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

Size Zheng, Siyuan Chen, Peidi Song, Renze Chen, Xiuhong Li, Shengen Yan, Dahua Lin, Jingwen Leng, Yun Liang, Yunchao Shi, and Wei Gao

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.

Chimera, 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 generates fused kernels for compute-intensive operators. A100 Tensotor Core

TABLE I

The Compute/Memory Breakdown of ML Models and the Performance of Different Accelerators.

<table>
<thead>
<tr>
<th>Name</th>
<th>ML Model Breakdown</th>
<th>%MI</th>
<th>%CI</th>
<th>%BMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformer</td>
<td></td>
<td>19.45%</td>
<td>40.51%</td>
<td>40.04%</td>
</tr>
<tr>
<td>Bert-Base</td>
<td></td>
<td>30.56%</td>
<td>42.79%</td>
<td>26.65%</td>
</tr>
<tr>
<td>ViT-Huge</td>
<td></td>
<td>15.63%</td>
<td>50.85%</td>
<td>33.52%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compute and Memory Characteristics of Accelerators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>Xeon Gold</td>
</tr>
<tr>
<td>A100</td>
</tr>
<tr>
<td>Memory BW</td>
</tr>
<tr>
<td>Peak Perf</td>
</tr>
<tr>
<td>Peak Perf BW</td>
</tr>
</tbody>
</table>

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.

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.

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.

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×, 2.29×, and 2.39× 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×, 1.64×, and 1.14× 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).
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 mmkl (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], ASStitch [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 design 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. TASSO [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 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Codegen</th>
<th>Inter-block Optimization</th>
<th>Intra-block Optimization</th>
<th>Supported HW</th>
<th>Optimization Methodology</th>
</tr>
</thead>
<tbody>
<tr>
<td>AKG [56]</td>
<td>Yes</td>
<td>Minimize Reuse Distance</td>
<td>Loop Transformation</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DNNFusion [36]</td>
<td>Yes</td>
<td>Template-based Fusion</td>
<td>Fixed Micro Kernel</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>FASO [23]</td>
<td>No</td>
<td>Graph Substitution Rules</td>
<td>None</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>AStitch [63]</td>
<td>Partial</td>
<td>Kernel Stitching Rules</td>
<td>Fixed Micro Kernel</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>CoSA [21]</td>
<td>No</td>
<td>Minimize Compute Cycles</td>
<td>None</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Atomic [58]</td>
<td>No</td>
<td>Minimize Inter-engine Movement</td>
<td>None</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>MOpt [26]</td>
<td>Yes</td>
<td>Optimize Single-op Locality</td>
<td>Fixed Micro Kernel</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Roller [58]</td>
<td>Yes</td>
<td>tProgram Generation Algorithm</td>
<td>Generated Micro Kernel</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Ansor [57]</td>
<td>Yes</td>
<td>Sketch Generation Rules</td>
<td>Loop Transformation</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>BOLT [54]</td>
<td>Partial</td>
<td>Persistent Kernels</td>
<td>Fixed Micro Kernel</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Chimera (ours)</td>
<td>Yes</td>
<td>Minimize Data Movement</td>
<td>Replaceable Micro Kernel</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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
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 in Figure 2, there are 24 different reordering choices but not \((3 + 3)! = 720\). This is because the two GEMMs have two common dimensions \(m, n\) and \(l, t\), 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 \(S\) 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 \(S\) for each permutation choice. By doing so, we can minimize the data movement volume by finding a suitable \(S\) 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 \(mnlk\) 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 which has no effect on the second GEMM (observation 3). We define, the optimization problem as

\[
\min_{S} \text{DV, s.t. } \text{MU} \leq \text{MemoryCapacity} \tag{1}
\]

To solve this constrained optimization problem, we first solve Equation 1 in real number domain \((\mathbb{R})\) 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 \(S^*\). We then get approximate integer candidate solutions by the floor rounding of \(S^*\). 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 \(mnlk\) 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 \(S = (T_M, T_N, T_K, T_L)\)). DM represents data movement volume, and DF represents data movement 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

\[
\text{DV}_{\text{GEMM Chain}} = \text{DM}_A + \text{DM}_B + \text{DM}_C + \text{DM}_D + \text{DM}_E
\]

\[
= MK\left[\frac{L}{T_L}\right] + KL\left[\frac{M}{T_M}\right] + NL\left[\frac{M}{T_M}\right] + MN\left[\frac{L}{T_L}\right]
\]

The peak memory usage \(\text{MU}\) of all the tensors is

\[
\text{MU} = \max\{\text{GEMM1}_\text{MU}, \text{GEMM2}_\text{MU}\}
\]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>(MK\times\frac{L}{T_L})</td>
<td>(KL\times\frac{M}{T_M})</td>
<td>0</td>
<td>(NL\times\frac{M}{T_M})</td>
<td>(MN\times\frac{L}{T_L})</td>
</tr>
</tbody>
</table>

### Table III

Data movement volume and memory usage for GEMM chain under the order of \(mnlk\).
To minimize the total data movement volume without exceeding memory capacity limit, the optimization problem is formulated as follows:

$$\min \ DV_{\text{GEMM Chain}} \quad \text{s.t. } \ MU \leq \text{MemoryCapacity}$$

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. $\alpha$ 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

$$T^* = T + \alpha \sqrt{MC}, T_N = \alpha$$

bound of $T$. Further, we convert real values to integers by

$$T^* = T + \alpha \sqrt{MC}, T_N = \alpha$$

movement stage through all the memory levels. Therefore, we can 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 ($DV_{\text{app}}$) to the optimal value ($DV^*$) to show the difference:

$$\frac{DV_{\text{app}}}{DV^*} \leq \max_{X \in \{M,L\}} \{1 + \frac{TX}{X} + \frac{1}{TX} \} \leq \max_{X \in \{M,L\}} \left\{1 + \frac{MC}{X} + \frac{1}{\min\{X,\sqrt{MC}\}} \right\}, \quad (MC >> \alpha)$$

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 $DV_d(S_d)$, where $S_d$ is the decomposition parameter list for level $d$. Then, the data movement cost $Cost_d(S_d)$ from level $d+1$ to level $d$ is calculated as follows:

$$Cost_d(S_d) = DV_d(S_d)/bw_d$$

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,

$$\min_{S_1, S_2, \ldots, S_D} \left\{\max\{Cost_1(S_1), \ldots, Cost_D(S_D)\}\right\}, \quad \text{s.t. } \text{MU}_1 \leq MC_1, \ldots, \text{MU}_D \leq MC_D$$

$MC_d$ is the MemoryCapacity of level $d$ memory; $\text{MU}_d$ 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.

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 $16 \times 16$ matrix multiplication using high-level loop nests and register three different low-level micro kernels implementation 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 × NI consecutive FMA instructions together (MI × 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.

\[
\max_{M_1, N_1, M_{II}} AI = \frac{\text{#ComputeInst}}{\text{#LoadStoreInst}}
\]

s.t. \( \text{RegUsed} \leq \# \text{Registers} \)

where \( \text{#ComputeInst} = M_1 \times N_1 \times K_1 \)
\( \text{#LoadStoreInst} = K_1 \times (M_1 + N_1) + 2M_1 \times N_1 \)
\( \text{RegUsed} = M_1 \times N_1 + N_1 + M_{II} \)

For example, for Cascade lake 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/output| A[MI, KI], B[KI, NI*RegLen]          |
| register    | RegA[MII], RegB[NI], RegC[MI, NI]   |

1. for m in [0, MI, 1) do
2.   for n in [0, NI, 1) do
3.     vecLoad(RegC[m,n*RegLen: (n+1)*RegLen], C[m,n])
4.     for k in [0, KI, 1) do
5.       vecLoad(RegB[n*RegLen: (n+1)*RegLen], B[n])
6.     for m in [0, MII, 1) do
7.       vecLoad(RegA[mi*RegLen: (mi+1)*RegLen], A[mi])
8.       for n in [0, NI, 1) do
9.         FMA(RegC[mo+mi,n], RegA[mi], RegB[n])

GPU Micro Kernels. On Tensor Core GPUs, we can use the WMMA mmu_sync intrinsic to compute a 16 × 16 × 16 matrix multiplication at a time. However, directly using the intrinsic is not efficient because each mmu_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:

\[C[m_1, n_1, m_2, n_2] = A[m_1, k_1, m_2, k_2] \times B[k_1, n_1, n_2, k_2]
\]

\((m_1 \leq M_1, m_2 \leq M_2, n_1 \leq N_1, n_2 \leq N_2, k_1 \leq K_1, k_2 \leq K_2)\)

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

\[AI = \frac{M_1 \times M_2 \times N_1 \times N_2}{M_1 \times M_2 + N_1 \times N_2}\]

We maximize the AI by setting

\[M_2 = N_2 = \text{Lane_of_cube_units}\]

and setting \(M_1 = N_1\) 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]. For the whole network evaluation, we use Transformer [48], Bert [16], and ViT [17]. We use three server-class accelerators: Intel Xeon Gold 6120 12-core server (1.225MB 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], TASSO [23], TVM+Cutlass [54], and AKG [56] (compiler for NPU).
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, (batch, $M, K$) $\times$ (batch, $K, L$) is the first batch GEMM problem size, (batch, $M, L$) $\times$ (batch, $L, N$) is the second batch GEMM problem size. In Table V, the first convolution problem size is (batch, $IC, H, W$) $\times$ ($OC_1, IC, k_1, k_1$), and the second convolution problem size is (batch, $OC_1, [H/st_1, [W/st_1]] \times (OC_2, OC_1, k_2, k_2)$. $st_1$ 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 $\times$ 3) are more complicated than batch GEMM. The sliding windows of 3 $\times$ 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

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

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

<table>
<thead>
<tr>
<th>Name</th>
<th>$IC$</th>
<th>$H$</th>
<th>$W$</th>
<th>$OC_1$</th>
<th>$OC_2$</th>
<th>$st_1$</th>
<th>$st_2$</th>
<th>$k_1$</th>
<th>$k_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>64</td>
<td>112</td>
<td>112</td>
<td>192</td>
<td>128</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>C2</td>
<td>32</td>
<td>147</td>
<td>147</td>
<td>64</td>
<td>80</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>C3</td>
<td>64</td>
<td>56</td>
<td>56</td>
<td>128</td>
<td>64</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>C4</td>
<td>128</td>
<td>28</td>
<td>28</td>
<td>256</td>
<td>128</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>C5</td>
<td>16</td>
<td>227</td>
<td>227</td>
<td>64</td>
<td>16</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>C6</td>
<td>64</td>
<td>56</td>
<td>56</td>
<td>64</td>
<td>64</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>C7</td>
<td>64</td>
<td>56</td>
<td>56</td>
<td>64</td>
<td>64</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C8</td>
<td>256</td>
<td>56</td>
<td>56</td>
<td>256</td>
<td>64</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
chains with ReLU. The speedups are in line with those of fusing two convolutions (2.87× to Pytorch, 2.30× to Relay, and 1.71× 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× to PyTorch, 3.30× to TASO, 1.69× to Relay, 1.33× to Ansor, and 2.29× 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×. 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×. 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× to Relay and 1.64× to Ansor.

For fusing convolution and convolution (Figure 6 part c), the average speedups to PyTorch and TensorRT are 5.79× and 2.01×. 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×; the average speedup to Ansor is 1.30×.

**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× speedup to TBE on average. The average speedup to AKG is 1.14×. 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.
Chimera’s optimization is much faster than Ansor (21.89x on average) and achieves 1.39x 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 2.37x speedup, fusion techniques can bring 1.89x speedup, and micro kernel can bring 1.61x 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 Ansoc [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**
