OptiFX: Automatic Optimization for Convolutional Neural Networks with Aggressive Operator Fusion on GPUs[*][†]

XUEYING WANG and SHIGANG LI, Beijing University of Posts and Telecommunications, China
HAO QIAN, University of New South Wales, Australia
FAN LUO and ZHAOYANG HAO, Institute of Computing Technology, Chinese Academy of Sciences, China and University of Chinese Academy of Sciences, China
TONG WU, Beijing University of Posts and Telecommunications, China
RUIYUAN XU, HUIMIN CUI, and XIAOBING FENG, Institute of Computing Technology, Chinese Academy of Sciences, China and University of Chinese Academy of Sciences, China
GUANGLI LI, Institute of Computing Technology, Chinese Academy of Sciences, China, University of Chinese Academy of Sciences, China, and University of New South Wales, Australia
JINGLING XUE, University of New South Wales, Australia

Convolutional Neural Networks (CNNs) are fundamental to advancing computer vision technologies. As CNNs become more complex and larger, optimizing model inference remains a critical challenge in both industry and academia. On modern GPU platforms, CNN operators are typically memory-bound, leading to significant performance degradation due to memory wall effects. While recent advancements have utilized operator fusion—merging multiple operators into one—to enhance inference performance, the fusion of multiple region-based operators like convolution is seldom addressed. This paper introduces AFusion, a novel operator fusion technique aimed at improving inference performance, and OptiFX, an automatic optimization framework based on this approach. OptiFX employs a cost-based backtracking search to identify optimal sub-graphs for fusion and utilizes template-based code generation to create efficient kernels for these fused sub-graphs. We evaluate OptiFX across seven prominent CNN architectures—GoogLeNet, ResNet, DenseNet, MobileNet, SqueezeNet, NasNet, and UNet—on Nvidia A6000 Ada, RTX 4090, and Jetson AGX Orin platforms. Our results demonstrate that OptiFX significantly outperforms existing methods, achieving average speedups of 2.91×, 3.30×, and 2.09× in accelerating inference performance on these platforms, respectively.

CCS Concepts: • **Software and its engineering → Compilers**; • **Computing methodologies → Parallel computing methodologies**; • **Computer systems organization →** *Neural networks.*

Additional Key Words and Phrases: Deep Learning Systems, Convolutional Neural Networks, Operator Fusion

[*]New Paper, Not an Extension of a Conference Paper.
[†]Shigang Li (lishigang@bupt.edu.cn) and Guangli Li (liguangli@ict.ac.cn) are the corresponding authors of this paper.

Authors' addresses: Xueying Wang, wangxueying@bupt.edu.cn; Shigang Li, lishigang@bupt.edu.cn, Beijing University of Posts and Telecommunications, 10 Xitucheng Road, Haidian District, Beijing, China, 100876; Hao Qian, hao.qian@unsw.edu.au, University of New South Wales, NSW, Sydney, Australia, 2052; Fan Luo, luofan23z@ict.ac.cn; Zhaoyang Hao, haozhaoyang24s@ict.ac.cn, State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, 6 Kexueyuan South Road, Haidian District, Beijing, China, 100190 and University of Chinese Academy of Sciences, 6 Kexueyuan South Road, Haidian District, Beijing, China, 100190; Tong Wu, 2023140627wt@bupt.edu.cn, Beijing University of Posts and Telecommunications, 10 Xitucheng Road, Haidian District, Beijing, China, 100876; Ruiyuan Xu, xuruiyuan23s@ict.ac.cn; Huimin Cui, cuihm@ict.ac.cn; Xiaobing Feng, fxb@ict.ac.cn, State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, 6 Kexueyuan South Road, Haidian District, Beijing, China, 100190 and University of Chinese Academy of Sciences, 6 Kexueyuan South Road, Haidian District, Beijing, China, 100190; Guangli Li, liguangli@ict.ac.cn, State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, 6 Kexueyuan South Road, Haidian District, Beijing, China, 100190 and University of Chinese Academy of Sciences, 6 Kexueyuan South Road, Haidian District, Beijing, China, 100190 and University of New South Wales, NSW, Sydney, Australia, 2052; Jingling Xue, j.xue@unsw.edu.au, University of New South Wales, NSW, Sydney, Australia, 2052.

1 INTRODUCTION

Convolutional Neural Networks (CNNs) are pivotal for intelligent applications such as image classification and object detection [9, 12, 13, 15]. As computer vision tasks become more complex, CNN architectures grow more sophisticated and models larger, resulting in increased computational demands. Consequently, optimizing model inference performance has become a critical challenge in developing efficient deep learning systems.

Current deep learning frameworks, such as PyTorch [31] and TensorFlow [1], typically model a neural network as a computational graph and associate graph operators with executable kernels. In comparison to vendor-specific libraries like cuDNN [7] on Nvidia GPUs and oneDNN [25] on Intel CPUs, deep learning compilers [5, 22, 30, 38, 45] offer more versatile options for graph-level optimizations and the generation of high-performance kernels across various computing platforms. A key technique is operator fusion [17, 28, 47, 53], which merges small operators into a larger one, reducing data movement and memory usage, as seen in FlashAttention [8].

Recent advancements have enhanced model inference through vertical fusion [14, 28, 53] and horizontal fusion [17, 26, 47] methods, as depicted in Figure 1a–c using an Inception module sub-graph from GoogLeNet [36]. Vertical operator fusion (VFusion) minimizes data movement by directly passing outputs between operators, optimizing high-level cache utilization. In contrast, horizontal operator fusion (HFusion) increases kernel parallelism on multi-core platforms by merging underutilized operators. Specifically, VFusion combines region-based operators (ROPs) like convolution and pooling with element-wise operators (EOPs) like Bias and ReLU to form CBR operators (Figure 1a to Figure 1b), while HFusion merges multiple ROPs within the same region (Figure 1b to Figure 1c). However, configurations such as max pooling followed by a CBR operator can impede the vertical fusion of multiple ROPs, limiting further optimization.

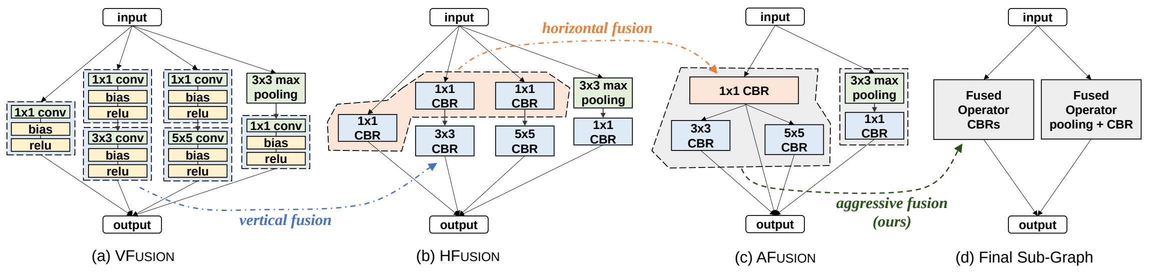


Fig. 1. An illustration of operator fusion techniques on the sub-graph of a GoogLeNet Inception structure. The dashed boxes in (a), (b), and (c) delineate the regions targeted for fusion using vertical, horizontal, and aggressive fusion techniques, respectively.

Due to the nature of region-based computation, vertically fusing one ROP with another can introduce extra computational overhead, even though it reduces data movement between main memory and cache [41]. Many ROPs in model inference are memory-bound, constrained by memory bandwidth. We propose that the additional computations from fusion can be offset during memory accesses when fusing these memory-bound ROPs vertically. Therefore, we have developed a technique called aggressive operator fusion (AFusion), illustrated in Figure 1c and Figure 1d, to enhance existing fusion methods like VFusion and HFusion. In our approach, we fuse three ROPs—a 1×1 CBR, a 3×3 CBR, and a 5×5 CBR—as well as a 3×3 max pooling and a 1×1 CBR operator. Contemporary CNNs feature complex computational graphs with numerous interconnected operators, posing significant challenges for kernel fusion. Therefore, AFusion builds upon VFusion and HFusion by enabling the fusion of ROPs that vary in shape—as demonstrated in Figure 1c and Figure 1d—or require multiple input or output tensors, as illustrated in Figure 7.

In this paper, we address a fundamental question in kernel fusion: *How can neural network model inference be effectively accelerated by automatically exploring optimization opportunities with aggressive operator fusion?* To answer this, we introduce OptiFX, an automatic model optimization framework based on operator fusion (Optimization with Fusion eXploration). We first use the Roofline model to identify computation- or memory-bound operators in neural networks, pinpointing potential sub-graphs for optimization. We then automatically explore these sub-graphs using AFusion and a cost-based backtracking search, employing a template-based code generator to create high-performance kernels for the fused operators. We validated OptiFX using seven representative CNNs—GoogLeNet, ResNet, DenseNet, MobileNet, SqueezeNet, NasNet, and UNet—across three GPU platforms, confirming the efficacy of our framework

In summary, this paper makes the following three major contributions:

- We performed a systematic analysis of aggressive operator fusion's potential to accelerate model inference, using the Roofline model to identify optimization opportunities in convolutional neural networks.
- We developed OptiFX, an automatic model optimization framework that implements aggressive fusion exploration with a cost-based backtracking search and utilizes a template-based code generator to produce high-performance kernels for fused sub-graphs.
- We evaluated OptiFX on seven real-world CNN models across three GPU platforms, achieving inference speedups over existing methods ranging from 1.47× to 7.42× on the Nvidia A6000 Ada, 1.49× to 6.28× on the Nvidia RTX 4090, and 1.31× to 4.37× on the Nvidia Jetson AGX Orin.

The remainder of this paper is organized as follows. Section 2 provides background and motivation for our fusion optimization. Section 3 details our aggressive fusion method, while Section 4 describes the OptiFX optimization framework. Section 5 presents experimental results, including performance analyses of end-to-end model and sub-graph inference. Section 6 discusses future work and implications, and Section 8 concludes the paper.

2 BACKGROUND AND MOTIVATION

We highlight the opportunities and necessity for fusing memory-bound operators in CNN models (Section 2.1) and then motivate our aggressive fusion approach to mitigate memory-wall effects (Section 2.2).

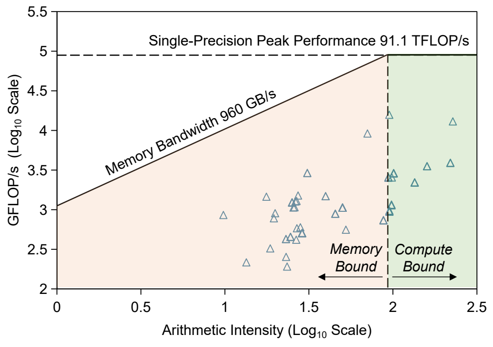
2.1 Performance Characteristics of CNN Operators on GPU Platforms

To pinpoint performance bottlenecks in a program, we use the classic Roofline model [42]. We rely on arithmetic intensity, denoted as $\mathcal{I}$, as the primary metric:

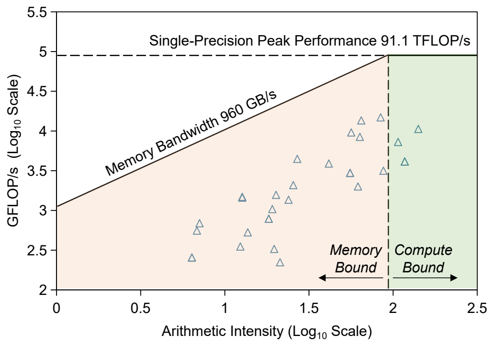
$$\mathcal{I} = \frac{\text{Number of FLOPs}}{\text{Number of Memory Operations}} \tag{1}$$

where the numerator represents the total number of floating-point operations (FLOPs), and the denominator accounts for total memory operations (in bytes), including memory access and data movement. For a given computing platform, the maximum arithmetic intensity is expressed as $\mathcal{I}_{\mathsf{max}} = \mathcal{P}_{\mathsf{max}}/\mathcal{M}_{\mathsf{max}}$, where $\mathcal{P}_{\mathsf{max}}$ signifies the peak performance and $\mathcal{M}_{\mathsf{max}}$ denotes the peak memory bandwidth. The attainable performance ceiling, $\widetilde{\mathcal{P}}$, is determined by:

$$\widetilde{\mathcal{P}} = \begin{cases} \mathcal{M}_{\mathsf{max}} \cdot \mathcal{I}, & \text{if } \mathcal{I} < \mathcal{I}_{\mathsf{max}} \\ \mathcal{P}_{\mathsf{max}}, & \text{if } \mathcal{I} \geq \mathcal{I}_{\mathsf{max}} \end{cases} \tag{2}$$



(a) GoogLeNet



(b) SqueezeNet

Fig. 2. Operator performance in GoogLeNet and SqueezeNet using the Roofline model on the Nvidia A6000 Ada platform.

This equation illustrates that the performance of a program on a given platform is dominated by memory resources (i.e., *memory-bound*) when $\mathcal{I} < \mathcal{I}_{\mathtt{max}}$, and by computational resources (i.e., *compute-bound*) when $\mathcal{I} \geqslant \mathcal{I}_{\mathtt{max}}$. Consequently, computational resources are under-utilized when running memory-bound programs.

In contemporary CNN models, convolutional operators are among the most critical, dominating model inference performance. Therefore, we focus on these operators under the Roofline model to identify performance bottlenecks. The arithmetic intensity of a single convolutional operator in FP32 (four bytes per element) can be calculated as follows:

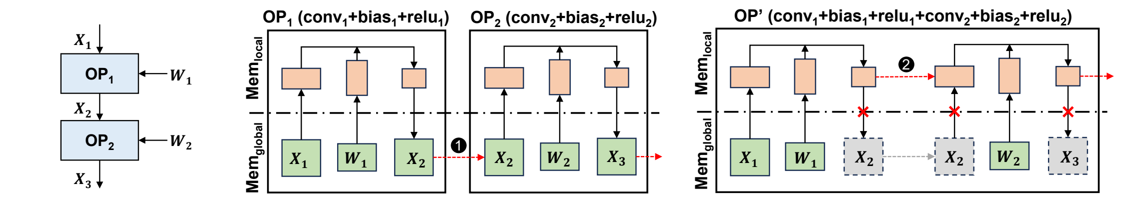
$$\mathcal{I}_{\mathsf{conv}} = \frac{2 \times (N \times K \times H' \times W') \times (C \times R \times S)/G}{4 \times (N \times C \times H \times W + C \times K \times R \times S + N \times K \times H' \times W')} \tag{3}$$

Here, N is the batch size; $H \times W$ and $H' \times W'$ are the input and output dimensions; C is the number of input channels; K is the number of filters (or output channels); $R \times S$ are the filter dimensions; and G is the number of groups in separable convolutions. The factor 2 in the numerator accounts for both multiplication and addition operations performed by the convolution operator, while the factor 4 in the denominator reflects the four bytes per data element accessed.

Figure 2 illustrates the performance of all convolution operators in GoogLeNet [36] and SqueezeNet [16] using the Roofline model on the Nvidia A6000 Ada platform. This platform provides a global memory bandwidth of 960 GB/s and a peak performance of 91.1 TFLOP/s, with $\mathcal{I}_{\mathtt{max}} = 94.90$. The Roofline model shows the relationship between computational complexity and arithmetic intensity, where each point represents a single convolution operator. Operators are categorized into two groups: those with a low compute-to-memory ratio, limited by memory bandwidth, and those with a high ratio, limited by computational power. In the model, memory-bound operators appear in the light red region on the left, while compute-bound operators are in the light green region on the right. GoogLeNet contains 61 memory-bound and 33 compute-bound convolution operators, respectively. For SqueezeNet, these numbers are 26 and 4. Given the prevalence of memory-bound convolution operators in CNNs, which often face memory wall effects leading to bottlenecks, optimizing these operators is crucial for enhancing CNN performance.

2.2 Mitigating Memory Wall Effects with AFusion

The performance of model inference on modern GPUs is often constrained by "memory wall" effects, as many operators are memory-bound due to the growing disparity between processing units and memory access bandwidth. Operator fusion is a key DNN acceleration technique that minimizes data movement across operators, thereby increasing arithmetic intensity and improving performance. As shown in Figure 1, traditional methods primarily use VFusion and


X_1
OP_1
W_1
X_2
OP_2
W_2
X_3
OP_1 (conv$_1$+bias$_1$+relu$_1$)
OP_2 (conv$_2$+bias$_2$+relu$_2$)
OP' (conv$_1$+bias$_1$+relu$_1$+conv$_2$+bias$_2$+relu$_2$)
Mem$_{local}$
Mem$_{global}$
❶
❷

(a) Two-OP Structure (b) Data movement without AFusion (c) Data movement with AFusion

Fig. 3. An example of optimizing data movement for a sub-graph containing two CBR operators.

HFusion to transform an original sub-graph $\mathcal{G}_{\texttt{naive}}$ into an optimized sub-graph $\mathcal{G}_{\texttt{opt}}$. However, we will demonstrate that by strategically applying our new fusion approach, AFusion, we can further optimize $\mathcal{G}_{\texttt{opt}}$.

Figure 3 illustrates how AFusion optimizes data movement for a sub-graph consisting of two optimized CBR operators: $\texttt{OP}_1$ (comprising $\texttt{conv}_1$, $\texttt{bias}_1$, and $\texttt{relu}_1$) and $\texttt{OP}_2$ (comprising $\texttt{conv}_2$, $\texttt{bias}_2$, and $\texttt{relu}_2$). In Figure 3a, this sub-graph, denoted as $\mathcal{G}_{\texttt{opt}}$, has already been enhanced using existing operator fusion methods like VFusion and HFusion [5, 17, 18, 30]. Figure 3b depicts the data movement during the execution of $\mathcal{G}_{\texttt{opt}}$. Initially, the input X_1 is transferred from $\texttt{Mem}_{\texttt{global}}$ to the high-speed cache $\texttt{Mem}_{\texttt{local}}$. After processing by $\texttt{OP}_1$, the intermediate result X_2 is stored back to $\texttt{Mem}_{\texttt{global}}$. Subsequently, X_2 must be reloaded into $\texttt{Mem}_{\texttt{local}}$ for $\texttt{OP}_2$, incurring significant costs—especially when the operators are memory-bound (❶). To minimize this overhead, we will refine $\mathcal{G}_{\texttt{opt}}$ into $\mathcal{G}'_{\texttt{opt}}$ using AFusion, as shown in Figure 3c. In this configuration, $\texttt{OP}_1$ and $\texttt{OP}_2$ are merged into a single operator $\texttt{OP}'$, allowing the intermediate result X_2 to remain in $\texttt{Mem}_{\texttt{local}}$ (❷) for immediate reuse by $\texttt{OP}_2$.

To demonstrate the efficacy of AFusion, we present two examples from real-world CNN models: one from GoogLeNet and one from SqueezeNet. Figure 4 shows the sub-graph structure (left) and corresponding Roofline performance (right) for each example on the Nvidia A6000 Ada platform. In these sub-graphs, each `OP` represents a fused operator combining `conv`, `bias`, and `relu`, annotated with the filter size of its convolution operation. The shapes of input/output tensors are displayed, where C denotes the number of channels, and H and W are the height and width, respectively.

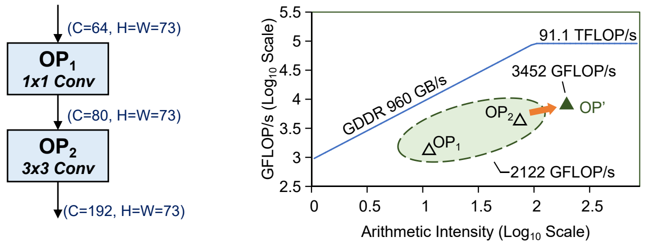
Figure 4a illustrates applying AFusion to optimize a two-operator sub-graph in GoogLeNet, where $\texttt{OP}_1$ and $\texttt{OP}_2$ are memory-bound. Originally operating at 2122 GFLOP/s, merging these operators into `OP'` using AFusion reduces memory data movement (as illustrated in Figure 3), enhancing performance to 3452 GFLOP/s.

Figure 4b presents the other example, optimizing a three-operator sub-graph in SqueezeNet, where $\texttt{OP}_1$, $\texttt{OP}_2$, and $\texttt{OP}_3$ are all memory-bound. Initially, $\texttt{OP}_1$'s output serves both $\texttt{OP}_2$ and $\texttt{OP}_3$, causing significant data movement and limiting performance to only 1174 GFLOP/s. By merging all three operators into `OP'` using AFusion, the sub-graph becomes compute-bound, boosting performance significantly to 2238 GFLOP/s.

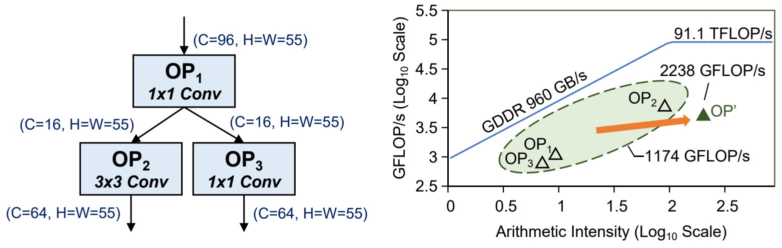
These two examples highlight the potential of AFusion to mitigate memory wall effects and enhance the performance of CNN models. However, given the increasing complexity of modern CNN architectures, determining the most effective way to apply AFusion is not straightforward. *What is needed is a strategy that can automatically explore the computational graph of any given CNN model using AFusion to optimize its performance.*

3 AGGRESSIVE OPERATOR FUSION

We introduce AFusion, an aggressive operator fusion approach capable of fusing two or more ROPs that vary in shape or require multiple input or output tensors, and analyze its potential to enhance model performance. Starting with a single-input, single-output fusion strategy, we extend it to a multi-input/output approach to accommodate more complex



(a) Fusing a two-operator structure in GoogLeNet



(b) Fusing a three-operator structure in SqueezeNet

Fig. 4. Two optimizing examples with AFusion under the Rooline model on the Nvidia A6000 Ada platform.

sub-graph structures in real-world CNN models (Section 3.1). While AFusion reduces memory operations, it may require additional redundant computations to maintain data dependencies among fused ROPs. We provide a theoretical analysis of these redundant computations to inform the optimization and exploration of AFusion (Section 3.2).

3.1 Operator Fusion Strategies

3.1.1 Single-Input Single-Output Operators. AFusion operates in a single-input, single-output (SISO) scenario by fusing a two-ROP sub-graph, $\mathtt{OP}_1$ and $\mathtt{OP}_2$. As shown in Figure 5, X_1 enters $\mathtt{OP}_1$, producing X_2, which then feeds into $\mathtt{OP}_2$ to yield X_3. Our approach enables retaining X_2 in high-speed cache for direct reuse, thereby enhancing efficiency. In practice, our SISO fusion strategy is widely utilized as it aligns with the prevalent structures in modern CNNs.

ROPs, such as convolution and pooling operators, process input tensors in tiles, producing corresponding tiled results [5, 28, 48]. Typically, an ROP divides an input tensor into multiple independent, homogeneous, tile-based tasks executed in parallel, with tile size being a critical hyper-parameter. Each task, handled by a single processing element (PE), e.g., an SM on GPUs, retrieves a tile from $\mathtt{Mem}_{\mathtt{global}}$ to $\mathtt{Mem}_{\mathtt{local}}$ (registers and shared memory), performs the required computation prescribed by the task, and writes the output tile back to $\mathtt{Mem}_{\mathtt{global}}$.

In the SISO configuration shown in Figure 5, tiles from tensors X_1, X_2, and X_3 are labeled x_1, x_2, and x_3 in green, blue, and orange, respectively. The tile-based task $\mathtt{OP}_1$ processes x_1 to produce x_2, which is then used by $\mathtt{OP}_2$ to generate x_3. AFusion aims to keep x_2 in $\mathtt{Mem}_{\mathtt{local}}$, eliminating its transfer to and from $\mathtt{Mem}_{\mathtt{global}}$ and enabling direct reuse by $\mathtt{OP}_2$. However, effectively fusing such tile-based tasks remains challenging.

To apply AFusion in a SISO configuration, input and output operator sizes must be related. For a convolution with a filter size $R \times S$ and a stride *stride*, the input tile size $h \times w$ is determined by the corresponding output tile size $h' \times w'$:

$$h \times w = \big((h' - 1) \times stride + R\big) \times \big((w' - 1) \times stride + S\big) \tag{4}$$

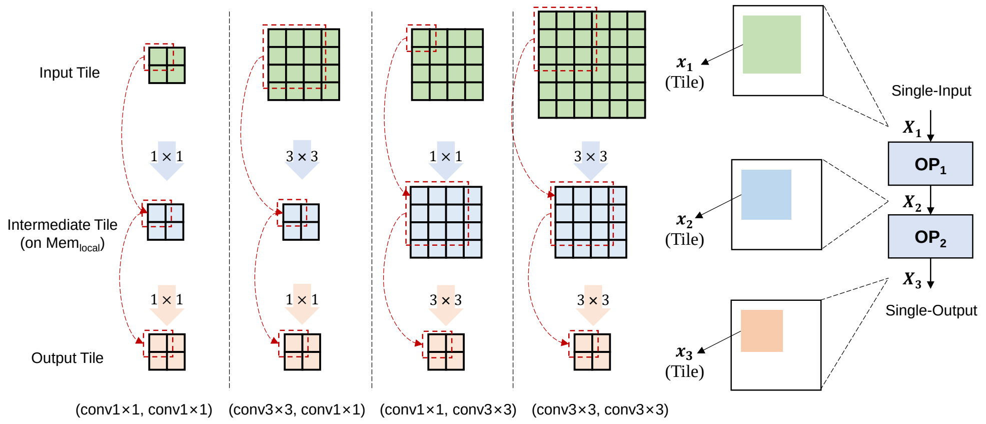


Fig. 5. Examples of single-input, single-output operator fusion with varying filter sizes (stride = 1). For each tensor X_i, one tile x_i is displayed. The tile sizes for the three tensors are related according to the constraint specified in Equation 4.

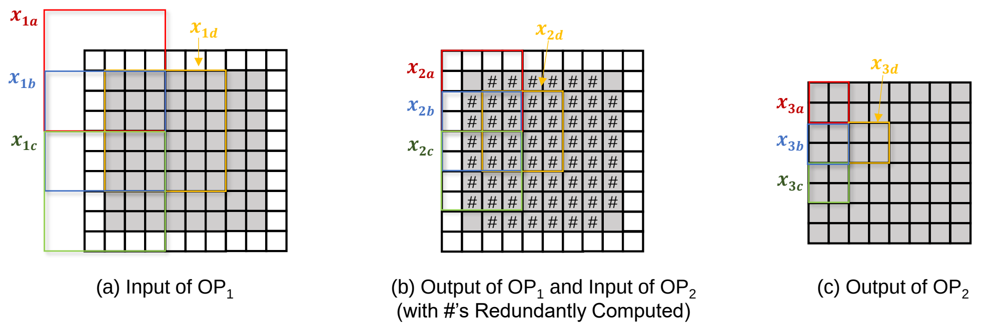


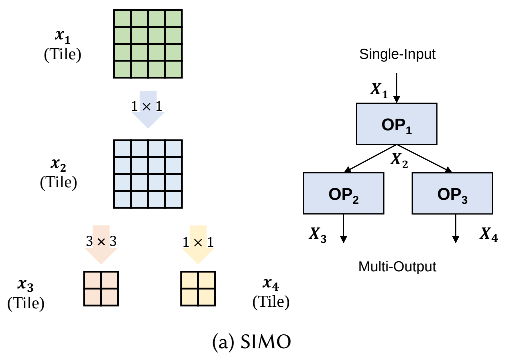
Fig. 6. Redundant computation operations introduced by AFusion for the fusion of $(\texttt{conv}_{3\times3}, \texttt{conv}_{3\times3})$ operators, as shown in Figure 5. The dimensions of X_1, X_2, and X_3 are depicted in gray cells, with padding shown in white cells.

Note that input tile sizes exceed output tile sizes for non-unity filter sizes ($R \times S > 1 \times 1$).

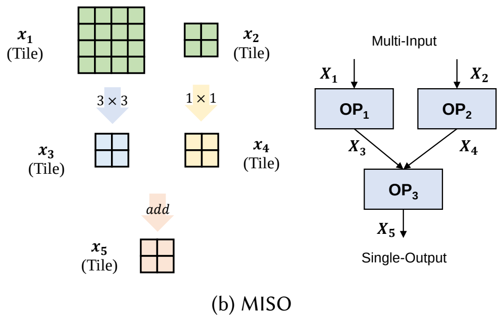
Figure 5 shows how to apply AFusion to fuse four different $\texttt{OP}_1$ and $\texttt{OP}_2$ pairs, including $(\texttt{conv}_{1\times1}, \texttt{conv}_{1\times1})$, $(\texttt{conv}_{3\times3}, \texttt{conv}_{1\times1})$, $(\texttt{conv}_{1\times1}, \texttt{conv}_{3\times3})$, and $(\texttt{conv}_{3\times3}, \texttt{conv}_{3\times3})$, assuming $stride = 1$. In each case, given that the tile size of x_3 is chosen to be 2×2, the sizes of x_2 and x_1 are determined recursively by applying Equation 4.

As AFusion aims to fuse a pair of operators, $\texttt{OP}_1$ and $\texttt{OP}_2$, at the tile level, it is crucial to consider the data dependencies among x_1, x_2, and x_3. Additionally, to ensure that x_2 produced by $\texttt{OP}_1$ and used by $\texttt{OP}_2$ can be available directly in $\texttt{Mem}_{\texttt{local}}$ without incur further memory operations, some redundant computations may be introduced.

Figure 6 illustrates redundant computations in the fusion of $(\texttt{OP}_1, \texttt{OP}_2) = (\texttt{conv}_{3\times3}, \texttt{conv}_{3\times3})$ operators from Figure 5. Initially, tensors X_1, X_2, and X_3 each have dimensions of 8×8 (shown in gray cells). After padding (shown in white cells), X_1 and X_2 expand to 10×10. For each tensor X_i, four tiles $(x_{ia}, x_{ib}, x_{ic}, x_{id})$ at the top-left corner are highlighted, leading to four tile-based tasks with tile sizes 2×2, 4×4, and 6×6 for X_1, X_2, and X_3, respectively. Each task T_t, where $t \in \{a, b, c, d\}$, processes x_{1t} as input (Figure 6a) to produce x_{3t} (Figure 6c) by sequentially applying $\texttt{OP}_1$ and $\texttt{OP}_2$ through an intermediate tile x_{2t} (Figure 6b) on a separate PE, such as an SM on GPUs. This process includes redundantly computing the boundaries of tile x_{2t} (marked by "#"), which are also processed by adjacent tasks. As a result, within X_1 and separately within X_2, adjacent tiles overlap. Retaining the intermediate tile x_{2t}, once produced by $\texttt{OP1}$, in $\texttt{Mem}_{\texttt{local}}$



(a) SIMO



(b) MISO

Fig. 7. Examples of multi-input/output operator fusion with varying filter sizes ($stride = 1$).

allows it to be directly reused by OP2, thus eliminating the need for costly memory operations via $\texttt{Mem}_{\texttt{global}}$ to transfer boundary results across PEs. AFusion enhances performance by leveraging these redundant computations to avoid costly memory transfers, optimizing memory-bound operators in contemporary CNNs.

3.1.2 Multi-Input/Output Operators.

In addition to SISO configurations, AFusion can optimize more complex setups involving three or more ROPs. Figure 7 demonstrates two examples of fusing three operators, $\texttt{OP}_1$ through $\texttt{OP}_3$, where x_i represents a tile of an input or output tensor for each operator as before. In Figure 7a, a Single-Input Multi-Output (SIMO) structure is used where $\texttt{OP}_1$ and $\texttt{OP}_2$ are 1×1 convolution operators, and $\texttt{OP}_3$ is a 3×3 convolution operator. $\texttt{OP}_1$ processes a single tensor X_1 into an output tensor X_2, which then feeds both $\texttt{OP}_2$ and $\texttt{OP}_3$. If x_3 and x_4 are each sized 2×2, then x_1 and x_2 would accordingly be 4×4, following Equation 4 (as x_2 depends on both x_3 and x_4).

Figure 7b shows a Multi-Input Single-Output (MISO) structure where $\texttt{OP}_1$ is a 3×3 convolution operator and $\texttt{OP}_2$ is a 1×1 convolution operator. The outputs of $\texttt{OP}_1$ and $\texttt{OP}_2$ are combined by $\texttt{OP}_3$. If x_5 is sized 2×2, then x_3 and x_4 are also 2×2, while x_1 and x_2 are 4×4 and 2×2, respectively. These SIMO and MISO strategies enhance AFusion's capability to optimize CNNs with complex multi-branch structures, such as those in SqueezeNet.

Similar multi-input/output fusion strategies can be developed to extend this approach.

3.2 Trade-off Between Memory and Computation Operations

Let us explore the trade-off AFusion makes by performing redundant computations to avoid costly memory operations, as demonstrated with the fusion of $(\texttt{conv}_{3\times3}, \texttt{conv}_{3\times3})$ operators in Figure 6. We have updated Figure 6b in Figure 8, replacing each "#" with an integer to indicate how many times each computation is redundantly performed. Here, X_2 is the intermediate tensor, initially $H \times W = 8 \times 8$. After padding, its dimensions expand to 10×10.

As discussed in Section 3.1, X_2 produced by $\texttt{OP}_1$ serves as the input for $\texttt{OP}_2$, an $R \times S$ convolution operator with a stride of $stride$. Let T_H and T_W represent the number of tiles derived from dimensions H and W of X_2, respectively. According to Equation 4, $\varphi_r = R - stride$ and $\varphi_s = S - stride$ define the boundary overlaps that contain redundant computations due to tile overlapping (Figure 6b). The total number of redundant computations, T_{red}, resulting from tile overlaps along both dimensions, can be exactly calculated as follows:

$$T_{\text{red}} = (T_H - 1) \times \varphi_r \times W + (T_W - 1) \times \varphi_s \times H + (T_W - 1) \times (T_H - 1) \times \varphi_r \times \varphi_s$$

In Figure 8, with $H = W = 8$, $T_H = T_W = 4$, and $\varphi_r = \varphi_s = 2$, we calculate $T_{\text{red}} = 132$. Note that smaller tile sizes, as used here for illustration purposes, yield a relatively higher number of redundant computations.

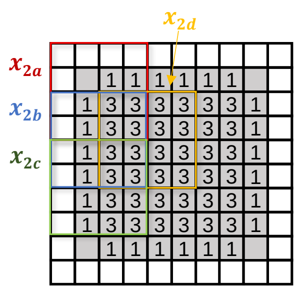
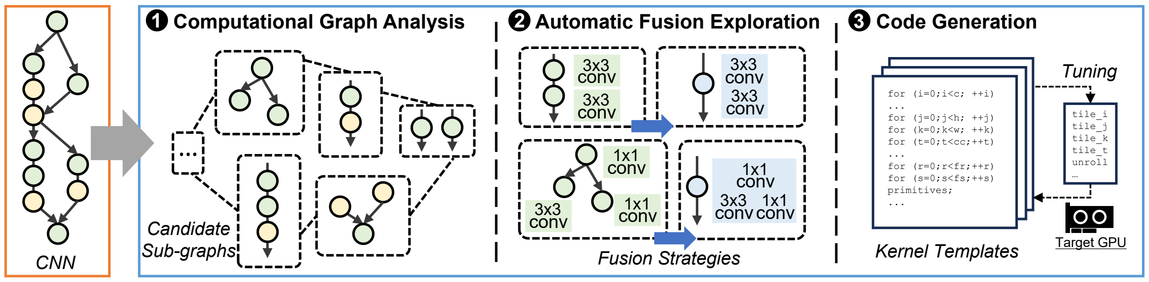


Fig. 8. Illustration of redundant computations from Figure 6, showing each "#" replaced by the frequency of redundant computations.

AFusion eliminates costly memory operations required for tasks to retrieve boundary results from adjacent tasks by introducing additional computational operations. This creates a trade-off between memory usage and computational resources in our aggressive fusion optimization strategy. To effectively identify and analyze fusion opportunities, we employ the Roofline model (Equation 2) to evaluate the benefits of fusion, considering both the reduction in memory operations and the increase in computational demands.

4 DESIGN OF OPTIFX

We have developed OptiFX, an automated model optimization framework that explores fusion opportunities within CNN models to reduce inference times. Figure 9 provides an overview of OptiFX, which consists of three main components: computational graph analysis, automatic fusion exploration, and code generation. OptiFX utilizes an arithmetic-intensity-aware computational graph (ACG), $\mathcal{G}_{\texttt{model}}$, where each node corresponds to an operator such as a convolution, and each edge represents a tensor. Nodes in the ACG include meta-information about arithmetic intensity, covering both computation and memory operations. This enhanced representation allows OptiFX to seamlessly optimize graphs from existing deep learning frameworks without requiring modifications to the models.



```
for (i=0;i<c; ++i)
...
for (j=0;j<h; ++j)
for (k=0;k<w; ++k)
for (t=0;t<cc;++t)
...
for (r=0;r<fr;++r)
for (s=0;s<fs;++s)
primitives;
...
```

```
tile_i
tile_j
tile_k
tile_t
unroll
…
```

Fig. 9. Overview of OptiFX.

The functionalities of OptiFX's three components are described below:

- **Computational Graph Analysis.** OptiFX analyzes the CNN model's graph, designated as $\bar{\mathcal{G}}_{\texttt{model}}$, and transforms it into an arithmetic-intensity-aware computational graph, $\mathcal{G}_{\texttt{model}}$. This transformation strictly follows the predefined operator specifications. Leveraging the meta-information related to computation and memory operations, OptiFX identifies and assembles a collection of candidate sub-graphs, denoted as $\mathcal{S}_{\texttt{model}} = \{\mathcal{G}_i \mid 1 \leq i \leq n\}$, where n quantifies the number of sub-graphs extracted from $\mathcal{G}_{\texttt{model}}$.

- **Automatic Fusion Exploration.** OptiFX utilizes a cost-based backtracking search to methodically explore the potential optimization paths offered by AFusion. It applies different fusion strategies to the candidate sub-graphs $\mathcal{S}_{\texttt{model}}$, assessing each sub-graph's cost by examining its arithmetic intensity and measured execution time.
- **Code Generation.** Upon identifying an optimal AFusion-determined sub-graph, OptiFX proceeds to generate a high-performance computational kernel. These kernels are crafted using predefined fusion templates and are implemented in CUDA C++ for Nvidia GPUs, ensuring optimization through techniques such as loop tiling, unrolling, and register allocation to maximize performance.

4.1 Computational Graph Analysis

OptiFX starts by transforming the original computational graph from deep learning frameworks, $\bar{\mathcal{G}}_{\texttt{model}}$, into an arithmetic-intensity-aware computational graph (ACG), $\mathcal{G}_{\texttt{model}}$. It analyzes the meta-information from each operator, including computation and memory operations, to calculate arithmetic intensity for each operator or sub-graph. Using a recursive graph split approach, OptiFX segments sub-graphs into smaller, disjoint sections to identify potential candidates for fusion. Concurrently, it employs the Roofline model for the target hardware to filter out less promising sub-graphs, optimizing the search space and focusing fusion exploration on the most beneficial sub-graphs.

Algorithm 1 describes OptiFX's cost-model-based approach for pruning candidate sub-graphs, beginning with the original graph $\bar{\mathcal{G}}_{\texttt{model}}$ and a hardware specification expressed in terms of $\mathcal{M}_{\texttt{max}}$ and $\mathcal{P}_{\texttt{max}}$. Initially, $\mathcal{S}_{\texttt{model}}$ is empty, and $\mathcal{G}_{\texttt{model}}$ is configured from $\bar{\mathcal{G}}_{\texttt{model}}$, populating necessary meta-information for each node (Line 1). The maximum arithmetic intensity $\mathcal{I}_{\texttt{max}}$ is calculated using $\mathcal{P}_{\texttt{max}}$ and $\mathcal{M}_{\texttt{max}}$, essential for the application of the Roofline model (Line 2).

For each node in $\mathcal{G}_{\texttt{model}}$, its meta-information from operator configurations is calculated to complete the ACG representation (Lines 3–4). The `GraphSplit` function recursively generates candidate sub-graphs. For each sub-graph below the threshold $T_{\texttt{graph}}$ (Line 6), the SimulateFusion() function is invoked to conduct a simulated fusion using VFusion, HFusion, and AFusion to produce a fused sub-graph $\mathcal{G}'$ (Line 7). SimulateFusion() operates like ApplyFusion() in Algorithm 2 (detailed in Section 4.2), returning a fused sub-graph, $\mathcal{G}'$, with optimal arithmetic intensity estimated by Equation 2. This avoids the need to compile and tune each potential fused sub-graph, as required by ApplyFusion(). The arithmetic intensities of both $\mathcal{G}$ and $\mathcal{G}'$ are calculated (Line 8) and their performance potential is estimated using the Roofline model from Equation 2 (Line 9). If $\mathcal{G}'$'s performance, $\widetilde{\mathcal{P}}_{\mathcal{G}'}$, exceeds $(\theta_{\texttt{opt}} \times \widetilde{\mathcal{P}}_{\mathcal{G}})$—where $\theta_{\texttt{opt}}$ ($\theta_{\texttt{opt}} > 1$) is a performance trade-off hyper-parameter—$\mathcal{G}'$ is selected for further exploration (Lines 10–11).

For sub-graphs larger than $T_{\texttt{graph}}$, a minimum cut method divides $\mathcal{G}$ into two parts, $\mathcal{G}_1$ and $\mathcal{G}_2$, which are then independently processed with the `GraphSplit` function (Lines 12–14). This sequence generates a set of candidate sub-graphs, $\mathcal{S}_{\texttt{model}}$, prepared for further exploration (Lines 15–16).

4.2 Automatic Fusion Exploration

Existing methods have leveraged search processes [17, 18] and dynamic programming [28] to pinpoint optimal graphs within the search space. OptiFX enhances these approaches by implementing a cost-based backtracking search specifically tailored for AFusion, refining the search space using candidate sub-graphs generated by Algorithm 1. To evaluate the performance of optimized graphs, OptiFX utilizes a cost model that combines the theoretical maximum performance from the Roofline model with actual latency measurements on the target hardware.

Algorithm 2 outlines our backtracking search process, starting with the initial graph $\mathcal{G}_{\texttt{init}}$. This approach utilizes candidate sub-graphs from $\mathcal{S}_{\texttt{model}}$ to explore fusion strategies aimed at optimizing configurations. Initially, $\mathcal{G}_{\texttt{opt}}$ is set

Algorithm 1: Candidate Sub-Graph Generation

```
Input: $\bar{\mathcal{G}}_{\texttt{model}}$ (Original Model), $\mathcal{M}_{\texttt{max}}$ and $\mathcal{P}_{\texttt{max}}$ (Hardware Specification)
Output: $\mathcal{S}_{\texttt{model}}$ (Candidate Sub-Graphs)
1  $\mathcal{G}_{\texttt{model}} \leftarrow \bar{\mathcal{G}}_{\texttt{model}}, \mathcal{S}_{\texttt{model}} \leftarrow \emptyset$      ▷ Initialize $\mathcal{G}_{\texttt{model}}$ and $\mathcal{S}_{\texttt{model}}$
2  $\mathcal{I}_{\texttt{max}} = \mathcal{P}_{\texttt{max}} / \mathcal{M}_{\texttt{max}}$
3  for each node $\mathcal{N}$ in $\mathcal{G}_{\texttt{model}}$ do
4      Calculate and insert meta-information of $\mathcal{N}$      ▷ Insert meta-information
5  Function GraphSplit($\mathcal{G}$):
6      if $|\mathcal{G}| \leq T_{\texttt{graph}}$ then
7          $(\mathcal{G}', \mathcal{I}_{\mathcal{G}'}) \leftarrow$ SimulateFusion($\mathcal{G}$)      ▷ Generates a fused sub-graph with optimal $\mathcal{I}'_{\mathcal{G}}$
8          Compute $\mathcal{I}_{\mathcal{G}}$      ▷ Compute arithmetic intensity
9          Estimate $\widetilde{\mathcal{P}}_{\mathcal{G}}$ and $\widetilde{\mathcal{P}}_{\mathcal{G}'}$ according to Equation 2      ▷ Estimate upper-bound performance
10         if $\widetilde{\mathcal{P}}_{\mathcal{G}'} > (\theta_{\texttt{opt}} \times \widetilde{\mathcal{P}}_{\mathcal{G}})$ then return $\mathcal{G}$ ;
11         else return $\emptyset$ ;
12     else
13         $\{\mathcal{G}_1, \mathcal{G}_2\} \leftarrow$ MinimumCut-Split($\mathcal{G}$);      ▷ Split current sub-graph
14         return {GraphSplit($\mathcal{G}_1$), GraphSplit($\mathcal{G}_2$)}
15 $\mathcal{S}_{\texttt{model}} \leftarrow$ GraphSplit($\mathcal{G}_{\texttt{model}}$);
16 return $\mathcal{S}_{\texttt{model}}$;
```

Algorithm 2: Backtracking Search

```
Input: $\mathcal{G}_{\texttt{init}}$ (Initial Graph), $\mathcal{S}_{\texttt{model}}$ (Candidate Sub-Graphs)
Output: $\mathcal{G}_{\texttt{opt}}$ (Optimized Model Graph)
1  $\mathcal{G}_{\texttt{opt}} \leftarrow \mathcal{G}_{\texttt{init}}$      ▷ Initialize the best explored model graph
2  $\mathcal{L} = \{\mathcal{G}_{\texttt{init}}\}$      ▷ Priority queue containing graphs sorted by ascending computational costs
3  while $\mathcal{L} \neq \{\}$ do
4      $\mathcal{G} \leftarrow \mathcal{L}$.pop()      ▷ Current explored graph
5      foreach $\mathcal{G}_{\texttt{sub}} \in \mathcal{S}_{\texttt{model}}$ do
6          $\mathcal{G}' \leftarrow$ ApplyFusion($\mathcal{G}, \mathcal{G}_{\texttt{sub}}$)      ▷ Apply fusion strategies
7          if Cost($\mathcal{G}'$) < Cost($\mathcal{G}_{\texttt{opt}}$) then
8              $\mathcal{G}_{\texttt{opt}} \leftarrow \mathcal{G}'$      ▷ Record the best explored model graph
9          else if Cost($\mathcal{G}'$) < ($\theta_{\texttt{ex}}$ × Cost($\mathcal{G}_{\texttt{opt}}$)) then
10             $\mathcal{L}$.push($\mathcal{G}'$)
11 return $\mathcal{G}_{\texttt{opt}}$;
```

to $\mathcal{G}_{\texttt{init}}$ (Line 1). A priority queue $\mathcal{L}$, which begins with $\mathcal{G}_{\texttt{init}}$, manages the graphs pending exploration and prioritizes them based on their costs, specifically focusing on latency in increasing order (Line 2).

If $\mathcal{L}$ is not empty, the process retrieves the first graph $\mathcal{G}$ from the queue, and applies fusion strategies, including common fusion techniques like VFusion [5, 30] and HFusion [17, 18], and AFusion, to a matching sub-graph $\mathcal{G}_{\text{sub}}$ within $\mathcal{G}$, resulting in a new fused graph $\mathcal{G}'$ (Lines 3–6). In Line 6, ApplyFusion() uses a template-based approach to generate a high-performance, tile-based kernel by fusing $\mathcal{G}_{\text{sub}}$. To enhance kernel performance, we implement an auto-tuning approach to optimize tile sizes, reducing redundant computations and maximizing parallelism in the kernel. The best graph $\mathcal{G}_{\text{opt}}$ is then updated if $\mathcal{G}'$ presents a lower cost than $\mathcal{G}$ (Lines 7–8).

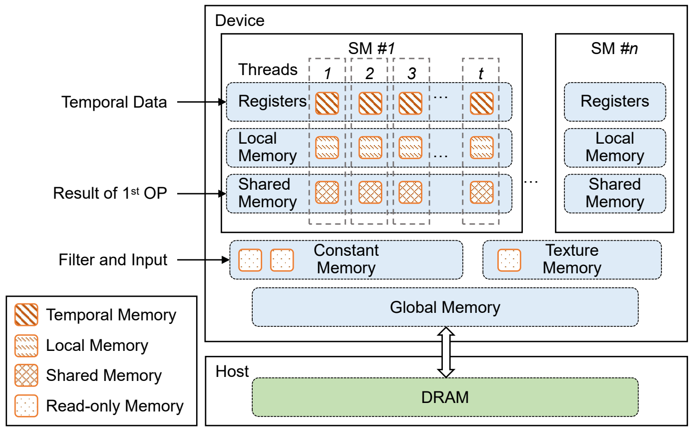


Fig. 10. Memory usage of fusion kernels on Nvidia GPU platforms.

A hyper-parameter θ_{ex} ($\theta_{\text{ex}} > 1$) sets a threshold, allowing only graphs $\mathcal{G}'$ with an extra cost up to θ_{ex} times that of the current optimal graph $\mathcal{G}_{\texttt{opt}}$ to be added to the queue $\mathcal{L}$ for further exploration (Lines 9–10). The process concludes with $\mathcal{G}_{\texttt{opt}}$ as the graph achieving optimal performance (Line 11).

4.3 Fusion Code Generation

OptiFX uses a template-based code generator to create high-performance GPU kernels for fused sub-graphs. Figure 10 illustrates the memory allocation for these kernels on Nvidia GPU platforms: temporal data is stored in registers, filters and read-only input tensors are loaded into constant memory, and intermediate results are placed in shared memory to promote reuse. Sub-graphs fused by AFusion are divided into tile-based tasks, each executed on an SM (Figure 6).

Figure 11 presents the pseudo-code for a tile-based task in a fused kernel for an SISO sub-graph consisting of two convolution operators, applicable to MISO and SIMO scenarios. Intermediate results from the first operator are stored in shared memory for reuse by the second operator, with a synchronization step to ensure data correctness. To boost kernel performance, we optimized the code to maximize on-chip data reuse. Fusion boundaries are automatically determined, and during index computation, we establish a mapping from "$\textit{input}[] \rightarrow \textit{shared\_mem}[] \rightarrow \textit{output}[]$", accounting for convolution padding and data offsets. These optimizations enhance efficient data management in on-chip memory.

In our kernel, we partition computations into tile-based tasks along the width, height, and batch dimensions, using a tile size of `tau_w` × `tau_h` × `tau_k`. On GPU platforms, each task executes on an SM using multiple parallel threads. We use identifiers (`g_x`, `g_y`, `g_z`) for tile-based tasks and (`b_x`, `b_y`, `b_z`) for threads within these tasks (Lines 2-3).

The first convolution operator, with filter dimensions `R_1` × `S_1`, assigns threads to compute several elements within their tile boundaries using the `tiled_bound()` function (Lines 4-6). Each thread initializes a temporary variable, performs a `R_1` × `S_1` convolution over all input channels `C_1`, and accumulates results in `t_1` (Lines 7-12). Input for this convolution is fetched from global memory, adjusted by an offset via `getInputOffset()`.

Bias and activation operations are performed in-place, and results are stored in shared memory using an offset from `getSharedStoreOffset()`, ready for use by the second convolution operator (Lines 13-16). To ensure consistency, thread synchronization occurs between the first and second convolution operations (Line 17).

For the second convolution operator with a filter size of `R_2` × `S_2`, threads retrieve necessary intermediate results directly from shared memory, minimizing data transfers that would otherwise be incurred from global memory (Lines

```
1  __device__ tile_based_task(input, output, filter_1, filter_2, bias_1, bias_2):
2    g_x, g_y, g_z = getGridIdx();
3    b_x, b_y, b_z = getBlockIdx();
4    for h in range(b_y, tiled_bound(H_1, tau_h), tau_h):
5      for w in range(b_x, tiled_bound(W_1, tau_w), tau_w):
6        for k in range(b_z, tiled_bound(K_1, tau_k), tau_k):
7          initialize(t_1);
8          for c in range(0, C_1, 1):
9            for r in range(0, R_1, 1):
10             for s in range(0, S_1, 1):
11               offset = getInputOffset(g_z, g_y, g_x, b_y, b_x, ...)
12               t_1 = conv(t_1, input[offset], filter_1[]);          ◁ convolution
13         t_1 += bias_1[];                                           ◁ bias
14         t_1 = activation(t_1);                                     ◁ activation
15         offset = getSharedStoreOffset(b_x, b_y, b_z, ...)
16         shared_mem[offset] = t_1;
17   __synctheads();
18   for h in range(b_y, bound(H_2), tau_h):
19     for w in range(b_x, bound(W_2), tau_w):
20       for k in range(b_z, bound(K_2), tau_k):
21         initialize(t_2);
22         for c in range(0, K_1, 1):
23           for r in range(0, R_2, 1):
24             for s in range(0, S_2, 1):
25               offset = getSharedLoadOffset(b_x, b_y, ...)
26               t_2 = conv(t_2, shared_mem[offset], filter_2[]);     ◁ convolution
27         t_2 += bias_2[];                                           ◁ bias
28         t_2 = activation(t_2);                                     ◁ activation
29         offset = getOutputOffset(g_z, g_y, g_x, b_y, b_x, b_z, ...)
30         output[offset] = t2;
```

Fig. 11. The pseudo-code for a tile-based kernel for SISO-pattern sub-graph.

18-25). Threads then perform a R_2 × S_2 convolution (Line 26), and post-process with bias and activation, finally storing results back to global memory with offsets calculated via `getOutputOffset()` (Lines 27-30).

The balance between shared memory usage and computational resource occupancy on GPUs is critical. The amount of shared memory used per block largely depends on the tile size, which should be sufficiently large to promote memory reuse within blocks and minimize redundant computations. However, increasing the tile size may reduce parallelism, though it could potentially enhance throughput per block. Furthermore, GPU hardware limits the maximum shared memory available. Part of this memory is allocated for storing intermediate results, constrained by the GPU's maximum memory capacity. In this paper, we empirically set the parameter search space of the tile size to $\{(\texttt{tau\_w}\times\texttt{tau\_h}\times\texttt{tau\_k})\}$ where $\texttt{tau\_w} \in \{2, 4, 8, 16\}$, $\texttt{tau\_w} \in \{2, 4, 8, 16\}$, $\texttt{tau\_k} \in \{2, 4, 8, 16\}$. To ensure optimal parallelism and active utilization of all blocks, we use an auto-tuning process to determine an appropriate tile size that balances performance with hardware constraints.

5 EVALUATION

We demonstrate that OptiFX outperforms several leading fusion frameworks in accelerating inference for a variety of CNN models and their sub-graphs across three GPU platforms. We also provide a comprehensive performance analysis to highlight the effectiveness of our aggressive fusion approach, AFusion. Finally, we discuss the limitations of OptiFX and suggest avenues for future research.

5.1 Experimental Setting

Hardware and Software Environments. We conducted experiments on three GPU platforms: Nvidia RTX 6000 Ada, Nvidia RTX 4090, and Nvidia Jetson AGX Orin, both operating under Linux-based systems. The A6000 Ada GPU exhibits a maximum arithmetic intensity of 94.90, supported by a peak performance of 91.1 TFLOP/s and a global memory bandwidth of 960 GB/s. The RTX 4090 GPU shows a maximum arithmetic intensity of 81.94, with a peak performance of 82.6 TFLOP/s and a global memory bandwidth of 1008 GB/s. The Jetson AGX Orin is an edge platform with a maximum arithmetic intensity of 25.98, which has 5.32 TFLOP/s peak performance and 204.8GB/s memory bandwidth. Our experiments employed the CUDA toolkit (version 12.1) and the cuDNN library (version 8.0.0). To reduce runtime system interference, we conducted each experiment 100 times and reported the average execution time.

Benchmarks. To validate the effectiveness of OptiFX, we evaluated it using seven representative real-world CNN models as benchmarks, including GoogLeNet [36], ResNet [12], DenseNet [15], MobileNet [13], SqueezeNet [16], NasNet [37], and UNet [34]. Notably, MobileNet and NasNet incorporate separable convolution operators, which are a variation from the conventional convolution operators used in the other models. Table 1 provides detailed specifications of these benchmarked models, including the number of repeated blocks and the number of operators in each model, highlighting the structural similarities and differences in terms of size and channel count.

Table 1. Benchmarked CNN Models (“conv”: conventional convolution operators; “sep-conv”: separable convolution operators).

CNN Model	Number of Blocks	Number of Operators	Convolution Type(s)
GoogLeNet [36]	11	138	conv
ResNet-50 [12]	16	70	conv
DenseNet [15]	3	250	conv
MobileNet [13]	17	62	conv, sep-conv
SqueezeNet [16]	10	50	conv
NasNet [37]	13	431	conv, sep-conv
UNet [34]	18	69	conv

5.2 End-to-End Model Performance

We assessed the end-to-end model inference performance of our OptiFX framework against several established frameworks, including TensorFlow [1] (version 2.16), TensorFlow-XLA [30], and TASO [17], all tested with a batch size of one unless otherwise specified. TensorFlow-XLA leverages the XLA compiler for rule-based graph optimizations, enhancing TensorFlow’s performance. TASO, a top-tier DNN optimization framework, performs graph substitutions for neural network optimization and is built on MetaFlow [18]. In addition, we compared the end-to-end performance of OptiFX with that of TensorRT [29] (version 10.6) and TorchInductor [2] (version 2.5.1).

In our experiments, we set three empirical thresholds for the two algorithms in Section 4: $T_{\texttt{graph}} = 3$, $\theta_{\texttt{opt}} = 1.05$ in Algorithm 1, and $\theta_{\texttt{ex}} = 1.1$ in Algorithm 2. The threshold $T_{\texttt{graph}} = 3$ is used to divide large sub-graphs into smaller ones, aligning with the maximum number of operators in our fusion patterns (SISO, SIMO, and MISO). We set $\theta_{\texttt{opt}} = 1.05$, slightly above 1.0, to filter out fused sub-graphs with trivial improvements in arithmetic intensity. The value $\theta_{\texttt{ex}} = 1.1$ is chosen to include sub-graphs that may not strictly reduce latency but allow a moderate expansion of the search space while ensuring the search process is completed within a reasonable time.

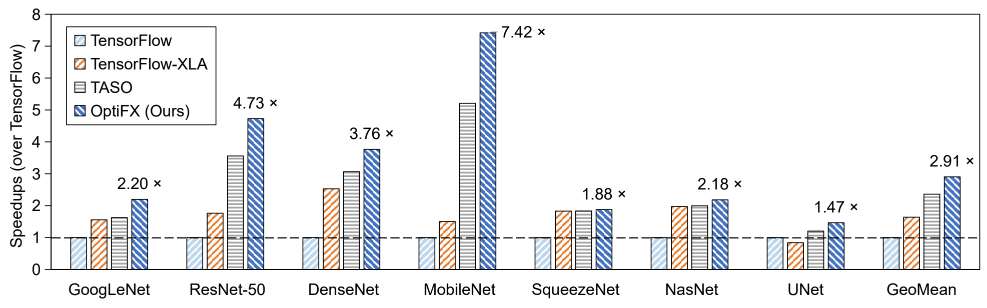


Fig. 12. End-to-end CNN performance on Nvidia A6000 Ada.

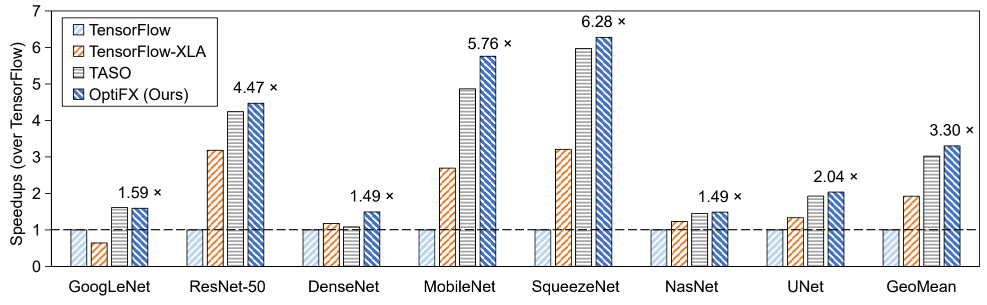


Fig. 13. End-to-end CNN performance on Nvidia RTX 4090.

To account for variations in hardware capabilities and fusion optimization strategies, we evaluated OptiFX's end-to-end model inference performance on three GPU platforms: Nvidia A6000 Ada, RTX 4090, and Jetson AGX Orin. This evaluation demonstrated OptiFX's effectiveness and adaptability across different settings.

Figure 12 presents the end-to-end model inference performance of seven CNN models on the Nvidia A6000 Ada GPU. It compares four frameworks with TensorFlow (without XLA) as the baseline. TensorFlow-XLA, which utilizes rule-based optimization, outperforms standard TensorFlow. TASO, an advanced graph-level optimization framework, generally surpasses TensorFlow-XLA. OptiFX delivers the highest overall performance, achieving a 2.91× speedup over TensorFlow, 1.77× over TensorFlow-XLA, and 1.23× over TASO (on average). OptiFX notably excels in optimizing MobileNet (7.42×) and ResNet-50 (4.73×) due to AFusion, a fusion approach not utilized by the other frameworks.

Figure 13 demonstrates OptiFX outperforming established frameworks on the RTX 4090 GPU, with average speedups of 3.30× over TensorFlow, 1.72× over TensorFlow-XLA, and 1.09× over TASO. OptiFX notably excels in SqueezeNet, benefiting from MISO fusion optimization, and MobileNet, due to its use of separable convolutions. Figure 14 presents these comparisons at a batch size of 8, where OptiFX continues to perform well, achieving speedups of 3.26×, 2.07×, and 1.16× over TensorFlow, TensorFlow-XLA, and TASO, respectively, affirming its efficacy in large-batch settings.

Figure 15 shifts focus to the Nvidia Jetson AGX Orin, an edge device with a peak performance of 5.32 TFLOP/s (FP32) and a memory bandwidth of 204.8 GB/s. Despite the typically lower memory bandwidth and processing power of edge devices compared to high-end GPUs, OptiFX effectively optimizes CNN models on such platforms. OptiFX outperforms the three baselines, achieving average speedups of 2.09×, 1.57×, and 1.23× over TensorFlow, TensorFlow-XLA, and TASO, respectively. Particularly, OptiFX shows the most significant improvement in MobileNet, a lightweight model.

Table 2 shows that OptiFX surpasses both TorchInductor [2] and TensorRT [29] across most CNN models on the Nvidia A6000 GPU, achieving average speedups of 2.49× over TorchInductor and 3.19× over TensorRT. However, for

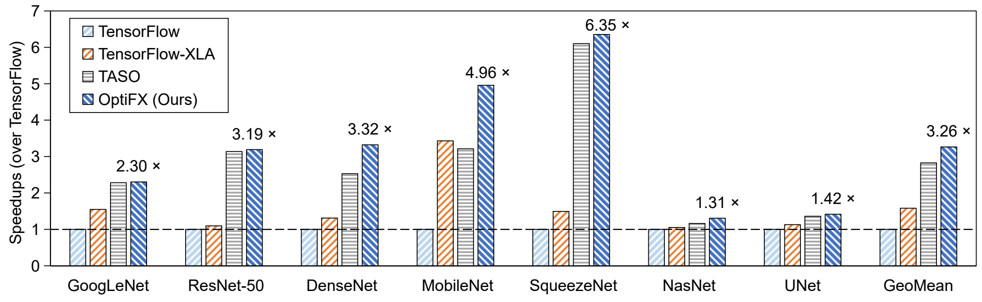


Fig. 14. End-to-end CNN performance on Nvidia RTX 4090 with a batch size of 8.

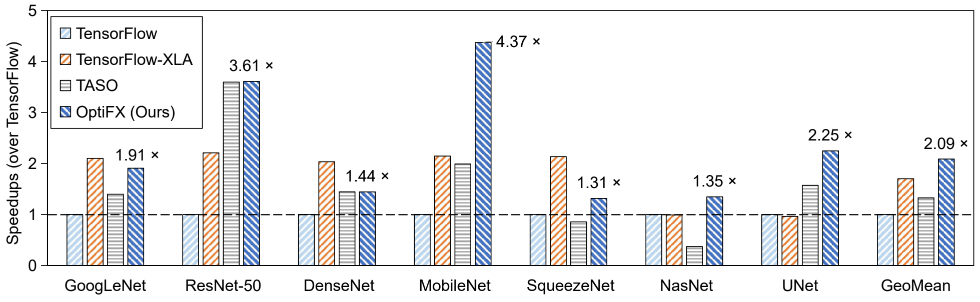


Fig. 15. End-to-end CNN performance on Nvidia Jetson AGX Orin.

Table 2. Speedups of OptiFX over TorchInductor and TensorRT on Nvidia A6000 Ada.

	GoogLeNet	ResNet-50	DenseNet	MobileNet	SqueezeNet	NasNet	UNet	GeoMean
TorchInductor	1.05×	1.90×	2.70×	2.51×	0.65×	27.87×	2.40×	2.49×
TensorRT	3.17×	3.68×	1.54×	1.81×	1.53×	7.50×	8.95×	3.19×

SqueezeNet, OptiFX underperforms TorchInductor due to the latter's use of highly optimized kernels generated by the Triton backend.

These findings validate OptiFX's capability to enhance inference performance for diverse modern CNN models across multiple GPU platforms, outperforming established frameworks.

5.3 Sub-Graph Performance

We assessed the inference performance enhancements offered by OptiFX, specifically through AFusion, by examining seven distinct sub-graphs selected from seven different CNN models. These sub-graphs, designated $\mathcal{G}_1$ through $\mathcal{G}_7$, are elaborated in Table 3. The description for each sub-graph includes the dimensions of the input tensor(s) ($H \times W$), with the output tensor dimensions derived using Equation 4. Additional details include the number of input channels (C), output channels (K), filter size ($R \times S$), and stride ($Stride$). The specific type of AFusion applied—whether SISO, SIMO, or MISO—is also specified for each sub-graph.

Table 4 presents experimental results for seven selected sub-graphs, G_1 through G_7, on the Nvidia A6000 Ada platform, comparing performance before and after applying AFusion. Data for each sub-graph includes arithmetic intensity $\mathcal{I}$, theoretical computational complexity, and actual latency. In particular, AFusion has effectively optimized sub-graphs G_1, G_3, G_5, and G_7, as detailed in Section 3. These sub-graphs, which could not be successfully optimized by

Table 3. Selected sub-graphs from seven different CNN models.

Sub-Graph	CNN Model	$H \times W$	$(C, K, R \times S, Stride)$	Type of AFusion Applied
$\mathcal{G}_1$	GoogLeNet	73×73	$(64, 80, 1 \times 1, 1)$ $(80, 192, 3 \times 3, 1)$	SISO (Figure 5c)
$\mathcal{G}_2$	ResNet	56×56	$(64, 64, 3 \times 3, 1)$ $(64, 64, 1 \times 1, 1)$	SISO (Figure 5b)
$\mathcal{G}_3$	DenseNet	32×32	$(60, 48, 1 \times 1, 1)$ $(48, 12, 3 \times 3, 1)$	SISO (Figure 5c)
$\mathcal{G}_4$	MobileNet	56×56	$(144, 144, 3 \times 3, 2)$ $(144, 32, 1 \times 1, 1)$	SISO (Figure 5b)
$\mathcal{G}_5$	SqueezeNet	55×55	$(128, 16, 1 \times 1, 1)$ $(16, 64, 3 \times 3, 1)$, $(16, 64, 1 \times 1, 1)$	SIMO (Figure 7a)
$\mathcal{G}_6$	NasNet	56×56	$(128, 128, 1 \times 1, 1)$, $(128, 128, 1 \times 1, 1)$ (element-wise add)	MISO (Figure 7b)
$\mathcal{G}_7$	UNet	56×56	$(16, 16, 3 \times 3, 2)$ $(16, 32, 1 \times 1, 1)$, $(16, 32, 3 \times 3, 1)$	SIMO (Figure 7a)

Table 4. Performance improvements achieved by AFusion on sub-graphs on Nvidia A6000 Ada.

Sub-Graph	Original Performance			Optimized Performance			Speedup
	$\mathcal{I}$	GFLOP/s	Latency (ms)	$\mathcal{I}$	GFLOP/s	Latency (ms)	
$\mathcal{G}_1$	157.09	2122.33	0.651	249.72	3452.09	0.401	1.62×
$\mathcal{G}_2$	118.34	248.58	0.986	145.53	1248.60	0.197	5.02×
$\mathcal{G}_3$	22.93	345.39	0.046	50.70	762.70	0.021	2.20×
$\mathcal{G}_4$	3.27	269.17	0.033	4.82	619.17	0.014	2.29×
$\mathcal{G}_5$	21.34	1194.10	0.057	23.89	2043.14	0.035	1.62×
$\mathcal{G}_6$	31.73	2307.61	0.086	42.67	2935.67	0.067	1.27×
$\mathcal{G}_7$	14.99	210.875	0.055	27.26	367.179	0.031	1.79×

previous methods [10, 17], demonstrate AFusion's unique capability. For these memory-bound operators, reducing memory operations significantly improves performance. These results emphasize AFusion's effectiveness in enhancing performance across various sub-graph structures in modern CNN models.

5.4 Performance Analysis

We performed multiple experiments to assess the performance characteristics of OptiFX and to demonstrate the effectiveness of our aggressive fusion approach, AFusion. Each figure or table presenting our results clearly specifies the GPU platform used (if relevant).

5.4.1 Kernel Launch Reductions in OptiFX Compared to TASO. We assessed the reduction in kernels launched by OptiFX compared to TASO across seven CNN models, as shown in Figure 16. Convolution operations, major components of CNN computations, are often fused with bias and activation functions, supported by library-level fused operators. OptiFX reduces kernel launches by an average of 10.95%, with the most significant reduction observed in SqueezeNet—22%. This improvement is primarily due to SqueezeNet's narrow channel configurations in the early output layers of its fire modules, which, due to AFusion, require minimal shared memory for storing intermediate results, thereby providing substantial optimization opportunities. Although TASO and OptiFX sometimes achieve similar reductions, OptiFX's vertical fusion strategy typically leads to better performance. By consolidating optimized sub-graphs into fewer kernels compared to TASO, OptiFX minimizes runtime and kernel launch overhead through AFusion. As a result, OptiFX outperforms TASO in accelerating model inference across seven CNN models, as detailed earlier in Section 5.2.

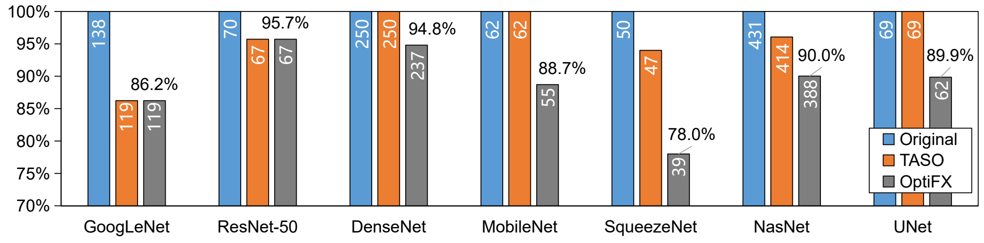


Fig. 16. Reduced kernel launches in OptiFX compared to the TASO baseline on Nvidia A6000 Ada.

Table 5. Effectiveness of OptiFX's cost-model-based pruning in selecting candidate sub-graphs for fusion.

	GoogLeNet	ResNet	DenseNet	MobileNet	SqueezeNet	NasNet	UNet
Original	190	88	384	120	208	158	136
Pruned	122	37	16	51	108	90	34
Reduction	64%	42%	4%	43%	52%	57%	25%
Consistent $\mathcal{G}_{\texttt{opt}}$ Post-Pruning	✓	✓	✓	✓	✓	✓	✓

5.4.2 OptiFX's Cost-Model-based Pruning for Efficient Exploration. OptiFX employs Algorithm 1 to evaluate all sub-graphs within a CNN model, narrowing down to a viable set of fusion candidates ($\mathcal{S}_{\texttt{model}}$) based on the Roofline cost model. This model theoretically assesses the acceleration potential of these sub-graphs, eliminating the need for on-device evaluation (Line 7). When optimizing a CNN, OptiFX uses Algorithm 2 to compile and evaluate only the selected sub-graphs from $\mathcal{S}_{\texttt{model}}$, which streamlines the compilation and tuning process (Line 6).

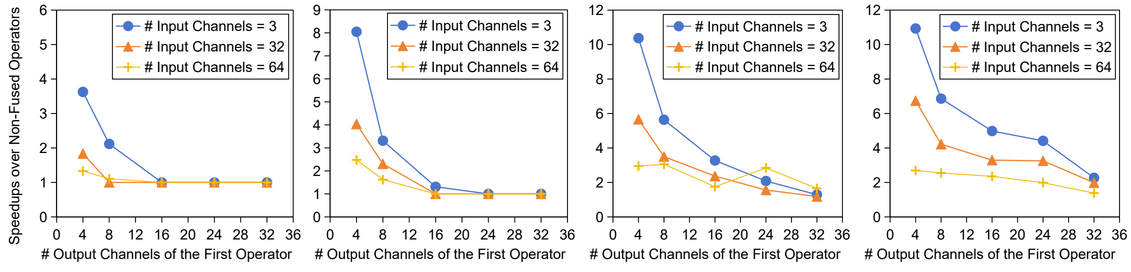
Table 5 illustrates that our cost-model-based pruning approach (Algorithm 1) effectively reduces the exploration space for sub-graphs, achieving a reduction of up to 64% across seven evaluated CNN models. Furthermore, OptiFX consistently generates the same optimal graph $\mathcal{G}_{\texttt{opt}}$ for each CNN model, regardless of whether it starts with the original or the selected sub-graphs in $\mathcal{S}_{\texttt{model}}$.

5.4.3 Impact of Input Shapes on OptiFX's Performance. We studied the impact of tensor shapes on inference performance enhancements when applying OptiFX to two distinct sub-graphs. Figure 17 illustrates an SISO fusion pattern with two vertically aligned operators that use 1×1 and 3×3 filters (Figure 5), highlighting the feasibility of fusion despite the presence of redundant computations. Figure 18 examines a fire-like module from SqueezeNet using an SIMO fusion pattern (Figure 7a). The performance improvements are assessed by varying the number of input channels and adjusting the output channels of the first operator, which also influence the input channels for subsequent operators.

Figures 17 and 18 show that AFusion significantly accelerates these two sub-graphs across different tensor configurations. Notably, sub-graphs with smaller input or output dimensions generally yield better speedups, as AFusion's performance is often constrained by the on-chip shared memory bandwidth under high memory pressure.

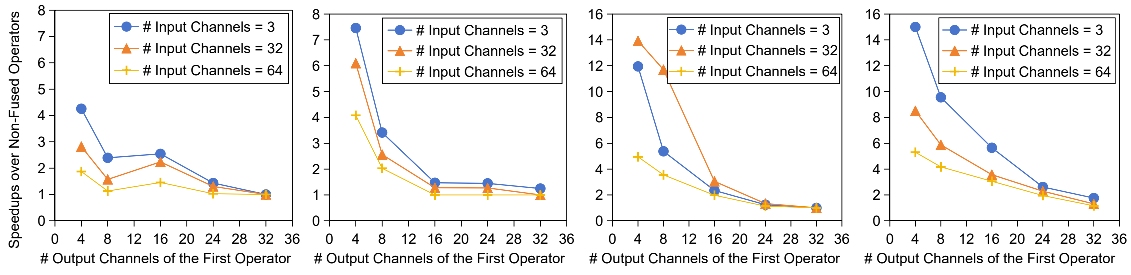
5.4.4 OptiFX's Performance under Different Batch Sizes. For the same SISO-pattern sub-graph discussed in Section 5.4.3 (Figure 17), Figure 19 displays the performance speedups AFusion achieves across four different batch sizes. We observe enhanced speedups at larger batch sizes, particularly with the largest input size (e.g., 224×224), attributable to improved parallelism and resource utilization. AFusion consistently enhances performance across diverse batch size settings.

5.4.5 OptiFX's Performance for Low-Precision Types. Our Roofline-based analysis is versatile, applicable to various precision types, including low-precision, demonstrating the extensive utility of AFusion. We illustrate this using an SISO-pattern sub-graph with two convolution operators: Conv1 ($C = 64, K = 48, R \times S = 1 \times 1, Stride = 1$) and Conv2



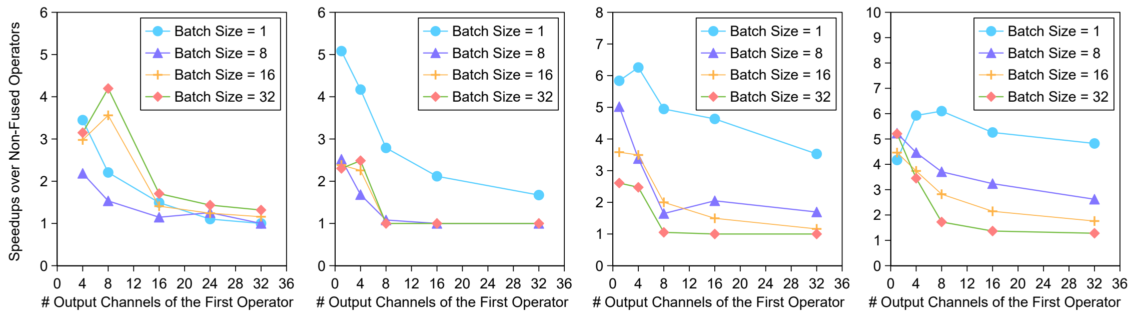
(a) Input Size = 224 × 224 (b) Input Size = 112 × 112 (c) Input Size = 56 × 56 (d) Input Size = 28 × 28

Fig. 17. Speedups achieved by OptiFX for an SISO fusion pattern across various tensor shapes on Nvidia A6000 Ada.



(a) Input Size = 224 × 224 (b) Input Size = 112 × 112 (c) Input Size = 56 × 56 (d) Input Size = 28 × 28

Fig. 18. Speedups achieved by OptiFX for an SIMO fusion pattern across various tensor shapes on Nvidia A6000 Ada.



(a) Input Size = 224 × 224 (b) Input Size = 112 × 112 (c) Input Size = 56 × 56 (d) Input Size = 28 × 28

Fig. 19. Speedups achieved by AFusion for the sub-graph used in Figure 17 with different batch sizes on Nvidia RTX 4090 GPU.

Table 6. Performance results achieved by AFusion for an SISO-pattern sub-graph under FP16 and FP32 on Nvidia RTX 4090.

	Original Graph		Fused Graph		Speedup
	Arithmetic Intensity	Time (ms)	Arithmetic Intensity	Time (ms)	
FP16	73.56	0.078	138.46	0.034	2.29×
FP32	36.98	0.082	70.31	0.043	1.89×

($C = 48, K = 12, R \times S = 3 \times 3, Stride = 1$), for an input shape ($N = 1, C = 64, H = 32, W = 32$). This analysis and performance improvement under the FP16 low-precision type are demonstrated on the Nvidia RTX 4090 GPU, which boasts a peak performance of 82.6 TFLOP/s and a peak bandwidth of 1008 GB/s.

Table 6 gives the results, detailing the arithmetic intensities of this sub-graph before and after applying AFusion, under FP16 precision, with FP32 results provided for comparison. Initially, the arithmetic intensity for the sub-graph stands at 73.56, with contributions from Conv1 at 46.79 and Conv2 at 100.33. Using the Roofline model to identify this sub-graph as a performance bottleneck, we then applied AFusion to fuse these two operators. This fusion increased the arithmetic intensity to 138.46. With AFusion, performance enhancements of 2.29× under FP16 and 1.89× under FP32 were achieved, confirming the applicability of our method across different precision levels in DNN models.

5.4.6 OptiFX's Compilation and Fusion Times. We evaluated the compilation and fusion times for OptiFX's offline process, as described in Algorithms 1 and 2. Table 7 presents these overheads across seven CNN models, categorized into "compilation," "search," and "others." "Compilation" includes compiling and tuning fused kernels (Line 6 of Algorithm 2), "search" involves backtracking to identify optimal sub-graph fusions (excluding Line 6), and "others" encompass additional system overheads (Algorithm 1).

Table 7. OptiFX's compilation and fusion times across CNN models on Nvidia RTX 4090.

	GoogLeNet	ResNet-50	DenseNet	MobileNet	SqueezeNet	NasNet	UNet
Search (%)	50.00%	9.09%	54.08%	15.38%	9.09%	73.92%	7.14%
Compilation (%)	34.00%	72.73%	27.55%	61.54%	72.73%	5.86%	64.29%
Others (%)	16.00%	18.18%	18.37%	23.08%	18.18%	20.22%	28.57%
Offline Time (s)	46.96	9.89	31.27	8.84	18.90	719.20	11.35

OptiFX typically requires tens to hundreds of seconds to optimize a CNN model using the offline process, which is acceptable for model optimization. Search time is dependent on the number of operators in a CNN model. For example, NasNet (431 operators) allocates 73.92% of the total time to search, whereas SqueezeNet (50 operators) has a lower search overhead of 9.09% but a significantly higher compilation time of 72.73%. Additionally, the offline process can be further accelerated by leveraging high-performance parallel platforms.

5.4.7 Profiling AFusion's Performance with nvprof. We used nvprof to examine the performance differences of a kernel before and after applying AFusion. We selected an SISO-pattern sub-graph from DenseNet, comprising Conv1 ($C = 64, K = 48, R \times S = 1 \times 1, Stride = 1$) and Conv2 ($C = 48, K = 12, R \times S = 3 \times 3, Stride = 1$), for an input shape ($N = 1, C = 64, H = 32, W = 32$). Memory and compute metrics were compared for both fused and non-fused versions.

Table 8 presents profiling results for key metrics, including memory reads, memory access efficiency, memory throughput, L1 cache hit ratio, and occupancy. These findings confirm that the fused kernel improves memory efficiency and throughput compared to the non-fused version by reducing DRAM read accesses and optimizing memory access patterns (such as leveraging L1 cache and shared memory). Consequently, the fused kernel achieves superior overall performance, running faster by 2.20× than the non-fused kernel.

5.4.8 Performance Improvement Derivation. We created OptiFX-Template by disabling fusion to identify the sources of OptiFX's performance improvements. Table 9 compares the end-to-end performance of OptiFX and OptiFX-Template (normalized to TensorFlow). OptiFX-Template achieved an average speedup of 1.071×, while OptiFX reached 1.288×. These results demonstrate that both templating and fusion contribute to OptiFX's enhanced performance.

6 DISCUSSION

Optimization Effectiveness. Our experimental results demonstrate the effectiveness of OptiFX in accelerating state-of-the-art convolutional neural networks through fusion exploration. AFusion broadens the optimization space by

Table 8. Profiling comparison of an SISO-pattern sub-graph before and after AFusion on Nvidia A6000 Ada.

	Metrics	AFusion	Non-Fused
Memory	DRAM Read (Bytes)	9492	16972
	Memory Access Efficiency (%)	17.38	4.81
	Memory Throughput (%)	19.36	4.60
	L1 Read Efficiency (%)	17.38	4.81
	L1 Hit Ratio (%)	97.42	43.84
	Shared Memory Throughput (%)	88.8	16.91
Compute	Compute Throughput (%)	19.36	4.85
	Achieved Occupancy (%)	24.8	4.10
Latency (ms)		0.021	0.046

Table 9. Performance speedups of OptiFX-Template and OptiFX (normalized to TensorFlow) on Nvidia RTX 4090.

OptiFX Configuration	GoogLeNet	ResNet-50	DenseNet	MobileNet	SqueezeNet	NasNet	UNet	GeoMean
OptiFX-Template	1.004×	1.022×	1.316×	1.134×	1.086×	1.023×	1.070×	1.071×
OptiFX	1.316×	1.050×	1.688×	1.260×	1.321×	1.227×	1.119×	1.288×

fusing sub-graphs with two or more ROPs. To determine if a model can benefit from AFusion on a target platform, we use a profiling-guided search approach. Although the speedups achieved by AFusion vary across different sub-graphs, the Roofline model provides a useful estimate of the potential performance gains.

Impact on Modern DNN Systems. We focus on CNN models, which are pivotal in intelligent applications, with fusion strategies specifically tailored to convolution operators. However, OptiFX provides a versatile fusion optimization framework that can be extended to other model architectures. It is expected to effectively mitigate memory wall challenges in modern DNN systems, particularly those deployed on high-performance computing platforms.

Roofline Model. The core objective of our fusion method is to minimize data movement between global and local memory (e.g., shared memory) on GPU accelerators. We employed a simplified Roofline model to effectively identify performance bottlenecks—whether compute-bound or memory-bound—by focusing solely on global memory and avoiding complex hardware intricacies. In future work, we aim to leverage an advanced Roofline model that incorporates memory hierarchy and overlapped data transfers using TMA techniques to further enhance kernel fusion.

Framework-Agnostic Approach. Our method employs an Arithmetic-Intensity-Aware Computational Graph (AGC) to represent CNN models, enhancing traditional computational graphs with operator meta-information. This AGC representation is compatible with existing deep learning frameworks, making AFusion a framework-agnostic optimization tool. Consequently, AFusion can optimize graphs generated by current frameworks without requiring model modifications. Once fused, the optimized model can be converted back to a traditional computational graph and executed using frameworks like TensorFlow.

Optimizing DNNs on Tensor Cores and other NPU Platforms. AFusion is a versatile operator fusion optimization framework applicable to various Domain-Specific Accelerators (DSAs), including Tensor Cores and other NPUs. For example, Google TPU v4 [19] offers higher compute performance and memory bandwidth ratios (275 TFLOP/s and 1200 GB/s) compared to the A6000 Ada CUDA Core (91.1 TFLOP/s and 960 GB/s). This makes memory-wall issues more pronounced on DSAs [20, 23, 24, 49], presenting opportunities for further optimization through AFusion by enhancing memory access and computational efficiency, especially for memory-bound systems. To extend AFusion for DSAs, two key optimizations may be explored:

(1) **Tiling Strategies.** Develop tiling approaches that align with hardware-accelerated tensor operations. This involves designing specific tiling patterns to leverage the specialized shapes of tensor computation instructions,

ensuring data and computation patterns are optimized for Tensor Core and NPU architectures and enhancing parallel code performance.

(2) **Data Reuse Strategies.** Improve on-chip cache utilization by optimizing the order of operations to minimize data footprints or implementing advanced caching techniques that dynamically adapt to the data access patterns of fused operators. These strategies help mitigate cache contention and efficiently manage limited on-chip memory, particularly for GEMM-based operations on Tensor Core and NPU.

Limitations. OptiFX can be enhanced in several ways. First, while CNN models are the focus, large language models (LLMs) like BERT and GPT rely heavily on matrix multiplications, which are often memory-bound on GPUs. We plan to extend our aggressive fusion approach to support additional operators, enabling broader applicability to intelligent applications. Second, OptiFX currently targets model inference acceleration through fusion optimization. Future work will explore applying fusion optimization to model training scenarios. Third, neural network quantization is a common technique for optimizing deep learning models. Therefore, supporting low-precision (such as INT8 and INT4) model fusion is another direction for future research. Finally, we plan to explore integrating our fusion method with tensor compilers like TVM [5], leveraging its auto-scheduler, Ansor [49], to achieve further performance improvements.

7 RELATED WORK

In this section, we review related work on operator fusion, deep learning compilers, and other optimization methods for accelerating CNN inference. Existing deep learning frameworks and compilers, such as TensorFlow [1] and PyTorch [2, 31], utilize kernel fusion methods on DNN graphs. These frameworks, often supported by backend libraries like cuDNN [7], fuse operators like CBR into single kernels, ensuring data dependencies are maintained.

Recent advances in industry and academia have introduced multi-level fusion methods, including loop fusion, kernel fusion, operator fusion, and sub-graph fusion. Loop fusion techniques, such as Halide's producer-consumer fusion schedule [33], generate fused parallel code based on loop structures. Overlapped tiling approaches like PolyMage [27] and Hierarchical Overlapped Tiling [54] implement fusion tiling schedules on output images, achieving effective scheduling results. These methods apply overlapped producer-consumer fusion at the loop level for individual operators.

Polyhedral compilers, including Tensor Comprehensions [38] and Tiramisu [3], enhance performance through polyhedral tiling [43, 44] and loop transformations. AKG [48] specializes in automatic polyhedral code generation for machine learning applications on NPUs.

Kernel fusion approaches [32, 39, 41, 53] reduce the overhead of multiple kernel functions in memory-bound applications. Frameworks like Occam [11] improve data reuse through kernel fusion, while others, like TVM [5, 6] and TensorFlow-XLA [1, 20, 30], use machine learning-assisted technologies for fusion.

Operator fusion optimizes computational graphs by merging multiple operators into a single entity. Frameworks such as TASO [17] and PET [40] automatically replace sub-graphs with fused operators. MIOpen [21] and Jittor [14] decompose computing steps for fusion, while Chimera [51] and Apollo [47] enhance locality and optimize kernel fusion using polyhedral techniques. The key contributions of our AFusion approach include fusing multiple convolution operators and supporting fusion for sub-graphs with branching structures, enabling graph-level fusion among operators.

Sub-graph fusion methods combine computational graph and operator dependency information for comprehensive optimization. Xenos [46] accelerates edge-based inference using data flow information, while DNNFusion [28] and Astitch [53] replace inefficient operators with efficient fusion operators. Rammer [26] and EINNET [50] focus on high-performance kernel replacement and tensor program compilation optimization, respectively. Fine-grained tuning frameworks like DyCL [4], Welder [35], FlexTensor [52], and Ansor [49] further refine DNN optimization.

These frameworks perform fusion based on graph and operator dependencies, either vertically for ElemWiseOps [5, 20] or horizontally for RegionOps and ElemWiseOps [17, 18]. While existing frameworks have made significant progress in operator fusion, OptiFX introduces a more general fusion methodology, AFusion, which extends fusion capabilities to handle multiple dependency relationships, achieving higher performance.

8 CONCLUSION

In this paper, we investigated memory wall effects on the performance of modern CNNs on GPU platforms and introduced AFusion, an aggressive fusion approach for optimizing inference. We developed OptiFX, an automatic optimization framework that uses operator fusion strategies from AFusion to accelerate CNN model inference. OptiFX identifies and optimizes fusion opportunities within CNN models, utilizing the Roofline model to generate and evaluate candidate sub-graphs. Our evaluations across state-of-the-art CNN models demonstrate the effectiveness of OptiFX, achieving substantial speedups of 2.91×, 3.30×, and 2.09× on the Nvidia A6000 Ada, RTX 4090, and Jetson AGX Orin platforms, respectively, outperforming existing methods.

ACKNOWLEDGMENTS

We thank all reviewers for their insights, which have improved the quality of this work. We thank Yanfan Chen for his contributions to the early implementation of this work. This research is supported by the National Science and Technology Major Project (2023ZD0120502), the National Natural Science Foundation of China (62302479, 62232015, 62090024, 62372055), the fund of Laboratory for Advanced Computing and Intelligence Engineering, the China Post-doctoral Science Foundation (2024M750258, 2023M733566), the CCF-Tencent Rhino-Bird Open Research Fund, State Key Lab of Processors, Institute of Computing Technology, CAS (CLQ202411), the Innovation Funding of ICT, CAS (E361010, E261110), and the Australian Research Council (ARC) Grant (DP250104934).

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: a system for Large-Scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation*. 265–283.

[2] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. 2024. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 929–947.

[3] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE, 193–205.

[4] Simin Chen, Shiyi Wei, Cong Liu, and Wei Yang. 2023. Dycl: Dynamic neural network compilation via program rewriting and graph optimization. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 614–626.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation*. 578–594.

[6] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to optimize tensor programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. 3393–3404.

[7] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[8] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems* 35 (2022), 16344–16359.

[9] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[10] Roy Frostig, Matthew James Johnson, and Chris Leary. 2018. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning* 4, 9 (2018).

[11] Ashish Gondimalla, Jianqiao Liu, Mithuna Thottethodi, and TN Vijaykumar. 2022. Occam: Optimal data reuse for convolutional neural networks. *ACM Transactions on Architecture and Code Optimization* 20, 1 (2022), 1–25.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[13] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[14] Shimin Hu, Dun Liang, Guoye Yang, Guowei Yang, and Wenyang Zhou. 2020. Jittor: A novel deep learning framework with meta-operators and unified graph execution. *Science China Information Sciences* 63, 12 (2020), 1–21.

[15] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4700–4708.

[16] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. arXiv:1602.07360 [cs.CV]

[17] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 47–62. https://doi.org/10.1145/3341301.3359630

[18] Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2019. Optimizing DNN Computation with Relaxed Graph Substitutions. In *Proceedings of Machine Learning and Systems*, A. Talwalkar, V. Smith, and M. Zaharia (Eds.), Vol. 1. 27–39. https://proceedings.mlsys.org/paper_files/paper/2019/file/4dd1a7279a8cfeea2660fbc34f02a2bc-Paper.pdf

[19] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, et al. 2023. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–14.

[20] Sam Kaufman, Phitchaya Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. 2021. A Learned Performance Model for Tensor Processing Units. *Proceedings of Machine Learning and Systems* 3 (2021).

[21] Jehandad Khan, Paul Fultz, Artem Tamazov, Daniel Lowell, Chao Liu, Michael Melesse, Murali Nandhimandalam, Kamil Nasyrov, Ilya Perminov, Tejash Shah, Vasilii Filippov, Jing Zhang, Jing Zhou, Bragadeesh Natarajan, and Mayank Daga. 2019. MIOpen: An Open Source Library For Deep Learning Primitives. arXiv:1910.00078 [cs.LG]

[22] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14.

[23] Guangli Li, Xiu Ma, Xueying Wang, Hengshan Yue, Jiansong Li, Lei Liu, Xiaobing Feng, and Jingling Xue. 2022. Optimizing deep neural networks on intelligent edge accelerators via flexible-rate filter pruning. *J. Syst. Archit.* 124 (2022), 102431. https://doi.org/10.1016/J.SYSARC.2022.102431

[24] Guangli Li, Jingling Xue, Lei Liu, Xueying Wang, Xiu Ma, Xiao Dong, Jiansong Li, and Xiaobing Feng. 2021. Unleashing the Low-Precision Computation Potential of Tensor Cores on GPUs. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, Jae W. Lee, Mary Lou Soffa, and Ayal Zaks (Eds.). IEEE, 90–102. https://doi.org/10.1109/CGO51591.2021.9370335

[25] Jianhui Li, Zhennan Qin, Yijie Mei, Jingze Cui, Yunfei Song, Ciyong Chen, Yifei Zhang, Longsheng Du, Xianhang Cheng, Baihui Jin, et al. 2024. oneDNN Graph Compiler: A Hybrid Approach for High-Performance Deep Learning Compilation. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 460–470.

[26] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In *14th USENIX Symposium on Operating Systems Design and Implementation*. 881–897.

[27] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. Polymage: Automatic optimization for image processing pipelines. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 429–443.

[28] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: Accelerating Deep Neural Networks Execution with Advanced Operator Fusion *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 883–898. https://doi.org/10.1145/3453483.3454083

[29] Inc Nvidia. 2024. NVIDIA TensorRT. https://developer.nvidia.com/tensorrt Accessed: Dec 2024.

[30] OpenXLA. 2024. XLA architecture. https://openxla.org/xla/architecture Accessed: Dec 2024.

[31] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems* 32 (2019).

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

[33] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

[34] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-net: Convolutional networks for biomedical image segmentation. In *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*. Springer, 234–241.

[35] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. 2023. Welder: Scheduling Deep Learning Memory Access via Tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation*. 701–718.

[36] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1–9.

[37] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. 2019. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2820–2828.

[38] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. arXiv:1802.04730 [cs.PL]

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

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

[41] Xueying Wang, Guangli Li, Xiao Dong, Jiansong Li, Lei Liu, and Xiaobing Feng. 2020. Accelerating deep learning inference with cross-layer data reuse on GPUs. In *European Conference on Parallel Processing*. Springer, 219–233.

[42] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[43] Jingling Xue. 1997. On Tiling as a Loop Transformation. *Parallel Process. Lett.* 7, 4 (1997), 409–424. https://doi.org/10.1142/S0129626497000401

[44] Jingling Xue. 2000. *Loop Tiling for Parallelism*. The Kluwer International Series in Engineering and Computer Science, Vol. 575. Kluwer. http://www.springer.com/computer/communication+networks/book/978-0-7923-7933-1

[45] Feng Yu, Guangli Li, Jiacheng Zhao, Huimin Cui, Xiaobing Feng, and Jingling Xue. 2024. Optimizing Dynamic-Shape Neural Networks on Accelerators via On-the-Fly Micro-Kernel Polymerization. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. Association for Computing Machinery, New York, NY, USA, 797–812.

[46] Runhua Zhang, Hongxu Jiang, Jinkun Geng, Fangzheng Tian, Yuhang Ma, and Haojie Wang. 2024. A high-performance dataflow-centric optimization framework for deep learning inference on the edge. *Journal of Systems Architecture* 152 (2024), 103180.

[47] Jie Zhao, Xiong Gao, Ruijie Xia, Zhaochuang Zhang, Deshi Chen, Lei Chen, Renwei Zhang, Zhen Geng, Bin Cheng, and Xuefeng Jin. 2022. Apollo: Automatic Partition-based Operator Fusion through Layer by Layer Optimization. *Proceedings of Machine Learning and Systems* 4 (2022), 1–19.

[48] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, et al. 2021. AKG: automatic kernel generation for neural processing units using polyhedral transformations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1233–1248.

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

[50] Liyan Zheng, Haojie Wang, Jidong Zhai, Muyan Hu, Zixuan Ma, Tuowei Wang, Shuhong Huang, Xupeng Miao, Shizhi Tang, Kezhao Huang, and Zhihao Jia. 2023. EINNET: Optimizing Tensor Programs with Derivation-Based Transformations. In *17th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, Boston, MA, 739–755. https://www.usenix.org/conference/osdi23/presentation/zheng

[51] Size Zheng, Siyuan Chen, Peidi Song, Renze Chen, Xiuhong Li, Shengen Yan, Dahua Lin, Jingwen Leng, and Yun Liang. 2023. Chimera: An Analytical Optimizing Framework for Effective Compute-intensive Operators Fusion. In *2023 IEEE International Symposium on High-Performance Computer Architecture*. IEEE, 1113–1126.

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

[53] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, et al. 2022. AStitch: Enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 359–373.

[54] Xing Zhou, Jean-Pierre Giacalone, María Jesús Garzarán, Robert H Kuhn, Yang Ni, and David Padua. 2012. Hierarchical overlapped tiling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. 207–218.