRESENTATIVE WORK AND CHIMERA.

Name	Codegen	Inter-block Optimization	Intra-block Optimization	Supported HW			Optimization Methodology
				CPU	GPU	NPU	
AKG [56]	Yes	Minimize Reuse Distance	Loop Transformation	Yes	Yes	Yes	Polyhedral
DNNFusion [36]	Yes	Template-based Fusion	Fixed Micro Kernel	Yes	Yes	No	Tuning
TASO [23]	No	Graph Substitution Rules	None	No	Yes	No	Tuning
AStitch [63]	Partial	Kernel Stitching Rules	Fixed Micro Kernel	No	Yes	No	Rule-based
CoSA [21]	No	Minimize Compute Cycles	None	No	Yes	No	MIP
Atomic [58]	No	Minimize Inter-engine Movement	None	No	No	No	DP
MOpt [26]	Yes	Optimize Single-op Locality	Fixed Micro Kernel	Yes	No	No	Analytical
Roller [58]	Yes	rProgram Generation Algorithm	Generated Micro Kernel	No	Yes	No	Cost Model
Ansor [57]	Yes	Sketch Generation Rules	Loop Transformation	Yes	Yes	No	Tuning
BOLT [54]	Partial	Persistent Kernels	Fixed Micro Kernel	No	Yes	No	Tuning
Chimera (ours)	Yes	Minimize Data Movement	Replaceable Micro Kernel	Yes	Yes	Yes	Analytical

backends. For each backend, Chimera registers hardware-specific micro kernel implementations to a special abstraction called replaceable micro kernel so that hardware diversity can be handled in a unified manner (in Section V-A). The details on the low-level implementations are introduced in Section V-B.

IV. INTER-BLOCK OPTIMIZATION

In this section, we introduce our inter-block optimization: block decomposition and block reordering.

A. Block Decomposition

Each compute-intensive operator in machine learning models can be decomposed into a series of computation blocks. The decomposition is implemented via loop tiling and reordering. A computation block contains a small loop nest that accesses tiles of input data to produce a tile of output data. Usually, one computation block can be placed in one processing core and all the accessed data of one block can be accommodated by the on-chip local memory. The size of each computation block is controlled by decomposition parameters. We represent all the decomposition parameters using a vector $\vec{S} = (s_1, s_2, ..., s_I)$ (totally I parameters). For example, when we decompose a GEMM chain, we will use (T_M, T_N, T_K, T_L) because there are four dimensions to decompose; for a convolution chain, we will have up to ten dimensions.

In block decomposition, we aim to select the optimal block decomposition parameters $\vec{S}$ that can maximize the overall performance. Previous work [58] proposes to calculate these parameters independently to balance the computation overhead of different blocks. But as we will show in the next section, the decomposition parameters cannot be independently chosen because they influence the overall data reuse jointly with the block execution order.

B. Minimizing Data Movement Volume via Block Reordering

Our aim in inter-block optimization is to find the optimized block execution order and decomposition parameters $\vec{S}$ that minimize the total data movement volume. Minimizing data movement volume is equivalent to maximizing data locality (or reuse). The computation blocks from different operators can be reordered to obtain a better data reuse as introduced in Figure 2. For simplicity, we assume that there are two compute-intensive operators in the input program. For more compute-intensive operators, the analysis method remains similar. Note that there are no constraints on memory-intensive operators. For memory-intensive operators, we use the standard fusion optimizations as in previous work [43], [63], which will not be discussed in this paper.

We suppose there are P loops in the first compute-intensive operator and Q loops in the second compute-intensive operator. The different orders of these loops indicate different block

Algorithm 1: Data Movement Volume Calculation and Memory Usage Algorithm for Operator Chains

```
input  : Operator chain Ops
input  : Permutation Perm = (l_{p1}, l_{p2}, ..., l_{pI})
input  : Decomposition parameters S⃗ = (s_1, s_2, ..., s_I)
output : data movement volume DV
output : memory usage MU
DV = 0; MU = 0;
for op ∈ Ops do
    total_DF = 0;
    for tensor T ∈ op.allTensors() do
        DF = getFootprint(T, S⃗);
        total_DF += DF;
        if T ∈ Ops.IOTensors() then
            DM = DF;
            keep_reuse = true;
            for loop l_{pi} ∈ reversed(Perm) do
                if l_{pi} ∈ op.allLoops() then
                    if l_{pi} accesses tensor T then
                        keep_reuse = false;
                    if not keep_reuse then
                        DM *= ⌈L_{pi} / s_{pi}⌉
            DV += DM;
    for loop l_{pi} ∈ Perm do
        if l_{pi} is private to op then
            Perm.erase(l_{pi});
    MU = max(MU, total_DF);
return DV, MU;
```

execution orders as illustrated in the example in Figure 2 in Section II-B1. In general, the whole design space is composed of all the $(P+Q)!$ different permutations of these loops. But the actual design space size can be much smaller than $(P+Q)!$ because the two compute-intensive operators may share some common loops and the ordering of common loops has no effect on data reuse. In the example of GEMM chain in Figure 2, there are 24 different reordering choices but not $(3+3)! = 720$. This is because the two GEMMs have two common dimensions m and l, and there are only four independent loops (m, n, k, l). So the design space size is 4!. In the following, we only consider that there are I ($I \leq P+Q$, which corresponds to the number of parameters in $\vec{S}$) independent loops $(l_1, l_2, ..., l_I)$ and the actual design space size is $I!$. The original loop trip count of loop l_i is denoted as L_i. A permutation of these loops is denoted as $(l_{p_1}, l_{p_2}, ..., l_{p_I})$, where $(p_1, p_2, ..., p_I)$ is a permutation of $(1, 2, 3, .., I)$. The blocks execute from the right-most (innermost) loop to the left-most (outermost) loop.

The main idea of finding the optimized block execution order (i.e., loop permutation choice) is to analytically express the data movement volume with respect to the decomposition parameters $\vec{S}$ for each permutation choice. By doing so, we can minimize the data movement volume by finding a suitable $\vec{S}$ and get the optimized permutation choice that gives the minimal data movement volume among all the candidates.

Intuitively, the data movement volume for each tensor is the product of the footprint of the tensor and the trip counts of the surrounding loops. In addition, we make three observations about the data movement. First, some loops will not cause any data movement because both their iteration variables and their inner loops' iteration variables are not used in tensor access indices. Second, once a loop causes data movement, all the surrounding outer loops will cause data movement. Third, the loops that only appear (private) in producer operators will not cause data movement in consumer operators. We use the GEMM chain example in Figure 2 to explain the observations. Under $mknl$ order, loops n, l will not cause data movement for matrix A because their loop variables are not used to access matrix A (observation 1); under $mnlk$ order, loops n, l will cause data movement for matrix A because the inner loop k has already caused data movement (observation 2); under any block order, loop k will not cause data movement to matrices D, E because k is the reduction loop of the first GEMM, which has no effect on the second GEMM (observation 3). We use the observation 1 and 2 to calculate the data movement among blocks within one operator and use the observation 3 to detect data reuse between operators.

Algorithm 1 computes the data movement volume for a given permutation choice. In detail, for the target operator chain Ops, for a given permutation $(l_{p_1}, l_{p_2}, ..., l_{p_I})$, the Algorithm traverses the operator chain according to topology order (from producers to consumers, at line 2). Only the input/output tensors of the whole operator chain (returned by function $IOTensors$) are considered (line 7) because the intermediate results are all stored in on-chip memory. We use $getFootprint$ function to calculate the data tile footprint (DF) of each tensor (line 5) according to the decomposition parameters $\vec{S}$. To calculate the data movement volume, we need to figure out how many times the data tile is replaced during execution.

TABLE III
DATA MOVEMENT VOLUME AND MEMORY USAGE FOR GEMM CHAIN UNDER THE ORDER OF $mlkn$.

	A	**B**	**C**	**D**	**E**
DM	$MK\lceil\frac{L}{T_L}\rceil$	$KL\lceil\frac{M}{T_M}\rceil$	0	$NL\lceil\frac{M}{T_M}\rceil$	$MN\lceil\frac{L}{T_L}\rceil$
DF	$T_M T_K$	$T_K T_L$	$T_M T_L$	$T_L T_N$	$T_M T_N$

Note that only the loops that access the tensor will cause data tile replacement (observation 1). We use a flag $keep\_reuse$ to check whether the current loop l_{p_i} will cause replacement (line 12-14). If so, we increase the data movement of current tensor T by multiplying the loop trip counts (line 15). This flag remains true for all other outer loops and multiplies their trip counts to data movement volume (observation 2). Before we move to the consumer operators, we need to exclude the influence of the private loops of the producer operators (see line 17-19) because such private loops won't iterate over the tensors of the consumer operators (observation 3). The algorithm also returns the maximal memory usage MU, which will be used as the problem constraints.

After getting the data movement volume DV and the memory usage MU, we can define the optimization problem as

$$min_{\vec{S}} \;\; \text{DV}, \quad \text{s.t. MU} \leq MemoryCapacity \tag{1}$$

To solve this constrained optimization problem, we first solve Equation 1 in real number domain ($\mathcal{R}$) and then get the approximate integer solution by floor rounding. In detail, we use the Lagrange Multiplier method to get the extreme values of DV and the corresponding extreme points $\vec{S}^*$. We then get approximate integer candidate solutions by the floor rounding of $\vec{S}^*$. Finally, the integer candidate that minimizes DV is chosen as the final solution.

We use the GEMM chain example in Figure 2 to elaborate more on the optimization problem. We use the execution order $mlkn$ in Figure 2 (in row 6) for demonstration. By using Algorithm 1, we can get the data movement volume and footprint of matrix A, B, C, D, E as shown in Table III (in this example, the decomposition parameters are $\vec{S} = (T_M, T_N, T_K, T_L)$). **DM** represents data movement volume, and **DF** represents data footprint of each tensor. The **DM** of C is 0 because it is an intermediate result and is always reused in on-chip memory. So the total data movement volume of the GEMM chain is

$$\begin{aligned} \text{DV}_{\text{GEMM Chain}} &= DM_A + DM_B + DM_C + DM_D + DM_E \\ &= MK\lceil\frac{L}{T_L}\rceil + KL\lceil\frac{M}{T_M}\rceil + NL\lceil\frac{M}{T_M}\rceil + MN\lceil\frac{L}{T_L}\rceil \end{aligned}$$

The peak memory usage MU of all the tensors is

$$\begin{aligned} \text{MU} &= max\{\text{GEMM1}_{MU}, \text{GEMM2}_{MU}\} \\ \text{GEMM1}_{MU} &= DF_A + DF_B + DF_C = T_M T_K + T_K T_L + T_M T_L \\ \text{GEMM2}_{MU} &= DF_C + DF_D + DF_E = T_M T_L + T_L T_N + T_M T_N \end{aligned}$$

To minimize the total data movement volume without exceeding memory capacity limit, the optimization problem is formulated as follows

$$\textbf{min} \;\; \text{DV}_{\text{GEMM Chain}} \qquad \text{s.t. MU} \leq MemoryCapacity$$

By using Lagrange Multiplier method, we get the minimum point and the minimal data movement volume:

$$DV^* = \frac{2ML(K+N)}{T_M^*}, \quad T_M^* = T_L^* = -\alpha + \sqrt{\alpha^2 + MC}, T_N^* = \alpha$$

The MC is short for $MemoryCapacity$. α is a lower bound of T_N, T_K. We set the lower bound because T_N, T_K are free variables in this optimization problem. Further, we convert real values to integers by $T_X = \min\{\lfloor T_X^* \rfloor, X\}$ ($X \in \{M, N, K, L\}$). We could also estimate the gap between the approximated solution and the optimal one and show that our solution is close to the optimal one with constant bounds. We use the ratio of the approximated data movement volume (DV_{app}) to the optimal value (DV^*) to show the difference:

$$\frac{DV_{app}}{DV^*} \leq max_{X \in \{M,L\}}\{1 + \frac{T_X^*}{X} + \frac{1}{T_X}\} \leq$$
$$max_{X \in \{M,L\}}\{1 + \frac{\sqrt{MC}}{X} + \frac{1}{\min\{X, \sqrt{MC}\}}\}, \; (MC >> \alpha)$$

C. Optimization for Multi-level Memory Hierarchy

In previous sections, we only consider one level of memory. For multiple levels of on-chip memory, our computation blocks can be further decomposed into sub-blocks recursively. The reordering of these sub-blocks will influence the data movement volume in higher level on-chip memory. We also model the cost of data movement across different layers of memory with respect to hardware configurations. Suppose that we have D levels of on-chip memory. The data movement volume for level d is defined as $DV_d(\vec{S_d})$, where $\vec{S_d}$ is the decomposition parameter list for level d. Then, the data movement cost $Cost_d(\vec{S_d})$ from level $d+1$ to level d is calculated as follows.

$$Cost_d(\vec{S_d}) = DV_d(\vec{S_d})/bw_d \tag{2}$$

where bw_d is the memory bandwidth. To minimize the overall data movement cost, we need to minimize the slowest data movement stage through all the memory levels. Therefore, we formulate the optimization as follows,

$$\begin{aligned} min_{\vec{S_1}, \vec{S_2}, ..., \vec{S_D}} \;\; \{max\{Cost_1(\vec{S_1}), ..., Cost_D(\vec{S_D})\}\}, \\ \text{s.t. MU}_1 \leq MC_1, ..., \text{MU}_D \leq MC_D \end{aligned} \tag{3}$$

MC_d is the $MemoryCapacity$ of level d memory; MU_d is the memory usage of level d memory. Chimera uses this objective function to decide the optimal block decomposition parameters and execution order for each level of memory.

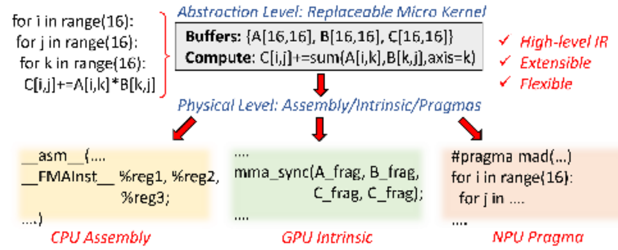


Fig. 4. Replaceable Micro Kernel.

V. Intra-block Optimization

In this section, we introduce the hardware-specific optimizations in Chimera. Different hardware accelerators require different optimizations to achieve high performance. Chimera leverages replaceable micro kernels to handle the hardware diversity.

A. Replaceable Micro Kernels

The programming model and optimization methods of different accelerators are different. For example, to implement a high-performance micro kernel for matrix multiplication, on CPUs, we need to program assembly to use the SIMD units; on GPU, we need to use Tensor Core intrinsic to map computations to Tensor Core units; on NPU, we need to add pragmas to loops to instruct the low-level compiler to generate accelerator instructions. To handle the hardware diversity through a unified approach, Chimera uses replaceable micro kernels, which are extensible and flexible for different hardware backends.

A replaceable micro kernel is an abstraction for the computation block that describes a naive loop nest over the input/output data buffers. For different accelerators, the replaceable micro kernel can be substituted by low-level hardware-specific implementations in either assembly, intrinsic, or pragmas. In Chimera, we register different hardware-specific micro kernels that perform the same computation (using different device instructions) under the same replaceable micro kernel. During compilation and code generation, Chimera will lower the replaceable micro kernel to the corresponding registered low-level micro kernel according to the target hardware. We use an example in Figure 4 to explain replaceable micro kernel in detail. In this example, we use a replaceable micro kernel to describe a 16×16 matrix multiplication using high-level loop nests and register three different low-level micro kernel implementations to this replaceable micro kernel. The micro kernels are written in low-level code (e.g., around 140 lines of assembly for CPU) and registered to Chimera using Chimera's Python interface. During code generation, the three different implementations will be automatically selected according to the target device. The registered low-level code will be automatically generated by the compiler.

B. Micro Kernel Code Generation

The code generation of micro kernels is tightly coupled with operators. Here, We focus on matrix multiplication micro kernels, which can be reused by various compute-intensive operators including GEMM, batch GEMM, and convolution.

CPU Micro Kernels. The pseudo-code of the micro kernel is displayed in Algorithm 2. We adopt an outer-product approach similar to [26], [31]. The micro kernel hides the latency of the register load/store by providing enough concurrent computations and keeps the FMA pipeline busy by emitting MI $\times$ NI consecutive FMA instructions together (MI $\times$ NI is the pipeline depth).

To decide parameters (MI, NI, MII, KI) of the microKernel, we maximize the arithmetic intensity (AI) under the constraint of available registers.

$$\begin{aligned} \max_{MI,NI,MII} \quad & AI = \#ComputeInst/\#LoadStoreInst \\ \text{s.t.} \quad & RegUsed \le \#Registers \\ \text{where} \quad & \#ComputeInst = MI \times NI \times KI \\ & \#LoadStoreInst = KI \times (MI + NI) + 2MI \times NI \\ & RegUsed = MI \times NI + NI + MII \end{aligned}$$

For example, for Cascadelake microarchitecture with 32 ZMM registers, we set MI, NI, MII to 6, 4, 2 and set KI dynamically according to the problem size with a pipeline depth of 24 to maximize the AI. During code generation, low-level assembly code will be generated according to Algorithm 2 and the parameters (MI, NI, MII, KI).

Algorithm 2: CPU Micro Kernel Design

```
constant     : RegLen # the vector register length.
parameter    : MI, NI, MII, KI
input        : A[MI, KI], B[KI, NI*RegLen]
input/output : C[MI, NI*RegLen]
register     : RegA[MII], RegB[NI], RegC[MI, NI]

for m in [0, MI, 1) do
    for n in [0, NI, 1) do
        vecLoad(C[m,n*RegLen: (n+1)*RegLen], RegC[m,n])
for k in [0, KI, 1) do
    for n in [0, NI, 1) do
        vecLoad(B[k,n*RegLen: (n+1)*RegLen], RegB[n])
    for mo in [0,MI, MII) do
        for mi in [0, MII, 1) do
            vecLoad(A[mo+mi,k], RegA[mi])
        for mi in [0, MII, 1) do
            for n in [0, NI, 1) do
                FMA(RegC[mo+mi,n], RegA[mi], RegB[n])
for m in [0, MI, 1) do
    for n in [0, NI, 1) do
        vecStore(C[m,n*RegLen: (n+1)*RegLen], RegC[m,n])
```

GPU Micro Kernels. On Tensor Core GPUs, we can use the WMMA *mma_sync* intrinsic to compute a $16 \times 16 \times 16$ matrix multiplication at a time. However, directly using the intrinsic is not efficient because each *mma_sync* intrinsic requires one corresponding matrix load and store operation. As a result, the arithmetic intensity is low, and the performance will be bounded by memory operations. To improve the arithmetic intensity, our micro kernel for GPU unrolls the inner loops and schedules the intrinsic order to perform a tiled outer-product. In detail, the micro kernel loads two 16×16 matrices for each operand matrix at a time and updates 2×2 tiles of 16×16 matrices for the result matrix. In this implementation, each loaded matrix tile is reused for two times and the overall arithmetic intensity is improved.

NPU Micro Kernels. The Ascend NPU uses a Python DSL with pragmas. The NPU micro kernel is implemented using pragmas that maps computations to dedicated hardware units (cube unit and vector unit). Low-level device binary code will be generated by the NPU's close-source compiler *CNCC* [1]. To implement the matrix multiplication micro kernel, we have to use the *mad* pragma, which expects six nested loops that computes a tiled matrix multiplication:

$$C[m1, n1, m2, n2] += A[m1, k1, m2, k2] * B[k1, n1, n2, k2]$$
$$(m1 \le M1, m2 \le M2, n1 \le N1, n2 \le N2, k1 \le K1, k2 \le K2)$$

To produce the expected loop nest and loop order, we pack the input matrices in on-chip memory using DMA instructions to produce contiguous data arrays. The overall arithmetic intensity of this micro kernel is

$$AI = \frac{M1 \times M2 \times N1 \times N2}{M1 \times M2 + N1 \times N2}$$

We maximize the AI by setting

$$M2 = N2 = Lane\_of\_cube\_units$$

and setting $M1 = N1$ according to the L0 on-chip buffer size of the NPU.

VI. Evaluation

A. Evaluation Setup

We test both subgraph fusion performance and full network performance. The subgraphs we use include the batch GEMM chains from Bert [16], ViT [17], and MLP-Mixer [46] and the convolution chains from CNNs such as SqueezeNet [22] and Yolo [40], [41]. For the whole network evaluation, we use Transformer [48], Bert [16], and ViT [17]. We use three server-class accelerators: Intel Xeon Gold 6240 AVX-512 CPU (1.125MB L1 cache, 18MB L2 cache, and 24.75MB L3 cache), Nvidia A100 Tensor Core GPU (up to 164KB/SM shared memory, 40.96MB L2 cache), and Huawei Ascend 910 NPU (64KB L0A/B buffer, 256KB L0C buffer, 1MB L1 buffer, 256KB Unified Buffer). Our baseline includes both hand-tuned libraries and state-of-the-art compilers. For libraries, we compare to PyTorch [38] (that uses MKL [3] and oneDNN [2] on CPU and uses CuBlas [5] and CuDNN [6] on GPU), TensorRT [8], and CANN (library for NPU). For compilers, we compare to the state-of-the-art machine learning compilers including Relay [43], Ansor [57], TASO [23], TVM+Cutlass [54], and AKG [56] (compiler for NPU).

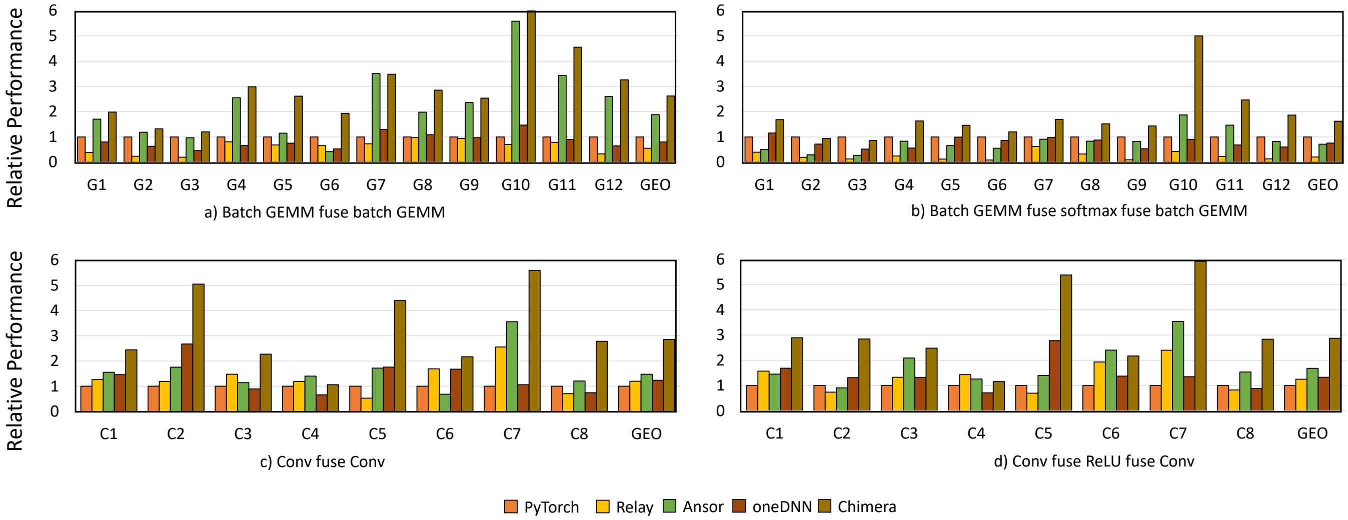


Fig. 5. The performance of fusing batch GEMM chains and fusing convolution chains on CPU.

TABLE IV
THE CONFIGURATIONS OF BATCH GEMM CHAINS.

Name	batch	M	N	K	L	Network
G1	8	512	64	64	512	Bert-Small
G2	12	512	64	64	512	Bert-Base
G3	16	512	64	64	512	Bert-Large
G4	12	256	64	64	256	ViT-Base/14
G5	16	256	64	64	256	ViT-Large/14
G6	16	256	80	80	256	ViT-Huge/14
G7	12	208	64	64	208	ViT-Base/16
G8	16	208	64	64	208	ViT-Large/16
G9	16	208	80	80	208	ViT-Huge/16
G10	1	512	64	64	256	MLP-Mixer
G11	1	768	64	64	384	MLP-Mixer
G12	1	1024	64	64	512	MLP-Mixer

TABLE V
THE CONFIGURATIONS OF CONVOLUTION CHAINS.

Name	IC	H	W	OC_1	OC_2	st_1	st_2	k_1	k_2
C1	64	112	112	192	128	2	1	3	1
C2	32	147	147	64	80	2	1	3	1
C3	64	56	56	128	64	1	1	3	1
C4	128	28	28	256	128	1	1	3	1
C5	16	227	227	64	16	4	1	3	1
C6	64	56	56	64	64	1	1	1	3
C7	64	56	56	64	64	1	1	1	1
C8	256	56	56	256	64	1	1	1	1

B. Subgraph Performance

The subgraphs we use in this section include batch GEMM chains and convolution chains. We have introduced them in Section II-A. For batch GEMM chains, we evaluate the performance of both with softmax as the intermediate memory-intensive operator and without any intermediate operator. For convolution chains, we evaluate the performance of both using ReLU as the intermediate operator and without any intermediate operators. The input configurations of the subgraphs are shown in Table IV and Table V. In Table IV, $(\text{batch}, M, K) \times (\text{batch}, K, L)$ is the first batch GEMM problem size. $(\text{batch}, M, L) \times (\text{batch}, L, N)$ is the second batch GEMM problem size. In Table V, the first convolution problem size is $(\text{batch}, IC, H, W) \times (OC_1, IC, k_1, k_1)$, and the second convolution problem size is $(\text{batch}, OC_1, \lfloor H/st_1 \rfloor, \lfloor W/st_1 \rfloor) \times (OC_2, OC_1, k_2, k_2)$. st_1 is the stride of the first convolution. st_2 is the stride of the second convolution.

AVX-512 CPU Performance. The results are shown in Figure 5. We show the relative performance normalized to PyTorch. Ansor requires a long time for tuning (about half an hour for one operator). We set it to tune 1000 trials for each subgraph. Chimera only needs several minutes to generate the fused kernels because it uses an analytical model. Relay can use hand-optimized templates without tuning. For batch GEMM fused with batch GEMM, Chimera can obtain speedups compared to both hand-tuned libraries and compilers because it can fuse the computations of two batch GEMMs and improve the overall locality. The overall speedups are $2.62\times$ to PyTorch, $4.78\times$ to Relay, $1.40\times$ to Ansor, and $3.28\times$ to oneDNN. For fusing batch GEMMs and softmax, Chimera achieves an average $1.62\times$ speedup to PyTorch. The speedups to Relay and Ansor are $7.89\times$ and $2.29\times$.

For fusing convolution chains, we also use the convolution layers from real-world networks [19], [22], [40], [41]. Convolutions (especially when kernel size is 3×3) are more complicated than batch GEMM. The sliding windows of 3×3 convolutions can result in re-computations after fusion. Relay and Ansor can't fuse these complex operators together. So they generate separate kernels for them. The speedup of Chimera is $2.38\times$ to Relay and $1.94\times$ to Ansor. In Figure 5 part d), we show the performance of Chimera when fusing convolution chains with ReLU. The speedups are in line with those of fusing two convolutions ($2.87\times$ to Pytorch, $2.30\times$ to Relay, and $1.71\times$ to Ansor).

Tensor Core GPU Performance. The results are shown in Figure 6. For fusing batch GEMM and batch GEMM (Figure 6 part a), the average speedup is $2.77\times$ to PyTorch, $3.30\times$ to TASO, $1.69\times$ to Relay, $1.33\times$ to Ansor, and $2.29\times$ to TensorRT. The speedup comes from fusing the memory-bound batch GEMMs together and reducing off-chip memory access. The total DRAM access of Chimera is reduced by $9.86\% - 59.54\%$ compared to PyTorch. Compilers such as TASO and Ansor don't fuse the two batch GEMMs and result in two separate kernel calls in the generated code.

We also compare to TVM+Cutlass [54] and the average speedup is $1.51\times$. Cutlass [7] is state-of-the-art open-source DNN template library for GPU. Recent work BOLT [54] explores the fusion of GEMM chains and convolution chains using Cutlass templates. The relevant code is open-sourced and is available in TVM [14]. We use the code to generate kernels for batch GEMM chain and show the performance in Figure 6, which is denoted as TVM+Cutlass. However, we profile the result code and find TVM+Cutlass fails to achieve high performance for our test cases. The reason is two-fold. First, the Cutlass templates are developed manually by experts and is limited in flexibility. In detail, TVM uses a front-end analysis to find fusible subgraphs in the input program by pattern matching. The pattern matching is not flexible and classifies batch GEMM chain as a non-fusible subgraph. Second, Cutlass templates only use a fixed block execution order, which may miss the optimal execution order when executing two consecutive GEMMs. By contrast, Chimera can explore different execution orders through an analytical model, which is the source of speedup.

For fusing batch GEMM chains with softmax (Figure 6 part b), the average speedup to PyTorch is $2.74\times$. We don't show the performance of TASO and TVM+Cutlass because they don't support softmax. Relay and Ansor generate three kernels for this subgraph because they can't fuse softmax. Softmax is more complicated than element-wise operators because it requires three dependent steps in calculation: *exp*, *sum*, and *div*. Chimera can fuse softmax because the *sum* operation of softmax can be merged into the second batch GEMM, and the order of *div* operation and the second batch GEMM can be swapped. As a result, the average speedup of Chimera is $1.74\times$ to Relay and $1.64\times$ to Ansor.

For fusing convolution and convolution (Figure 6 part c), the average speedups to PyTorch and TensorRT are $5.79\times$ and $2.01\times$. Not all the convolution layers in the CNNs are suitable for fusion. In general, Chimera gains speedups by fusion only when the second convolution in the convolution chain is memory-bound. Usually, point-wise convolutions tend to be memory-intensive when channel dimensions are small and they are commonly used in the initial layers of CNNs (image resolution is high and the channel feature size is small). But other convolution layers (e.g., 3×3 convolution) are usually compute-bound and are not suitable for fusion. We use case C6 in Table V to confirm this point by showing the performance of fusing point-wise convolution with 3×3 convolution. This subgraph comes from ResNet [19]. As shown in Figure 6 part c) and d), Chimera can't obtain speedup for C6 compared to Ansor because the second convolution is compute-bound. But for other subgraphs, Chimera can consistently get better performance than Ansor. For fusing convolution chain with ReLU, the average speedup to Relay is $4.32\times$; the average speedup to Ansor is $1.30\times$.

NPU Performance. At last, we evaluate the GEMM chains on NPU. For all the GEMM chains, we use batch size 1. Our baseline is the TBE library (Tensor Boost Engine) from CANN [1]. TBE provides hand-optimized GEMM implementations for Ascend NPUs. It cannot fuse two GEMMs within one kernel. Another baseline we compare to is AKG [56]. AKG can provide state-of-the-art performance on Ascend NPU for GEMM and support various fusion strategies. But fusing GEMM chain is not explored by AKG. As shown in Figure 7, Chimera achieves $2.39\times$ speedup to TBE on average. The average speedup to AKG is $1.14\times$. For some cases, Chimera doesn't obtain speedup to AKG. The reason is that the NPU we use has a small Unified Buffer to transfer intermediate results of the first GEMM. When the GEMM becomes large, the Unified Buffer becomes a bottleneck and slows down the overall execution.

C. Memory Analysis and Model Validation

We also profile the kernels generated by Chimera to provide insights into performance. We use CPU as target platform and profile the kernels of batch GEMM chains. For this subgraph, Chimera fuses the two batch GEMMs together. So we only need to profile one kernel for Chimera. For PyTorch, it uses two separate kernels and we have to profile the two batch GEMM kernels for it separately. As shown in Figure 8 part a) and b), the average L2 and L3 cache hit rates of Chimera exceed those of PyTorch. *PyTorch-1* refers to the first GEMM PyTorch uses; and *PyTorch-2* refers to the second GEMM PyTorch uses. The high cache hit rate of Chimera means that more data movement happens in fast cache (e.g., L1 and L2 cache), which is the source of speedups. We also profile the data movement amount between different levels of cache and find that the data movement between L2 and L3 cache is greatly reduced (by 59.75% on average) by Chimera compared to PyTorch as shown in Figure 8 part c). Similarly, the DRAM access of Chimera is reduced by 75.17% on average. Meanwhile, the data movement of Chimera between the L1 and L2 cache increases by 46% on average, which corresponds to the inter-op data movement.

To validate the accuracy of our data movement model, we profile the GEMM chain ($M = N = K = L = 2048$) for three different cases and show the predicted and measured data movement volume in Figure 8 part d)-f). For each case, we profile hundreds of different decomposition factors (tiling factors) and plot the corresponding data movement volume in the Figure. The x-axis is the predicted volume from our analytical model and the y-axis is the ground-truth measured

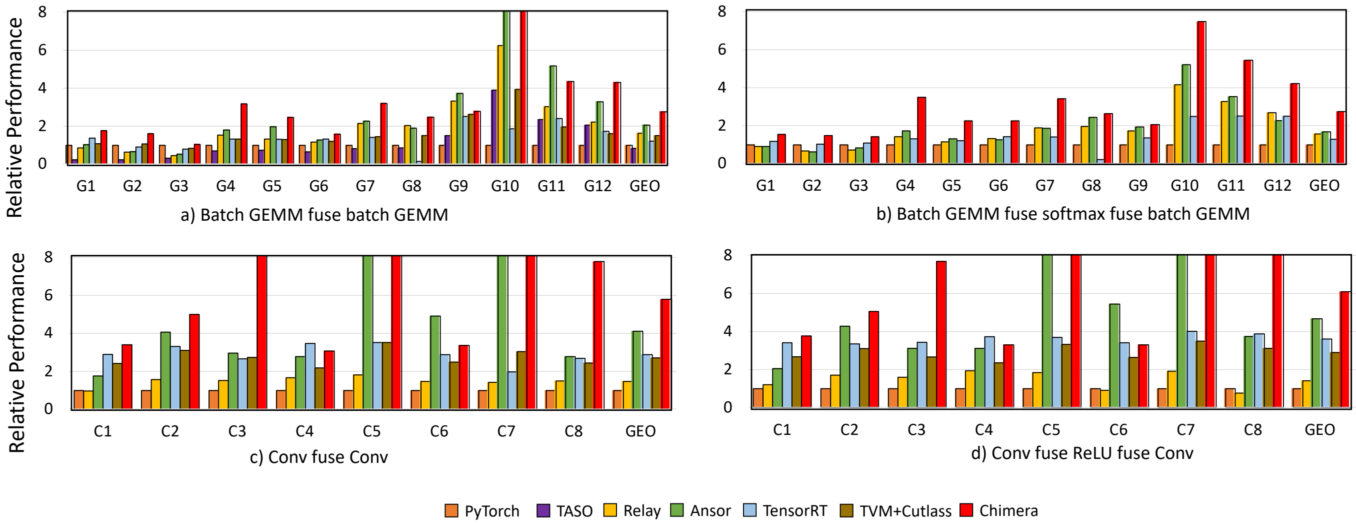


Fig. 6. The performance of fusing batch GEMM chains and fusing convolution chains on GPU.

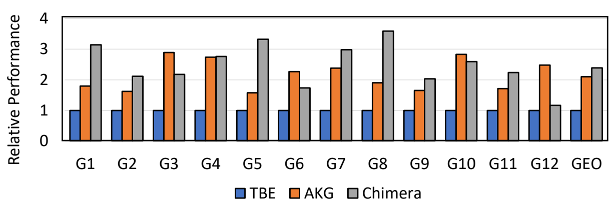


Fig. 7. The performance of fusing GEMM chain on NPU.

using hardware profiling. The plots will be close to the line $y = x$ if our predication is accurate. We focus on the data movement between L1 cache and L2 cache. For the case in part d), we use the block execution order $mlkn$. The results show that the predication accuracy is high and the correlation between the ground-truth and predictions is also high ($R^2 = 0.97$). We also show the predicted optimal data movement using a red point in the Figure. The predicted value is close to the ground-truth (the left bottom point in the Figure). For the case in part e), we use another order $mlnk$. The predictions are also accurate ($R^2 = 0.98$). In part f), we use the order $mlkn$ but force the second GEMM not to reuse the intermediate matrix C, which will result in more data movement. This case is used to show that reusing intermediate data is also critical to performance when generating fused kernels. Among the three cases, the optimal order is $mlkn$ with intermediate data reuse in part d). This order is actually the optimal order found by Chimera. Through this experiment, we show that our analytical model is efficient and accurate.

D. End-to-end Performance

For full network performance evaluation, we use Transformer (referred to as TF), Bert, and ViT (batch size is 1). *TF-Small*, *TF-Base*, *TF-Large* are three different configurations for Transformers, the sequence length of which is set to 512. The batch GEMM chain input shapes for the different configurations are shown in Table IV.

We use PyTorch with CuDNN enabled as baseline (denoted as *PyTorch+CuDNN*). We also compare Chimera to TensorRT, CuDNN, and Ansor. Relay is able to invoke TensorRT and CuDNN directly (denoted *Relay+TensorRT* and *Relay+CuDNN*). Ansor is integrated with Relay so we can use Ansor to generate batch GEMM chain kernels without using CuDNN (denoted as *Relay+Ansor*). We set Ansor to tune 1000 trials for each batch GEMM chain kernel. To compare the performance of Chimera, we integrate Chimera with Relay and replace the batch GEMM chain kernels of Relay with those of Chimera (denoted as *Relay+Chimera*).

We use one A100 GPU as the target device. The performance results are shown in Figure 9. *Relay+Chimera* is much faster than *PyTorch+CuDNN* because *Relay+Chimera* uses static graphs, while PyTorch uses dynamic graphs. Compared to *Relay+TensorRT*, *Relay+CuDNN*, and *Relay+Ansor*, the geometric speedups of *Relay+Chimera* are $1.42\times$, $1.31\times$, and $1.22\times$, respectively. *Relay+TensorRT* is slower than the other compilers because TensorRT can't fuse the softmax layer in the self-attention layer. Meanwhile, the batch GEMMs in the networks are irregular, which is not well optimized in TensorRT.

E. Discussion

Optimization Overhead. Chimera uses an analytical data movement analysis for inter-block and intra-block optimization. We compare the optimization overhead of Chimera with the state-of-the-art optimizing compiler Ansor [57] using batch GEMM chains on Intel Xeon Gold 6240 CPU. Ansor uses hardware-profiling to train a cost model and then uses the cost model to guide the exploration of the optimization space.

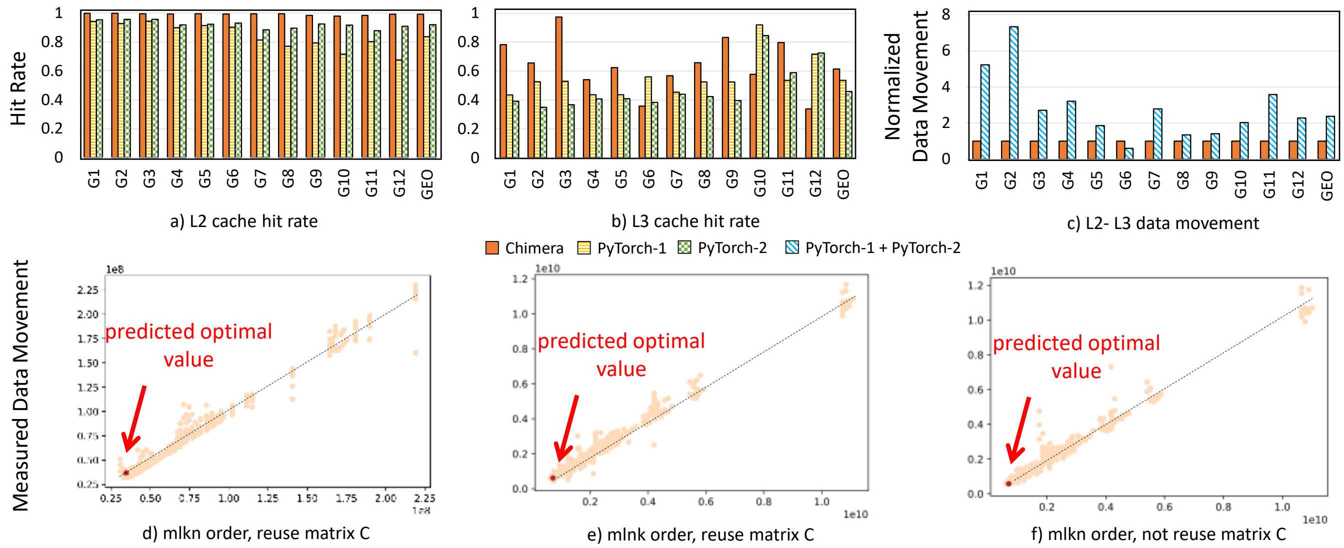


Fig. 8. Memory analysis and model validation of Chimera and PyTorch on CPU. We use batch GEMM chain as example.

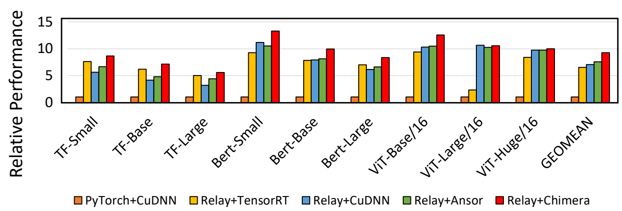


Fig. 9. The end-to-end network evaluation on A100 GPU

.

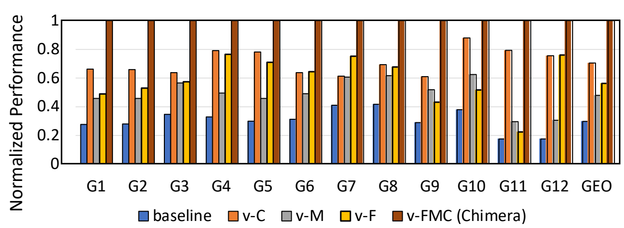


Fig. 10. Ablation study results on CPU.

Chimera's optimization is much faster than Ansor (21.89× on average) and achieves 1.39× speedup because it estimates data movement volume using analytical models before compilation. In contrast, Ansor needs to profile the kernels on hardware frequently during compilation.

Ablation Study. We perform an ablation study to show the performance contribution of our cost model (*C*), fusion techniques (*F*), and micro kernel (*M*), respectively. We use batch GEMM chains for evaluation. The normalized performance is shown in Figure 10. We prepare five versions of Chimera. *baseline* is Chimera with cost model, fusion, and micro kernel all disabled. For other versions, we use the name *C, F, M* to indicate if the corresponding optimization is enabled. For example, version *v-C* has only cost model enabled; version *v-F* has only fusion optimizations enabled. When cost model is disabled, Chimera randomly samples 100 candidate tiling factors for each block order and chooses the best one by evaluating them on hardware. On average, compared to *baseline*, cost model can bring 2.37× speedup, fusion techniques can bring 1.89× speedup, and micro kernel can bring 1.61× speedup. Collectively, cost model, fusion, and micro kernel optimizations are all critical to final high performance.

VII. Related Work

Various hand-tuned libraries [2], [3], [5]–[7], [28], code generation compilers [12], [14], [23], [32], [35], [36], [43], [45], [47], [49], [56], [57], [63], mappers [20], [21], [37], [55], and accelerators [10], [18], [33], [44], [51] are developed to improve the performance of machine learning models.

Library-based Fusion. Fusing compute-intensive operators has been exploited by several previous works. Wang et al. [50] empirically explore the fusion of convolution layers in CNNs. Ashari et al. [11] propose to implement fused kernels for a specific computation pattern in machine learning. Although providing extremely high performance, these works rely on hand-optimized kernels and is customized for specific workloads. Liang et al. [29] propose to fuse GPU kernels both spatially and temporally by threadblock interleaving to fully utilize the hardware resources. Rammer [32] and Versapipe [62] use persistent threadblocks to perform task scheduling for GPU kernel launching. Astra [45] can fuse GEMM workloads in RNNs. But it doesn't generate low-level code and relies on hand-tuned libraries. Li et al. [28] and TASO [23] can fuse parallel convolutions to increase parallelism. However, they can't fuse convolutions with dependencies. BOLT [54] uses Cutlass [7] template library to generate code for fused GEMM chains and convolution chains. Compared to these works, Chimera doesn't rely on external libraries and is more general for new operators and accelerators.

Transformation-based Fusion. Recent compilers also use loop transformation techniques to fuse operators. Halide [39] provides primitives to support kernel fusion and uses auto-schedulers [9], [34] to fuse kernels. But it focuses on image processing pipelines and the operators are not as complex as GEMM and convolution. TVM [14] uses different schedulers AutoTVM [15], FlexTensor [61], and Ansor [57] to provide fusion supports for memory-intensive operators. Fusion Stitching [64] and AStitch [63] enlarge the fusion scope by using shared memory and global memory as the intermediate buffer. However, they use compute-intensive operators as dividing lines for fusion and don't fuse compute-intensive operators together, missing the opportunities for further fusion optimizations. NeoFlow [59] explicitly avoids the fusion of compute-intensive operators because of its limited code generation flexibility. DNNFusion [36] is designed for mobile devices (e.g., ARM CPU and GPU). It fails to fuse compute-intensive operators because its fusion algorithm always considers fusing compute-intensive operators as non-beneficial. This rule gives good results for mobile device, but is too conservative for server-level accelerators because server-level accelerators have larger on-chip memory, which provides more opportunities for locality optimization for compute-intensive operator chains.

Hardware Accelerators and Mappers. Besides software fusion works, many hardware solutions for fusion are proposed. Xiao et al. [52] propose to fuse CNN layers and use heterogeneous algorithms to accelerate the fused layers on FPGA. FusedLayer [10], FixyNN [51], FixyFPGA [33], and Tangram [18], Ascend [30] implement efficient accelerators that can pipeline different DNN layers to gain inter-layer and intra-layer parallelism. Although they provide efficient hardware support for fusion, the performance of these accelerators for real workloads depends on the quality of the mappings between applications and hardware. Current mappers TimeLoop [37], Interstellar [55], Mind Mappings [20], CoSA [21], HASCO [53], and AMOS [60] are designed for perfect loop nests. However, fusion will produce imperfect loop nests. As a result, these mappers cannot fully exploit the high performance of the new accelerators. Chimera's analysis and optimizations are generally designed for the fusion of compute-intensive operator chains, which are able to exploit new hardware features for locality optimizations.

VIII. Conclusion

Generating fused kernels for compute-intensive operator chains in machine learning models is beneficial for performance. But the related optimizations in current libraries and compilers are rudimentary and thus they can't fully exploit the performance of emerging hardware. In this paper, we propose Chimera, an optimizing compiler that fuses memory-bound compute-intensive operators. It optimizes inter-block data movement and intra-block computations. It can generate efficient fused kernels for improving locality. On CPU, GPU, and NPU, the speedups to hand-tuned libraries are up to $2.87\times$, $2.29\times$, and $2.39\times$, respectively. Compared to state-of-the-art compilers, the speedups are up to $2.29\times$, $1.64\times$, and $1.14\times$ for CPU, GPU, and NPU.

Acknowledgements

We thank all the anonymous reviewers for their suggestions. This work is supported in part by the National Natural Science Foundation of China (NSFC) under grant No. U21B2017 and in part by Project 2020BD024 supported by PKU-Baidu Fund.

References

[1] "Huawei Compute Architecture for Neural Networks (CANN)," https://e.huawei.com/hk/products/cloud-computing-dc/atlas/cann.

[2] "Intel oneAPI Deep Neural Network Library," https://github.com/oneapi-src/oneDNN.

[3] "Intel oneAPI Math Kernel Library," https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html.

[4] "Nvidia Ampere Whitepaper," https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.

[5] "Nvidia CuBLAS," https://developer.nvidia.com/cublas.

[6] "Nvidia CuDNN," https://developer.nvidia.com/cudnn.

[7] "Nvidia CUTLASS," https://github.com/NVIDIA/cutlass.

[8] "Nvidia TensorRT," https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html.

[9] A. Adams, K. Ma, L. Anderson, R. Baghdadi, T. Li, M. Gharbi, B. Steiner, S. Johnson, K. Fatahalian, F. Durand, and J. Ragan-Kelley, "Learning to optimize halide with tree search and random programs," *ACM Trans. Graph.*, vol. 38, no. 4, pp. 121:1–121:12, 2019. [Online]. Available: https://doi.org/10.1145/3306346.3322967

[10] M. Alwani, H. Chen, M. Ferdman, and P. A. Milder, "Fused-layer CNN accelerators," in *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*. IEEE Computer Society, 2016, pp. 22:1–22:12. [Online]. Available: https://doi.org/10.1109/MICRO.2016.7783725

[11] A. Ashari, S. Tatikonda, M. Boehm, B. Reinwald, K. Campbell, J. Keenleyside, and P. Sadayappan, "On optimizing machine learning workloads via kernel fusion," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, A. Cohen and D. Grove, Eds. ACM, 2015, pp. 173–182. [Online]. Available: https://doi.org/10.1145/2688500.2688521

[12] R. Baghdadi, J. Ray, M. B. Romdhane, E. D. Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. P. Amarasinghe, "Tiramisu: A polyhedral compiler for expressing fast and portable code," *CoRR*, vol. abs/1804.10694, 2018. [Online]. Available: http://arxiv.org/abs/1804.10694

[13] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[14] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Q. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: an automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, 2018, pp. 578–594. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/chen

[15] T. Chen, L. Zheng, E. Q. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," in *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, 2018, pp. 3393–3404. [Online]. Available: http://papers.nips.cc/paper/7599-learning-to-optimize-tensor-programs

[16] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018. [Online]. Available: http://arxiv.org/abs/1810.04805

[17] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=YicbFdNTTy

[18] M. Gao, X. Yang, J. Pu, M. Horowitz, and C. Kozyrakis, "TANGRAM: optimized coarse-grained dataflow for scalable NN accelerators," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, Eds. ACM, 2019, pp. 807–820. [Online]. Available: https://doi.org/10.1145/3297858.3304014

[19] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, 2016, pp. 770–778. [Online]. Available: https://doi.org/10.1109/CVPR.2016.90

[20] K. Hegde, P. Tsai, S. Huang, V. Chandra, A. Parashar, and C. W. Fletcher, "Mind mappings: enabling efficient algorithm-accelerator mapping space search," in *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, T. Sherwood, E. Berger, and C. Kozyrakis, Eds. ACM, 2021, pp. 943–958. [Online]. Available: https://doi.org/10.1145/3445814.3446762

[21] Q. Huang, A. Kalaiah, M. Kang, J. Demmel, G. Dinh, J. Wawrzynek, T. Norell, and Y. S. Shao, "Cosa: Scheduling by constrained optimization for spatial accelerators," in *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021*. IEEE, 2021, pp. 554–566. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00050

[22] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[23] Z. Jia, O. Padon, J. J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "TASO: optimizing deep learning computation with automatic generation of graph substitutions," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, T. Brecht and C. Williamson, Eds. ACM, 2019, pp. 47–62. [Online]. Available: https://doi.org/10.1145/3341301.3359630

[24] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, P. L. Bartlett, F. C. N. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., 2012, pp. 1106–1114. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks

[25] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 2004, pp. 75–88. [Online]. Available: https://doi.org/10.1109/CGO.2004.1281665

[26] R. Li, Y. Xu, A. Sukumaran-Rajam, A. Rountev, and P. Sadayappan, "Analytical characterization and design space exploration for optimization of cnns," *CoRR*, vol. abs/2101.09808, 2021. [Online]. Available: https://arxiv.org/abs/2101.09808

[27] R. Li, Y. Xu, A. Sukumaran-Rajam, A. Rountev, and P. Sadayappan, "Analytical characterization and design space exploration for optimization of cnns," in *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, T. Sherwood, E. Berger, and C. Kozyrakis, Eds. ACM, 2021, pp. 928–942. [Online]. Available: https://doi.org/10.1145/3445814.3446759

[28] X. Li, Y. Liang, S. Yan, L. Jia, and Y. Li, "A coordinated tiling and batching framework for efficient GEMM on gpus," in *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, J. K. Hollingsworth and I. Keidar, Eds. ACM, 2019, pp. 229–241. [Online]. Available: https://doi.org/10.1145/3293883.3295734

[29] Y. Liang, H. P. Huynh, K. Rupnow, R. S. M. Goh, and D. Chen, "Efficient GPU spatial-temporal multitasking," *IEEE Trans. Parallel Distributed Syst.*, vol. 26, no. 3, pp. 748–760, 2015. [Online]. Available: https://doi.org/10.1109/TPDS.2014.2313342

[30] H. Liao, J. Tu, J. Xia, H. Liu, X. Zhou, H. Yuan, and Y. Hu, "Ascend: a scalable and unified architecture for ubiquitous deep neural network computing : Industry track paper," in *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021*. IEEE, 2021, pp. 789–801. [Online]. Available: https://doi.org/10.1109/HPCA51647.2021.00071

[31] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Orti, "Analytical modeling is enough for high-performance blis," *ACM Trans. Math. Softw.*, vol. 43, no. 2, aug 2016. [Online]. Available: https://doi.org/10.1145/2925987

[32] L. Ma, Z. Xie, Z. Yang, J. Xue, Y. Miao, W. Cui, W. Hu, F. Yang, L. Zhang, and L. Zhou, "Rammer: Enabling holistic deep learning compiler optimizations with rtasks," in *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, 2020, pp. 881–897.

[33] J. Meng, S. K. Venkataramanaiah, C. Zhou, P. Hansen, P. N. Whatmough, and J. Seo, "Fixyfpga: Efficient FPGA accelerator for deep neural networks with high element-wise sparsity and without external memory access," in *31st International Conference on Field-Programmable Logic and Applications, FPL 2021, Dresden, Germany, August 30 - Sept. 3, 2021*. IEEE, 2021, pp. 9–16. [Online]. Available: https://doi.org/10.1109/FPL53798.2021.00010

[34] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian, "Automatically scheduling halide image processing pipelines," *ACM Trans. Graph.*, vol. 35, no. 4, pp. 83:1–83:11, 2016. [Online]. Available: https://doi.org/10.1145/2897824.2925952

[35] S. Nakandala, K. Saur, G. Yu, K. Karanasos, C. Curino, M. Weimer, and M. Interlandi, "A tensor compiler for unified machine learning prediction serving," *CoRR*, vol. abs/2010.04804, 2020. [Online]. Available: https://arxiv.org/abs/2010.04804

[36] W. Niu, J. Guan, Y. Wang, G. Agrawal, and B. Ren, "Dnnfusion: accelerating deep neural networks execution with advanced operator fusion," in *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, S. N. Freund and E. Yahav, Eds. ACM, 2021, pp. 883–898. [Online]. Available: https://doi.org/10.1145/3453483.3454083

[37] A. Parashar, P. Raina, Y. S. Shao, Y. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. S. Emer, "Timeloop: A systematic approach to DNN accelerator evaluation," in *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2019, Madison, WI, USA, March 24-26, 2019*. IEEE, 2019, pp. 304–315. [Online]. Available: https://doi.org/10.1109/ISPASS.2019.00042

[38] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, 2019, pp. 8024–8035. [Online]. Available: http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library

[39] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. P. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, 2013, pp. 519–530. [Online]. Available: https://doi.org/10.1145/2491956.2462176

[40] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, 2016, pp. 779–788. [Online]. Available: https://doi.org/10.1109/CVPR.2016.91

[41] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.

[42] S. Ren, K. He, R. B. Girshick, and J. Sun, "Faster R-CNN: towards real-time object detection with region proposal networks," in *Advances in Neural Information Processing Systems 28: Annual*

Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds., 2015, pp. 91–99. [Online]. Available: http://papers.nips.cc/paper/5638-faster-r-cnn-towards-real-time-object-detection-with-region-proposal-networks

[43] J. Roesch, S. Lyubomirsky, M. Kirisame, J. Pollock, L. Weber, Z. Jiang, T. Chen, T. Moreau, and Z. Tatlock, "Relay: A high-level IR for deep learning," *CoRR*, vol. abs/1904.08368, 2019. [Online]. Available: http://arxiv.org/abs/1904.08368

[44] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. R. Pinckney, P. Raina, S. G. Tell, Y. Zhang, W. J. Dally, J. S. Emer, C. T. Gray, B. Khailany, and S. W. Keckler, "Simba: Scaling deep-learning inference with multi-chip-module-based architecture," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 2019, pp. 14–27. [Online]. Available: https://doi.org/10.1145/3352460.3358302

[45] M. Sivathanu, T. Chugh, S. S. Singapuram, and L. Zhou, "Astra: Exploiting predictability to optimize deep learning," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, Eds. ACM, 2019, pp. 909–923. [Online]. Available: https://doi.org/10.1145/3297858.3304072

[46] I. O. Tolstikhin, N. Houlsby, A. Kolesnikov, L. Beyer, X. Zhai, T. Unterthiner, J. Yung, A. Steiner, D. Keysers, J. Uszkoreit, M. Lucic, and A. Dosovitskiy, "Mlp-mixer: An all-mlp architecture for vision," *CoRR*, vol. abs/2105.01601, 2021. [Online]. Available: https://arxiv.org/abs/2105.01601

[47] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," *CoRR*, vol. abs/1802.04730, 2018. [Online]. Available: http://arxiv.org/abs/1802.04730

[48] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, pp. 5998–6008, 2017.

[49] M. Wahib and N. Maruyama, "Scalable kernel fusion for memory-bound GPU applications," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*, T. Damkroger and J. J. Dongarra, Eds. IEEE Computer Society, 2014, pp. 191–202. [Online]. Available: https://doi.org/10.1109/SC.2014.21

[50] X. Wang, G. Li, X. Dong, J. Li, L. Liu, and X. Feng, "Accelerating deep learning inference with cross-layer data reuse on gpus," in *Euro-Par 2020: Parallel Processing - 26th International Conference on Parallel and Distributed Computing, Warsaw, Poland, August 24-28, 2020, Proceedings*, ser. Lecture Notes in Computer Science, M. Malawski and K. Rzadca, Eds., vol. 12247. Springer, 2020, pp. 219–233. [Online]. Available: https://doi.org/10.1007/978-3-030-57675-2_14

[51] P. N. Whatmough, C. Zhou, P. Hansen, S. K. Venkataramanaiah, J. Seo, and M. Mattina, "Fixynn: Energy-efficient real-time mobile computer vision hardware acceleration via transfer learning," in *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*, A. Talwalkar, V. Smith, and M. Zaharia, Eds. mlsys.org, 2019. [Online]. Available: https://proceedings.mlsys.org/book/281.pdf

[52] Q. Xiao, Y. Liang, L. Lu, S. Yan, and Y. Tai, "Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on fpgas," in *Proceedings of the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA, June 18-22, 2017*. ACM, 2017, pp. 62:1–62:6. [Online]. Available: https://doi.org/10.1145/3061639.3062244

[53] Q. Xiao, S. Zheng, B. Wu, P. Xu, X. Qian, and Y. Liang, "HASCO: towards agile hardware and software co-design for tensor computation," in *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021*. IEEE, 2021, pp. 1055–1068. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00086

[54] J. Xing, L. Wang, S. Zhang, J. Chen, A. Chen, and Y. Zhu, "Bolt: Bridging the gap between auto-tuners and hardware-native performance," in *Proceedings of Machine Learning and Systems 2022, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022*, D. Marculescu, Y. Chi, and C. Wu, Eds. mlsys.org, 2022. [Online]. Available: https://proceedings.mlsys.org/paper/2022/hash/38b3eff8baf56627478ec76a704e9b52-Abstract.html

[55] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina, C. Kozyrakis, and M. Horowitz, "Interstellar: Using halide's scheduling language to analyze DNN accelerators," in *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, J. R. Larus, L. Ceze, and K. Strauss, Eds. ACM, 2020, pp. 369–383. [Online]. Available: https://doi.org/10.1145/3373376.3378514

[56] J. Zhao, B. Li, W. Nie, Z. Geng, R. Zhang, X. Gao, B. Cheng, C. Wu, Y. Cheng, Z. Li, P. Di, K. Zhang, and X. Jin, "AKG: automatic kernel generation for neural processing units using polyhedral transformations," in *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, S. N. Freund and E. Yahav, Eds. ACM, 2021, pp. 1233–1248. [Online]. Available: https://doi.org/10.1145/3453483.3454106

[57] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, J. E. Gonzalez, and I. Stoica, "Ansor: Generating high-performance tensor programs for deep learning," in *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 2020, pp. 863–879. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/zheng

[58] S. Zheng, X. Zhang, L. Liu, S. Wei, and S. Yin, "Atomic dataflow based graph-level workload orchestration for scalable DNN accelerators," in *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2022, Seoul, South Korea, April 2-6, 2022*. IEEE, 2022, pp. 475–489. [Online]. Available: https://doi.org/10.1109/HPCA53966.2022.00042

[59] S. Zheng, R. Chen, Y. Jin, A. Wei, B. Wu, X. Li, S. Yan, and Y. Liang, "Neoflow: A flexible framework for enabling efficient compilation for high performance dnn training," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[60] S. Zheng, R. Chen, A. Wei, Y. Jin, Q. Han, L. Lu, B. Wu, X. Li, S. Yan, and Y. Liang, "AMOS: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction," in *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, V. Salapura, M. Zahran, F. Chong, and L. Tang, Eds. ACM, 2022, pp. 874–887. [Online]. Available: https://doi.org/10.1145/3470496.3527440

[61] S. Zheng, Y. Liang, S. Wang, R. Chen, and K. Sheng, "Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system," in *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020 [ASPLOS 2020 was canceled because of COVID-19]*, J. R. Larus, L. Ceze, and K. Strauss, Eds. ACM, 2020, pp. 859–873. [Online]. Available: https://doi.org/10.1145/3373376.3378508

[62] Z. Zheng, C. Oh, J. Zhai, X. Shen, Y. Yi, and W. Chen, "Versapipe: a versatile programming framework for pipelined computing on GPU," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18, 2017*, H. C. Hunter, J. Moreno, J. S. Emer, and D. Sánchez, Eds. ACM, 2017, pp. 587–599. [Online]. Available: https://doi.org/10.1145/3123939.3123978

[63] Z. Zheng, X. Yang, P. Zhao, G. Long, K. Zhu, F. Zhu, W. Zhao, X. Liu, J. Yang, J. Zhai, S. L. Song, and W. Lin, "Astitch: enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures," in *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds. ACM, 2022, pp. 359–373. [Online]. Available: https://doi.org/10.1145/3503222.3507723

[64] Z. Zheng, P. Zhao, G. Long, F. Zhu, K. Zhu, W. Zhao, L. Diao, J. Yang, and W. Lin, "Fusionstitching: boosting memory intensive computations for deep learning workloads," *arXiv preprint arXiv:2009.10924*, 2020.

[65] H. Zhu, R. Wu, Y. Diao, S. Ke, H. Li, C. Zhang, J. Xue, L. Ma, Y. Xia, W. Cui *et al.*, "{ROLLER}: Fast and efficient tensor compilation for deep learning," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 233–248.