(article starts on next page)

Fusing Depthwise and Pointwise Convolutions for Efficient Inference on GPUs

Fareed Qararyah
qararyah@chalmers.se
Chalmers University of Technology
Göteborg, Sweden

Muhammad Waqar Azhar
waqar.azhar@zptcorp.com
ZeroPoint Technologies AB
Gothenburg, Sweden

Mohammad Ali Maleki
mohammad.ali.maleki@chalmers.se
Chalmers University of Technology
Göteborg, Sweden

Pedro Trancoso
ppedro@chalmers.se
Chalmers University of Technology
Göteborg, Sweden

ABSTRACT

Depthwise and pointwise convolutions have fewer parameters and perform fewer operations than standard convolutions. As a result, they have become increasingly used in various compact DNNs, including convolutional neural networks (CNNs) and vision transformers (ViTs). However, they have a lower compute-to-memory-access ratio than standard convolutions, making their memory accesses often the performance bottleneck.

This paper explores fusing depthwise and pointwise convolutions to overcome the memory access bottleneck. The focus is on fusing these operators on GPUs. The prior art on GPU-based fusion suffers from one or more of the following: (1) fusing either a convolution with an element-wise or multiple non-convolutional operators, (2) not explicitly optimizing for memory accesses, (3) not supporting depthwise convolutions. This paper proposes Fused Convolutional Modules (FCMs), a set of novel fused depthwise and pointwise GPU kernels. FCMs significantly reduce pointwise and depthwise convolutions memory accesses, improving execution time and energy efficiency. To evaluate the trade-offs associated with fusion and determine which convolutions are beneficial to fuse and the optimal FCM parameters, we propose FusePlanner. FusePlanner consists of cost models to estimate the memory accesses of depthwise, pointwise, and FCM kernels given GPU characteristics. Our experiments on three GPUs using representative CNNs and ViTs demonstrate that FCMs save up to 83% of the memory accesses and achieve speedups of up to 3.7x compared to cuDNN. Complete model implementations of various CNNs using our modules outperform TVMs' achieving speedups of up to 1.8x and saving up to two-thirds of the energy. FCM and FusePlanner implementations are open source: https://github.com/fqararyah/Fusing_DW_and_PW_on_GPUs

CCS CONCEPTS

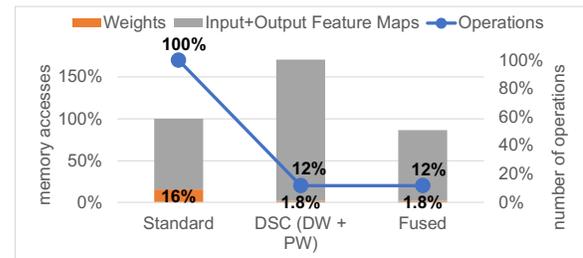• **Computing methodologies → Parallel computing methodologies**; *Machine learning*.

Figure 1: Operation count and memory accesses of a standard, DSC (DW+PW), and fused convolutions. The example is taken from MobileNet. All values are normalized to the standard convolution values

KEYWORDS

depthwise convolution, pointwise convolution, CNN, vision transformer, layer fusion, GPU

1 INTRODUCTION

Convolutions are core operators in many Deep Learning (DL) models, including Convolutional Neural Networks (CNNs) [14, 15, 20], Vision Transformers (ViTs) [10, 13, 43, 45, 48, 49], and Graph Convolutional Networks (GCNs) [33, 51]. Splitting a standard convolution into depthwise (DW) and pointwise (PW) convolutions reduces the model size and operation count [9, 15]. To give an example, XCeption CNN, which uses DW and PW convolutions, has an accuracy that surpasses ResNet-152's [14] despite being roughly three times smaller [9]. Hence, DW and PW convolutions are increasingly replacing standard convolutions in designing compact models that achieve state-of-the-art accuracy [9, 15, 36, 37, 43, 49].

Figure 1 demonstrates the effect of splitting a standard convolution into depthwise separable convolution (DSC) [15], *i.e.* DW plus PW, on the operation count, weights, and input and output sizes. The DW and PW convolutions require fewer weights and perform fewer operations than standard convolutions. In this example, the DSC operations are 12% of the standard convolution operations.

Fareed Qararyah, Muhammad Waqar Azhar, Mohammad Ali Maleki, and Pedro Trancoso

Similarly, the weights in DSC are reduced to 1.8% compared to 16% in standard convolution. However, their combined inputs and outputs are larger. The net result is having fewer operations but more memory access. In other words, DW and PW are more often memory-bound compared to standard convolutions [26]. As a result, their memory accesses form a performance bottleneck on most of the commonly used accelerators.

Operator fusion, or layer fusion, reduces off-chip memory accesses considerably compared to traditional layer-by-layer execution. In layer-by-layer execution, the convolution processes its inputs completely and writes the results to the main memory. However, fusing layers allows the intermediate results to be processed at finer granularity while on-chip [2, 6, 16, 50]. In the example in Figure 1, fusion saves 50% of the off-chip memory accesses of the DW and PW convolutions. Fusing a convolution with an element-wise operation like normalization and non-linearity is a common optimization applied by DL compilers like TVM [7] and DNNVM [46]. However, due to complex input-output dependencies among convolutions, fusing multiple convolutions could incur numerous redundant computations or memory accesses [2]. Nonetheless, the prior art has demonstrated that when handling the trade-offs properly, fusing convolutions is beneficial on custom accelerators [2, 4, 6, 12, 16, 25, 28, 31, 34, 38, 42, 44, 46, 47, 50, 54].

As GPUs have been key accelerators in the resurgence of DL [20] and are the most widely-supported accelerators by various DL frameworks [1, 3, 18, 29], they are an ideal target of various optimizations including layer fusion. However, when it comes to fusing multiple convolutions, GPU programming abstractions and memory level access constraints make managing cross-convolution dependencies challenging [2].

The prior art on layer fusion suffers from one or more of the following limitations. First, on GPUs, they either consider fusing a convolution and an element-wise or multiple non-convolutional operators [7, 11, 17, 24], do not explicitly model and optimize memory accesses, or do not support depthwise convolutions [41, 53]. Second, the work targeting other accelerators is either tightly coupled to a specific architecture or assumes complete hardware flexibility like that offered by FPGAs [2, 25, 38, 42, 44].

In this paper, we propose *Fused Convolutional Modules (FCMs)*, a set of novel fused GPU kernels of DW and PW convolutions. FCMs reduce global memory access considerably leading to improved latency and energy efficiency. To evaluate fusion trade-offs and decide when fusion gains outweigh its overheads, we propose *FusePlanner*. Given a set of DW and PW convolutions and GPU characteristics, *FusePlanner's* cost models estimate the memory accesses of depthwise, pointwise, and FCM kernels. FusePlanner explores the feasible FCMs and layer-by-layer implementations and suggests one that minimizes memory access. FCMs can be used as library routines, and with FusePlanner they can be used to derive complete CNN execution plans. Using FusePlanner-suggested FCM and layer-by-layer implementations, we implement and evaluate convolutional layers of state-of-the-art CNNs and ViTs on three GPUs. We compare our implementation with CuDNN [8] based implementations. Moreover, we compare the performance of full implementations of the CNNs, based on FusePlanner-suggested FCM and layer-by-layer kernels, to a DL compiler (TVM) [7] optimized implementations. Our contributions are as follows:

- We propose Fused Convolutional Modules (FCMs), a set of novel GPU kernels comprising DW and PW convolutions. FCMs mitigate these convolutions' memory access bottleneck leading to low-latency and energy-efficient execution.
- We propose *FusePlanner*, FusePlanner consists of cost models that estimate global memory accesses of DW, PW, and FCM kernels given a GPU architecture. *FusePlanner* decides which layers benefit from fusion and the implementation parameters that minimize global memory accesses.
- We evaluate FCMs by comparing their performance to custom, and standard DL library-based (cuDNN) convolution kernels from representative CNNs and ViTs on multiple GPUs. We also compare end-to-end implementations of the CNNs utilizing FCMs and FusePlanner-suggested layer-by-layer implementations to TVM-optimized models.

The proposed FCMs achieve up to $1.8x$ speedup over custom layer-by-layer implementations and up to $3.7x$ over the best cuDNN implementations using representative CNNs and ViTs. FCMs save up to 83% of the global memory accesses compared to CuDNN. End-to-end implementations of four CNNs using the proposed kernels achieve up to $1.8x$ speedup compared to TVM implementations and save up to two-thirds of the energy per inference.

2 BACKGROUND AND MOTIVATION

2.1 CNNs and ViTs

Convolutional neural networks (CNNs) are feed-forward DNNs [22]. As the name suggests, the main layers in a CNN are the convolutional layers. A convolutional layer has a set of trainable parameters or *weights* grouped into *filters*. The filters are applied to multi-dimensional arrays of input or intermediate results, extracting their embedded features [23]. The inputs of a layer are known as input feature maps (*IFMs*) and the outputs as output feature maps (*OFMs*). Feature Maps (*FMs*), or activations, refer to both IFMs and OFMs. FMs comprise a set of 2D slices known as *channels*.

Transformer models are based on a self-attention mechanism that learns the relationships between elements of a sequence [39]. In vision transformers (ViTs), *self-attention* allows modeling contextual information of the full image and long-range dependencies both in space and time [19]. This paper focuses on convolutional ViTs that combine self-attention with convolutions [36, 43, 49].

2.2 Depthwise and pointwise convolutions
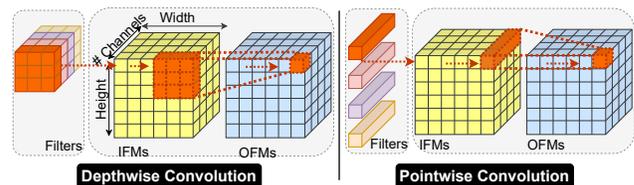

Figure 2: Depthwise and pointwise convolutions

Depthwise (DW) and Pointwise (PW) convolutions optimize DNNs' size-accuracy trade-off [5, 9, 15, 32, 36, 37, 43, 52]. They decouple the spatial and cross-channel correlations [9]. As Figure 2 shows, DW convolution is applied to the spatial dimensions, *i.e.* width, and height, where one filter is applied to a single channel.
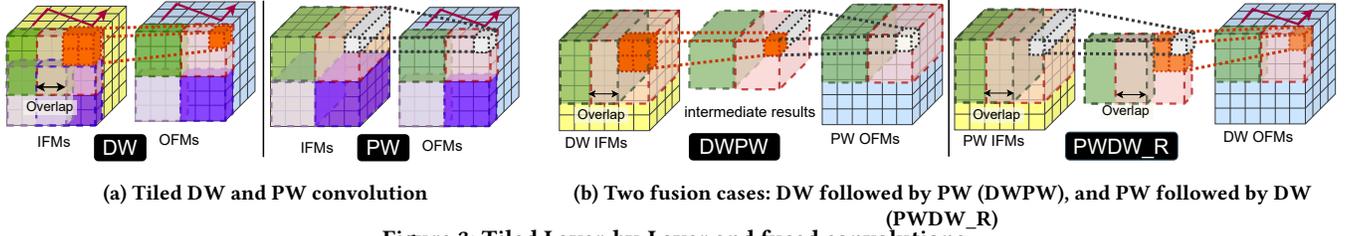
(a) Tiled DW and PW convolution

(b) Two fusion cases: DW followed by PW (DWPW), and PW followed by DW (PWDW_R)

Figure 3: Tiled Layer-by-Layer and fused convolutions

PW convolution is applied to the cross-channel dimension, where its 1×1 filters span over all channels. DW and PW convolutions are combined in various ways to build efficient modules or blocks, including *Depthwise Separable Convolutions (DSC)* and inverted residual with linear bottlenecks, or *inverted residuals* for short, [9, 15, 32, 37]. The DSC is composed of a DW followed by a PW layer. The inverted residuals comprise three convolutional layers: PW-DW-PW.

2.3 CUDA-Capable GPU architecture and programming model overview

A GPU architecture consists of a scalable array of streaming multiprocessors (SMs) [27]. An SM is a Single-Instruction-Multiple-Thread (SIMT) architecture that runs groups of parallel threads called *warps* in a lockstep fashion. A CUDA kernel is processed by a *grid* of threads. The grid consists of a set of *thread blocks*, threads in a block run on the same SM. GPU has a memory hierarchy of multiple levels with different access constraints. Each thread has private local registers. Each SM has a low-latency L1 cache, and a variable-sized portion of that cache is configurable to serve as programmer-managed *shared memory*. The shared memory is visible to all threads in a block and has the same lifetime of the block. The rest of the memory levels are globally accessible by threads of the entire CUDA kernel.

2.4 Fusing Convolutions on GPUs

There are multiple algorithms to implement convolution on GPU. We focus on the direct convolution implementation and use it as the basis for the layer-by-layer and the fused kernels. This is because other algorithms, including Winograde and FFT, require filters of greater than 1×1 width and height, so they are not applicable for PW convolution [21]. Moreover, Winograde, FFT, and GEMM optimize the computation at the cost of more memory bandwidth requirements, which does not suit PW and DW convolutions.

Figure 3a shows a simplified example of tiled DW and PW convolutions. For simplicity, each weight tile has only one filter and computes a 2d tile of 3×3 of the OFMs. The DW convolution has 2×2 filters and 4×4 tile of the IFMs. On a GPU, assuming an Output Stationary (OS) dataflow, each OFM tile gets assigned to a thread block that runs on one of the GPU's Streaming Multiprocessors (SMs). In the layer-by-layer execution, each layer is implemented as one or more CUDA kernels that process the IFMs and produce the complete OFMs. Because the SMs' L1/shared memory contents do not outlive a single kernel, all the OFMs must be written back and cannot be reused by the next layer. Note that because L1/shared memory is private to an SM, the overlap regions among the IFM tiles, in the case of DW, must be loaded multiple times depending on the number of tiles sharing them.

Figure 3b shows two fusion examples. The first example (DWPW) depicts a DW fused with its following PW, and the second (PWDW_R) shows a PW fused with a following DW. The _R indicates that this fusion entails redundant computations, as explained at the end of the section. The fused layers are implemented as a single kernel. On the one hand, unlike the layer-by-layer, the OFMs of the first layer, which are intermediate results when fusing, can be directly reused while in the L1/shared memory. This reduces the global memory access. On the other hand, the fused implementation has its own constraints and overheads. First, fusing convolutions enlarges the working set compared to the layer-by-layer implementation. In the layer-by-layer case, the working set consists of three tiles: OFMs, IFMs, and filter tiles. In the fused case, there are five tiles: IFMs of the first layer tile, OFMs of the second layer tile, two tiles of both layers' filters, and a tile of the intermediate results exchanged between the two layers. Note that we show one filter in the figures for simplicity, in practice a filter tile may contain hundreds or thousands of filters. As more tiles compete for the L1/shared memory, each has a smaller share. Smaller tiles lead to more overlapping, less reuse, and more frequent access to the global memory. Second, certain fusion cases restrict the viable tile sizes. For example, the PW layer in the DWPW fusion case in Figure 3b requires at least one element of each channel of the intermediate results to produce one valid output. Consequently, the intermediate results and the input tiles must contain all the channels. In other words, a DW tiling similar to the one shown in Figure 3a is not feasible. Third, the values located at the overlap regions of intermediate results tiles must be redundantly computed. PWDW_R in Figure 3b shows an example of this. Unlike the overlaps in the IFMs, the values at the overlap in the intermediate results do not exist before the fused kernel starts. They must be computed independently by the SMs computing the overlapping tiles.

Searching for fused implementations of PW and DW convolutions that minimize memory accesses, and consequently mitigate or overcome their bottlenecks requires evaluating the gains and overheads of the feasible fusions compared to layer-by-layer execution. We propose Fused implementations of PW and DW convolutions and cost models that evaluate the discussed overheads and suggest implementations that minimize the global memory accesses.

3 FUSED CONVOLUTIONAL MODULES (FCMS)

3.1 FCMs overview

DW and PW convolutions are commonly found in DNNs, e.g. CNNs and ViTs, in the form of *depthwise separable convolutions (DSC)*, or *inverted residuals* (Section 2). Figure 4 shows a sequence of two DSC blocks and a sequence of two inverted residuals. It depicts the three possible PW and DW combinations. Fused Convolutional Modules (FCMs) target such combinations, which are **DWPW**, **PWDW**,
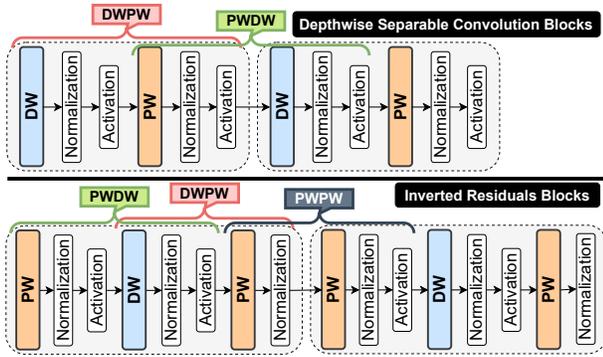
Figure 4: Possible FCMs in DNNs composed of Depthwise Separable Convolutions and Inverted Residuals blocks

and **PWPW**. The PWDW FCM has two variants, one that requires redundant computations (**PWDW_R** shown in Figure 3b), and one that does not (**PWDW**). The PWDW does not require redundant computations if there is no tiling across the width and height of an IFM. An FCM combines up to 6 layers, two convolutional layers, and the normalization and activation layers following each. As Figure 4 shows, FCMs fuse layers of a single separable convolution or inverted residual blocks, or layers belonging to two consecutive blocks. All the FCM kernels adopt the efficient *Output Stationary - Local Weight Stationary (OS-LWS)* dataflow [40].

3.2 FCM kernel structure

Listing 1: FCM kernels skeleton

```
1  __global__ void FCM_Skeleton(/*Parameters*/) {
2      /*****************part 1*****************/
3      __shared__ fms_dt commBuffer[BUFFER_SIZE];
4      //Other declarations
5      /*****************part 2*****************/
6      // Prefetch fused layers weight tiles
7      if (/*Thread ID in loader thread IDs*/) {
8          //Prefetch weights to shared memory or
9          //registers
10     }
11     // Synchronize
12     /*****************part 3*****************/
13     // First layer core
14     if (/*Thread ID in Conv1 thread IDs*/) {
15         //Compute Conv-Norm-Activation
16         //Pack and write to commBuffer
17     }
18     // Synchronize
19     /*****************part 4*****************/
20     // Second layer core
21     if (/*Thread ID in Conv2 thread IDs*/) {
22         //Load second layer IFM tile from commBuffer
23         //Compute Conv-Norm-Activation
24         //Pack and write back to the OFMs
25     }
26 }
```

Listing 1 highlights the main parts of the skeleton of an FCM kernel. The skeleton is divided into four main parts. *Part1* (lines 2-4) contains the declaration of the buffer used to communicate between the first set of layers (convolution-normalization-activation) and the second (*commBuffer*). The buffer is stored in an SM's shared memory. Shared memory banks are organized such that consecutive words map to consecutive banks. The access patterns to these banks

are crucial to the kernel performance. To fully utilize these banks' throughput, the data layout is selected based on the FCM layers implementation to always have a linear addressing with a stride of one, a conflict-free access pattern.

Part2 (lines 6-11) contains the prefetching of layer weights. The weights are fetched ahead of computation in two scenarios. The first scenario is when the implementation of either FCM's two convolutions does not access the weights contiguously by default due to a mismatch between the convolution dataflow and loop ordering, and the data layout of the weights buffer [24]. In such cases, separating the weights load from the computation allows to load weights contiguously. The second scenario is when warp-level primitives are used. We use a warp-level primitive (*__shfl_sync*) to exchange the weights between threads through registers rather than shared memory.

In *parts 3 and 4* (lines 13-24), if the weights have been fetched in part 2, they are now loaded from the shared memory or shuffled from other threads registers; otherwise, they are loaded from global memory. Then, a fused convolution-normalization-activation operation is applied. The implementations currently support both INT8 and FP32 data types. In the case of INT8, *__dp4a CUDA intrinsic* four-way dot product with 32-bit accumulate is used. The first convolution-normalization- activation of the FCM computes a tile of the intermediate results and writes it to the shared *commBuffer*. Then, the fused convolution-normalization-activation part reads the intermediate results from the *commBuffer*, computes, and writes back the FCM output to the OFMs buffer. Synchronization is necessary between these two parts as different threads may participate in each part. When using INT8, every four results are grouped, or packed, into one 32-bit integer before writing to any buffer. The weights are also packed, weight packing is done offline since the weights do not change in inference.

4 FUSEPLANNER

FusePlanner aims to identify the FCMs and layer-by-layer implementations that minimize global memory access, given a set of DW and PW layers and GPU specifications. Figure 5 shows an overview of FusePlanner. It takes as inputs: (1) GPU number of SMs, L1 size, and the portion configurable as shared memory; and (2) a DAG representing a model or set of layers, their weight and FM specifications, and the layers connectivity. We currently support generating model DAGs from Tensorflow. FusePlanner has two main components, *layer-by-layer global memory access estimator* and *FCMs global memory access estimator*. FusePlanner does a first pass over the layers and estimates their minimum global memory access using the layer-by-layer global memory access estimator. After that, it examines all the possible fusions and evaluates their global memory access using the FCMs global memory access estimator. Based on the layer-by-layer and FCM estimates, FusePlanner outputs: (1) which layers are to be fused and which are not, (2) which FCMs to use, and (3) the tiling that minimizes the global memory access in each case. FusePlanner is used offline (before inference) and is applied only once per model-GPU pair. In the model implementation, the layers that the FusePlanner suggests to fuse are replaced by calls to the corresponding FCM kernels.
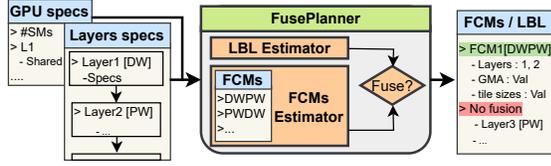
Figure 5: FusePlanner overview

4.1 Layer-by-layer global memory access estimator

We propose fast and simple cost models to explore the search space for implementation parameters that minimize global memory access efficiently. To construct a simple cost model, we make two assumptions that prune the search space by excluding implementations that do not perform well on GPUs. The first assumption is that the data layout guarantees that threads in a warp access consecutive memory locations and that the memory transactions are naturally aligned [27]. The second assumption is that the implementation follows an Output Stationary-Local Weight Stationary (OS-LWS) dataflow [40] and guarantees that the partial sums stay in registers and that only the final results are written to the memory. To ensure that, all the weights and IFM elements needed to produce one OFM element must be in the same tile (Section 2.4). To prove the effectiveness of this approach experimentally, we show that our layer-by-layer implementations outperform CuDNN (Section 6.2).

The global memory accesses of kernels that meet the two discussed assumptions are estimated using Equations 2 and 3. Where the *overlap* (described in Section 2.4) is obtained using Equation 1, the postfix *GMA* stands for global memory access, *Sz* stands for size, *W* for width, *H* for height, *D* for depth, and *HW* for *height* × *width*. As the equations show, the OFMs are written once, because the dataflow (OS-LWS) is a variant of OS. In the PW case, each weight tile is convolved with all IFM tiles, and each IFM tile is convolved with all filters. Hence, weights and IFMs memory accesses depend on each other's tiling. In the DW case, as at least one filter slice must be assigned to each SM (to guarantee assumption 2), there are no weight tiles splitting filters' height and width. As a result, the IFM elements, except the overlapping, are read only once. FusePlanner explores the tile sizes that meet two constraints. The first constraint is that the tiles fit into the L1 cache to avoid misses and redundant loading. Note that the subset of these tiles stored on the shared memory portion of the cache must also fit within that portion. However, the implementation must guarantee that; this is why it is not expressed in the equations. The second constraint is that the number of OFM tiles is greater than or equal to the number of GPU SMs. Having more OFM tiles than the number of GPU SMs ensures that the GPU resources are not underutilized.

$$Overlap = (\lceil \frac{ChannelW}{TileW} \rceil - 1) \times (FilterW - Strides) \times ChannelH + (\lceil \frac{ChannelH}{TileH} \rceil - 1) \times (FilterH - Strides) \times ChannelW \tag{1}$$

$$PwGMA = \lceil \frac{WeightsSz}{WeightsTileSz} \rceil \times IFMsSz + OFMsSz + \lceil \frac{OFMsSz}{OFMsTileSz} \rceil \times WeightsSz$$
$$where\ L1Sz \geq IFMsTileSz + OFMsTileSz + WeightsTileSz$$
$$and\ \#OFMsTiles \geq \#SMs \tag{2}$$

$$DwGMA = 2 \times IFMsD \times Overlap + IFMsSz + OFMsSz + \lceil \frac{OFMsHW}{OFMsTileHW} \rceil \times WeightsSz$$
$$where\ L1Sz \geq IFMsTileSz + OFMsTileSz + WeightsTileSz$$
$$and\ \#OFMsTiles \geq \#SMs \tag{3}$$

4.2 FCMs global memory access estimator

Estimating an FCM global memory access is based on Equations 2 and 3, with two key differences. First, neither the OFMs of the first convolutional layer of an FCM nor the IFMs of its second contribute to the global memory accesses. This is because they are now intermediate results communicated through the communication buffer. Secondly, the accesses of each of the two convolutional layers are affected by the other. Equation 4 shows an example of FCM's global memory access estimation, a PWDW FCM in this case. The equation again assumes that the fused kernel meets the two assumptions described in the previous section. The equation shows the mentioned two key differences. First, neither the OFMs of the PW nor the IFMs of the DW layer contribute to the global memory accesses. Secondly, the accesses to the first layer's IFMs depend on both layers' weights tiles because the OFMs of the first layer are not written to the global memory. Hence, when the second layer needs them, they must be recomputed, which requires redundant loading of the corresponding IFM elements. Finally, the overlap accesses depend on both layers' IFMs. As the equation shows, the overall overlap is obtained by multiplying the PW IFMs depth, rather than the DW IFMs depth, by DW IFM overlap. This is due to the same reason, *i.e.* OFMs of the FCM's first layer are not written to the global memory, and the overlap in the second layer's IFM elements are obtained by loading the first layer's IFMs and recomputing. The equations of the other FCMs are constructed from the PW and DW Equations 2, and 3 similarly.

$$PwDwGMA = (2 \times PwIFMsD \times DwOverlap + PwIFMsSz) \times max(\lceil \frac{PwWeightsSz}{PwWeightsTileSz} \rceil, \lceil \frac{DwWeightsSz}{DwWeightsTileSz} \rceil) + \lceil \frac{DwOFMsSz}{DwOFMsTileSz} \rceil \times PwWeightsSz + \lceil \frac{DwOFMsHW}{DwOFMsTileHW} \rceil \times DwWeightsSz$$
$$where\ L1Sz \geq PwIFMsTileSz + DwOFMsTileSz + PwWeightsTileSz + DwWeightsTileSz + CommBufferSz$$
$$and\ \#FCM\_OFMsTiles \geq \#SMs \tag{4}$$

The first constraint in Equation 4 is more restrictive in FCMs than layer-by-layer, as five tiles rather than three compete for the L1. And as the equation shows, simply having smaller tiles is not always a solution as it may increase the overall memory accesses. The effect of having two weight tiles per SM, compared to one tile in the layer-by-layer case, is not crucial when fusing DW and

Table 1: Used GPUs specifications

GPU	Compute Capability	#SM	CUDA cores	L1/shared (KB)	L2 (MB)	Off-chip Memory
GTX-1660	7.5	22	1408	96	1.5	GDDR5
RTX-A4000	8.6	128	6144	128	4	GDDR6
Jetson AGX **Orin**	8.7	16	2048	192	4	LPDDR5

PW since DW weights are much smaller than PWs' in most cases. However, the effect becomes considerable when two PW layers are fused. That is why PWPW fusion is less likely when the weights use FP32 compared to INT8 (Table 2).

FusePlanner explores all tile sizes that meet the constraints in Equations 2, 3, and 4 and identifies the ones that minimize the global memory accesses for the layer-by-layer and all the possible FCM cases. The explored tile sizes are restricted to multiples of the warp size to avoid resource underutilization. FusePlanner suggests fusing, when there is an FCM for which the minimum estimated global memory accesses are less than those of its constituting layers. Otherwise, a layer-by-layer implementation is suggested.

5 EXPERIMENTAL SETUP

5.1 Evaluation system

We use three GPUs, listed in Table 1, with different resources representing different points in the compute continuum. We refer to them as *GTX*, *RTX*, and *Orin* in the rest of the paper. Although the evaluated GPUs belong to Ampere and Turing architectures, the proposed modeling applies to other GPU architectures. CUDA-11.6 is used. CUDA events API is used to measure the execution time, and *nvidia-smi* utility to measure the power consumption on GTX and RTX and *tegrastats* on Orin. *NVIDIA Nsight Compute* is used to quantify accesses to all memory levels and their throughput and to categorize kernels into compute and memory-bound.

5.2 Workloads

We evaluate the proposed modules using PW and DW convolutions from four representative CNN models and two ViTs. These are MobileNet (**Mob_v1**) [15], MobileNetV2 (**Mob_v2**) [32], XCeption (**XCe**) [9], ProxylessNAS (**Prox**) [5], **CeiT** [49], and **CMT** [13]. The evaluation is done with FP32 and INT8, the original and the commonly used precision in inference, respectively. We do a fine-grained evaluation using pairs of layers, or fusion cases, from these DNNs that FusePlanner suggested. Table 2 lists these fusion cases that we use in our experiments, from which DNNs they are selected, and the ratios of redundant computations if there are any. These cases represent the scenarios where FusePlanner suggests the same fusion type across the three GPUs. A fusion case may occur in a DNN once or multiple times. This is because DNNs usually contain replicated blocks composed of layers of the same hyper-parameters. For example, F1_8 in the INT8 case represents the second and third layers of Mob_v1, but F2_8 represents five pairs of layers (pairs located between layers 14-24). The fused layers, consequently the FCMs, in the case of INT8 are not necessarily the same in the case of FP32. For example, F1_8 is different from F1 in FP32. Changing the bit-width changes the tile sizes causing FusePlanner to make different choices.

5.3 Baselines

To demonstrate the effect of fusion on global memory access reduction and performance improvement of PW and DW convolutions, we implement and compare the FCMs and layer-by-layer kernels (*LBL*). To isolate the fusion effect, the LBL kernels have similar dataflow and access patterns to their fused counterpart. We compare the FCMs and LBL kernels to *cuDNN* [8]. cuDNN gives the flexibility of choosing the convolution algorithms, we compare against three cuDNN algorithms that yielded the best performance on our workloads, namely GEMM, IMPLICIT_GEMM, and IMPLICIT_PRECOMP_GEMM. We also do end-to-end evaluations where the four CNNs, Mob_v1, Mob_v2, XCe, and Prox, are fully implemented using our FCMs and LBL kernels and compared with *TVM* [7]. TVM is an open-source and widely used end-to-end deep learning compiler. We use cuDNN as the backend of our TVM implementation to maintain consistency. TVM is configured with the first generation of NNVM. The models are converted into relay IR. TVM applies layer fusion between convolution and non-convolution layers as a core optimization. Not all TVM models ran successfully on all the GPUs after applying TVM's offline graph optimizations. Hence we ran auto-tuning for 20 iterations with the hardware in the loop which was sufficient for all models. TVM offers several tuning heuristic options, in our experiments, we compare our results with the best TVM heuristic in each experiment. Note that cuDNN and TVM implementations that we compare against fuse a single convolutional layer with the normalization and activation layers following it. However, we refer to their execution as layer-by-layer (LBL) since they do not fuse multiple convolutional layers.

6 EVALUATION

6.1 Fusion effect: comparing FCMs to LBL

This section analyzes the effect of fusion using various workloads, two precisions, and three GPUs. Figures 6 and 7 show the speedup achieved as a result of the fusion in the 24 cases (Table 2) on the three GPUs using FP32 and INT8 precision. FCMs outperform LBL in 67 out of the 72 experiments. The maximum achieved speedup using FP32 is 1.6x in the case of F8 on Orin and 1.8x using INT8 in the case of F1_8 on RTX. The average speedups are 1.3x and 1.4x using FP32 and INT8 respectively. Orin and RTX have the best average speedups using FP32 and INT8, respectively. GTX has the lowest speedups in both cases. In the rest of this section, we discuss three factors that determine the speedup achieved by FCMs. We then explore the fusion effect on different GPUs and finally with different precisions.
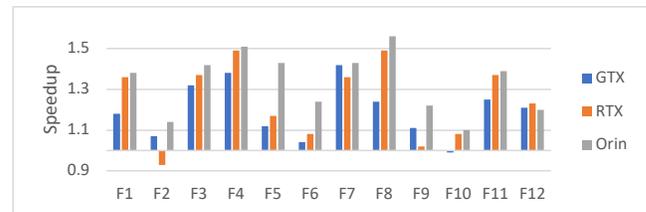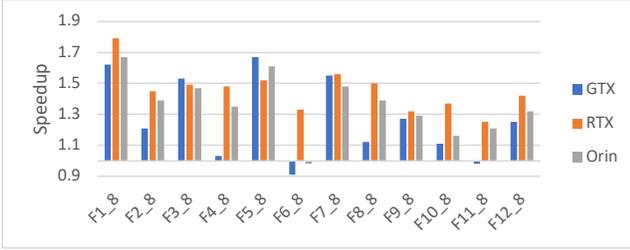

Figure 6: Speedup of FCMs over LBL using FP32

Factors that determine FCMs speedup: The first factor governing the effect of fusion on speedup is whether the fused kernels

Table 2: Fusion cases and their ratios of redundant computations. F1-F12 using FP32, and F1_8-F12_8 using INT8.

DNN	Mob_v1	Mob_v1	Mob_v2	Mob_v2	XCe	XCe	Prox	Prox	CeiT	CeiT	CMT	CMT
FP32	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12
	PWDW_R	PWDW_R	DWPW	PWDW_R	PWDW_R	PWDW_R	DWPW	PWDW_R	PWDW	PWDW_R	PWDW	PWDW_R
	7%	13%	-	18%	4%	7%	-	10%	-	16%	-	13%
INT8	F1_8	F2_8	F3_8	F4_8	F5_8	F6_8	F7_8	F8_8	F9_8	F10_8	F11_8	F12_8
	DWPW	PWDW	DWPW	PWPW	DWPW	PWDW_R	DWPW	PWPW	PWDW	PWDW	PWPW	PWDW
	-	-	-	-	-	15%	-	-	-	-	-	-

Table 3: Categorizing the FP32 LBL and FCMs into compute (C) and memory-bound (M) based on Roofline analysis.

		F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12
GTX	LBL	M, M	C, M	M, M	M, M	C, M	C, M	M, M	M, M	C, M	C, M	C, M	C, M
	FCM	C	C	M	C	C	C	M	C	C	C	C	C
RTX	LBL	M, M	C, M	M, M	M, M	C, M	C, M	M, M	M, M	C, M	C, M	M, M	C, M
	FCM	M	C	M	M	C	C	M	M	C	C	C	C

Figure 7: Speedup of FCMs over LBL using INT8

are **memory- or compute-bound**. In general, the memory access reduction is translated to speedup for memory-bound but not for compute-bound kernels. Table 3 shows which kernels fall under each category. In the case of RTX, F1, 3, 4, 7, 8, and 11, which consist of two-memory-bound layers, have higher speedups than the rest (Figure 6). The average speedup of these six layers is 1.4x compared to 1.1x for the other six. The same applies to GTX, the five cases where both layers are memory-bound have an average speedup of 1.3x compared to 1.1x for the rest. Speedups among layers within each category, compute-bound and memory-bound, are determined by the **amount of reduction in memory access time**. Figure 8 shows the global memory access time of both FCM and LBL executions normalized to that of LBL. For example, among the RTX six FCMs where both layers are memory-bound, F4 has the highest memory access reduction resulting in the highest speedup. And F12 has the highest memory access reduction resulting in the highest speedup among the six cases where at least one layer is compute-bound. However, there are some exceptions. The third factor, the existence of **redundant computations**, explains these exceptions. For example, on GTX, F7 has the highest speedup among FCMs where both layers are memory-bound even though F4 experiences a larger reduction of the memory access time. This is because, unlike F7, F4 has 18% redundant computations (Table 2).

There are two cases where there is a non-negligible slowdown. These cases are F2 on RTX and F6_8 on GTX. The reduction in memory access is not translated into speedup in these cases. The three factors discussed explain this slowdown. For example, in the case of F2 on RTX (Figure 6) not both layers are memory-bound (Table 3), the memory access reduction is relatively low (Figure 8), and there are redundant computations (Table 2).

Speedup across GPUs: Orin and RTX have higher speedups than GTX. Moreover, out of the five cases where FCMs do not have speedup over LBL, three are on GTX compared to one on

Orin and One on RTX. One reason is that GTX has the smallest L1/shared memory per SM (Table 1). This gives less room to the tiles competing on this memory, including the communication buffer (Section 4). Another reason is that GTX has fewer CUDA cores (Table 1), making fewer LBL kernels memory-bound. Table 3 shows examples of that. First, RTX has 6 out of 12 cases where both kernels are memory-bound compared to 5 out of 12 on GTX. Secondly, among the 5 memory-bound cases on both GPUs, 3 remain memory-bound after fusion on RTX, namely F1, F4, and F8. As long as a kernel is memory-bound, reducing global memory access time is, ideally, purely translated into speedup. However, on GTX, these turn into compute-bound when fused, meaning that their performance benefited from the global memory access reduction only partially. To summarise, our method identifies fusions that are advantageous across different GPUs. However, the fusion effect on performance varies depending on the GPU compute and memory resources.

Speedup and precision: Note that the FP32 FCMs are different from the INT8 ones, but both precision's FCMs are representative of layers selected by FusePlanner given the same DNN models (Section 5). Hence, we here comment on the general trends rather than having case-by-case comparisons. The maximum and the average speedups are higher using INT8 compared to FP32. This is mainly because reducing the data size allows the L1/shared memory to fit larger tiles (Section 4). This in turn permits fusion types that are not feasible in FP32. For example, as Table 2 shows, the dominant FCM using FP32 is PWDW_R which requires redundant computation, but in INT8 there is only one PWDW_R. In other words, most INT8 fusions do not have redundant computations making the fusion effect more apparent.

6.2 Comparison with CuDNN

Figure 9 shows a comparison between FCMs and CuDNN, and the speedup of FCMs over the best cuDNN algorithms, namely IMPL_PRECOMP_GEM. The maximum speedup is 3.7x, and the average is 2x. Our LBL implementations also outperform CuDNN in all cases and achieve a maximum speedup of 3x, and an average speedup of 1.5x. Although 50% of the speedup of FCMs over CuDNN, on average, comes from the speedup of LBL over CuDNN; the pure speedup resulted from fusing reaches up to 91% of the overall speedup. When comparing cuDNN implementations, the implicit GEMM implementations outperform direct GEMM. Implicit GEMMs do not explicitly form the matrix that holds the input data resulting in fewer memory accesses. Compared to implicit IMPL_PRECOMP_GEM, the best among the three CuDNN algorithms, our LBL implementations save up to 63% of the global memory accesses, and FCMs save up to 83%. Generally speaking, the results trend is similar to the one discussed in the previous section. For example, in the cases where the pair is composed of memory-bound layers on RTX and GTX, namely F1, 3, 4, 7, and 8, FCMs have relatively high speedups over cuDNN. In addition, both

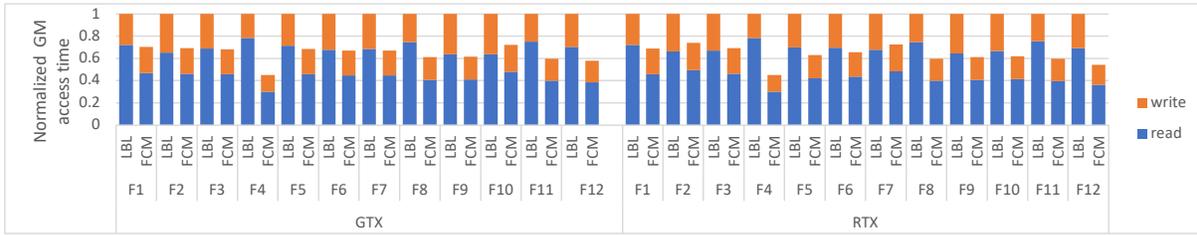Fareed Qararyah, Muhammad Waqar Azhar, Mohammad Ali Maleki, and Pedro Trancoso

Figure 8: Global memory access time of FCMs compared to LBL using FP32. The breakdown shows loads and stores' contribution. All values are normalized to the total global memory access time of the LBL case
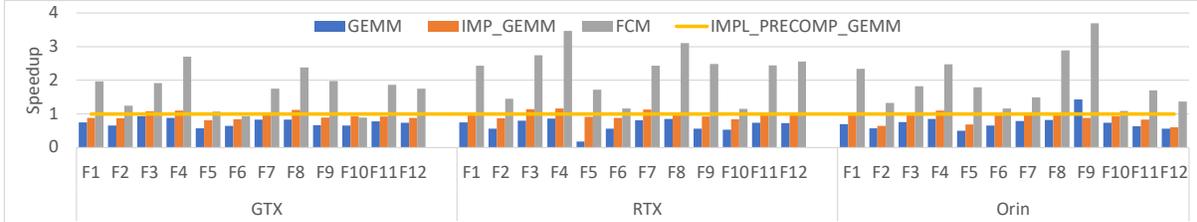

Figure 9: Speedup of FCMs over CuDNN using FP32. All values are normalized to IMPL_PRECOMP_GEMM.

RTX and Orin experience higher speedups compared to GTX. Both FCMs and LBL outperform cuDNN in the INT8 case as well. This is implicitly shown in the next section as we compare our results with TVM implementations configured to use cuDNN in the backend.

6.3 End-to-end comparison with TVM

This section compares end-to-end CNN implementations based on the proposed FCMs and FusePlanner-suggested LBL kernels to TVM. In our end-to-end implementations, FusePlanner iterates over the models' DAGs and suggests which layers to fuse and the tiling parameters that minimize the global memory access for both FCMs and LBL kernels (Section 4). Then the CNNs are implemented accordingly. The fused layers range from 46-58% of the convolutional layers of the four CNNs.

Figure 10a and 10b show the speedup of our implementations over TVM for the four CNNs using FP32 and INT8. Our implementations constantly outperform TVM, achieving maximum speedups of $1.6x$ and $1.8x$ and average speedups of $1.4x$ and $1.5x$ using FP32 and INT8, respectively. Different DNN GPU combinations have different speedups, but Mob_v1 has, on average, the highest speedup. Mob_v1 has a simple linear structure, but TVM graph optimizations are more impactful for DNNs with complex DAGs.

Figure 11a and 11b show the benefit of our implementations on energy efficiency, *i.e.* energy per inference. They show the energy-per-inference of our implementations normalized to that of TVM. On average, our implementations consume 0.59 and 0.54 of the energy consumed by TVM using FP32 and INT8 respectively. Using FP32, The lowest energy consumption is 0.34 of TVM's, which is the case of Mob_v1 on Orin. Using INT8, the lowest is 0.35 of TVM's in the case of Mob_v2 on Orin. Generally speaking, RTX and Orin have higher energy savings compared to GTX. An important observation is that energy savings are, on average, higher than the reduction in running time. This suggests that even when fusion does not improve the latency considerably, e.g. in cases of compute-bound convolutions, reducing the global memory access is still beneficial as it reduces energy consumption.
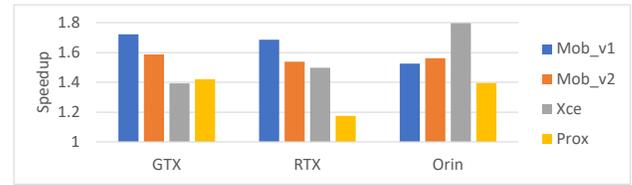
7 RELATED WORK

Layer-fusion is a key inter-layer optimization in many state-of-the-art DNN accelerators [2, 6, 12, 16, 31, 34, 42, 44, 46, 47, 54]. It enables processing the intermediate results immediately, which eliminates the need to frequently access main memory [2, 6, 16]. Fusion is also used to maintain high throughput on heterogeneous accelerators that process different CNN layers using multiple layer-custom engines [30, 31, 34, 38, 44]. In sparse DNNs, where fewer effectual operations and data-reuse opportunities are present within a layer, fusion maintains a reasonable efficiency by offering higher levels of reuse, and resource utilization [47].

The prior art has demonstrated the advantages of layer fusion on DNN accelerators. However, each has its own shortcomings. Alwani *et al.* [2] and Xiao *et al.* [44] proposals handle only linear CNNs like AlexNet and VGG [20, 35]. Zheng *et al.* [54], Zhuang et al. [55], and Jeong *et al.* [16] assume that fusion suffers from an inherent limitation which is the amount of redundant computations that scale with the number of fused layers. However, others [2, 6] have shown that redundant computations can be avoided at the cost of affordable extra buffering. Xing *et al.* [46] and Wei *et al.* [42] optimize the execution time but do not consider minimizing DRAM accesses as a main objective. Olyaiy *et al.* [28] proposed a fusing technique that targets the bottleneck block structures, the proposed technique reduces the multiplications by up to $20x$ at the cost of extra additions. FINN variants [4, 38] focus on aggressively quantized models with binary or ternary weights and intermediate results. Yang *et al.* [47] fuse layers of highly-sparse models. Other fusion and pipelining proposals, e.g. [12, 25, 34], work at the granularity of different inputs or batches. Working at such high granularity increases the inference latency and the off-chip traffic. Convfusion [41] proposes hardware-agnostic fusion but leaves supporting DW convolution and SIMD to future work.

On GPUs, the most common forms of fusion fall under the categories described by TVM authors [7]. First, multiple *injective*, or one-to-one e.g. *add* operators, are fused. Second, an *injective* operator is fused with a *reduction* operator. Third, the convolution
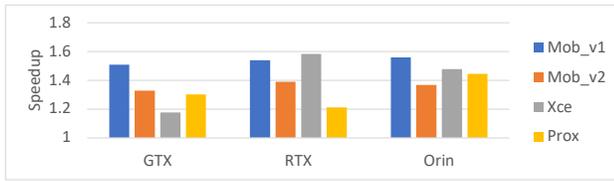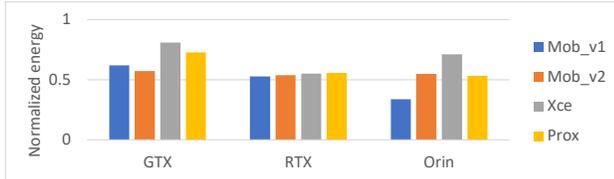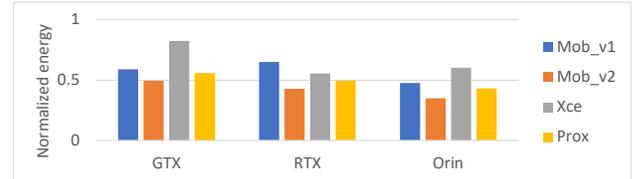
(a) Speedup over TVM using FP32. (b) Speedup over TVM using INT8
Figure 10: Speedup of CNN implementations using FCMs and FusePlanner suggested LBL kernels over TVMs'

(a) Energy per inference using FP32 normalized to TVM (b) Energy per inference using INT8 normalized to TVM
Figure 11: Energy efficiency of CNN implementations using FCMs and FusePlanner suggested LBL kernels compared to TVM

operator is fused with one or more element-wise operators like normalization and non-linearity. Jia *et al.* [17] propose a technique to fuse stages of the Winograd convolution algorithm. Li *et al.* [24] propose to fuse softmax layer implementation to reduce its memory accesses. Chimera [53] fuses multiple convolutions on GPUs but does not support DW and the modeling of inter-block optimizations and data movement estimations don't directly apply to DW.

Unlike the prior art, in this work, we explore fusing DW and PW convolutions to overcome memory access bottlenecks on GPUs. We identify the fusion challenges and trade-offs given the GPU architecture and propose cost models and a set of fused kernels that minimize these convolutions' global memory accesses leading to low latency and energy-efficient inference.

8 CONCLUSION

Depthwise and pointwise convolutions are used to design compact DNNs. However, they have a lower compute-to-memory access ratio than the standard convolution, making their global memory access often a bottleneck. This paper proposes fusion as a technique to reduce these convolutions' global memory accesses on GPUs leading to improvements in their efficiency. We propose a set of novel fused convolutional modules (FCMs), GPU kernels composed of fused depthwise and pointwise convolutions. We also propose Fuse-Planner which consists of cost models to estimate global memory access of layer-by-layer and FCM kernels. Given a GPU architecture, FusePlanner decides when to fuse, and which FCMs to use. Our experiments show that FCMs achieve up to 1.8x speedup over a layer-by-layer implementation and up to 3.7x over cuDNN. End-to-end implementations of four CNNs using the proposed kernels achieve up to 1.8x speedup compared to TVM-optimized models and consume as little as 34% TVM-optimized models energy.

ACKNOWLEDGMENTS

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).
[2] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–12.
[3] James Bergstra, Frédéric Bastien, Olivier Breuleux, Pascal Lamblin, Razvan Pascanu, Olivier Delalleau, Guillaume Desjardins, David Warde-Farley, Ian Goodfellow, Arnaud Bergeron, et al. 2011. Theano: Deep learning on gpus with python. In *NIPS 2011, BigLearning Workshop, Granada, Spain*, Vol. 3. Citeseer.
[4] Michaela Blott, Thomas B Preußer, Nicholas J Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. 2018. FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 11, 3 (2018), 1–23.
[5] Han Cai, Ligeng Zhu, and Song Han. 2018. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332* (2018).
[6] Xuyi Cai, Ying Wang, and Lei Zhang. 2021. Optimus: towards optimal layer-fusion on deep learning processors. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. 67–79.
[7] 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 (OSDI 18)*. 578–594.
[8] 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).
[9] François Chollet. 2017. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1251–1258.
[10] Zihang Dai, Hanxiao Liu, Quoc V Le, and Mingxing Tan. 2021. Coatnet: Marrying convolution and attention for all data sizes. *Advances in neural information processing systems* 34 (2021), 3965–3977.
[11] Shi Dong, Xiang Gong, Yifan Sun, Trinayan Baruah, and David Kaeli. 2018. Characterizing the microarchitectural implications of a convolutional neural network (cnn) execution on gpus. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. 96–106.
[12] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. 2019. Tangram: Optimized coarse-grained dataflow for scalable nn accelerators. In

Fareed Qararyah, Muhammad Waqar Azhar, Mohammad Ali Maleki, and Pedro Trancoso

Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. 807–820.

[13] Jianyuan Guo, Kai Han, Han Wu, Yehui Tang, Xinghao Chen, Yunhe Wang, and Chang Xu. 2022. Cmt: Convolutional neural networks meet vision transformers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.* 12175–12185.

[14] 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.

[15] 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).

[16] Hyuk-Jin Jeong, JiHwan Yeo, Cheongyo Bahk, and JongHyun Park. 2023. Pin or Fuse? Exploiting Scratchpad Memory to Reduce Off-Chip Data Transfer in DNN Accelerators. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization.* 224–235.

[17] Liancheng Jia, Yun Liang, Xiuhong Li, Liqiang Lu, and Shengen Yan. 2020. Enabling efficient fast convolution algorithms on GPUs via MegaKernels. *IEEE Trans. Comput.* 69, 7 (2020), 986–997.

[18] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia.* 675–678.

[19] Salman Khan, Muzammal Naseer, Munawar Hayat, Syed Waqas Zamir, Fahad Shahbaz Khan, and Mubarak Shah. 2022. Transformers in vision: A survey. *ACM computing surveys (CSUR)* 54, 10s (2022), 1–41.

[20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2017. Imagenet classification with deep convolutional neural networks. *Commun. ACM* 60, 6 (2017), 84–90.

[21] Andrew Lavin and Scott Gray. 2016. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition.* 4013–4021.

[22] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.

[23] Yann LeCun, Koray Kavukcuoglu, and Clément Farabet. 2010. Convolutional networks and applications in vision. In *Proceedings of 2010 IEEE international symposium on circuits and systems.* IEEE, 253–256.

[24] Chao Li, Yi Yang, Min Feng, Srimat Chakradhar, and Huiyang Zhou. 2016. Optimizing memory efficiency for deep convolutional neural networks on GPUs. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE, 633–644.

[25] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang. 2016. A high performance FPGA-based accelerator for large-scale convolutional neural networks. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL).* IEEE, 1–9.

[26] Gangzhao Lu, Weizhe Zhang, and Zheng Wang. 2021. Optimizing depthwise separable convolution operations on gpus. *IEEE Transactions on Parallel and Distributed Systems* 33, 1 (2021), 70–87.

[27] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for? *Queue* 6, 2 (2008), 40–53.

[28] MohammadHossein Olyaiy, Christopher Ng, and Mieszko Lis. 2021. Accelerating DNNs inference with predictive layer fusion. In *Proceedings of the ACM International Conference on Supercomputing.* 291–303.

[29] 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).

[30] Fareed Qararyah, Muhammad Waqar Azhar, and Pedro Trancoso. 2022. FiBHA: Fixed Budget Hybrid CNN Accelerator. In *2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD).* IEEE, 180–190.

[31] Fareed Qararyah, Muhammad Waqar Azhar, and Pedro Trancoso. 2024. An Efficient Hybrid Deep Learning Accelerator for Compact and Heterogeneous CNNs. *ACM Transactions on Architecture and Code Optimization* 21, 2 (2024), 1–26.

[32] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition.* 4510–4520.

[33] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. 2018. Modeling relational data with graph convolutional networks. In *The Semantic Web: 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, Proceedings 15.* Springer, 593–607.

[34] Yongming Shen, Michael Ferdman, and Peter Milder. 2017. Maximizing CNN accelerator efficiency through resource partitioning. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 535–547.

[35] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[36] David R So, Wojciech Mańke, Hanxiao Liu, Zihang Dai, Noam Shazeer, and Quoc V Le. 2021. Primer: Searching for efficient transformers for language modeling. *arXiv preprint arXiv:2109.08668* (2021).

[37] Mingxing Tan and Quoc Le. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning.* PMLR, 6105–6114.

[38] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays.* 65–74.

[39] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[40] Rangharajan Venkatesan, Yakun Sophia Shao, Miaorong Wang, Jason Clemons, Steve Dai, Matthew Fojtik, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, et al. 2019. Magnet: A modular accelerator generator for neural networks. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD).* IEEE, 1–8.

[41] Luc Waeijen, Savvas Sioutas, Maurice Peemen, Menno Lindwer, and Henk Corporaal. 2021. ConvFusion: A model for layer fusion in convolutional neural networks. *IEEE Access* 9 (2021), 168245–168267.

[42] Xuechao Wei, Yun Liang, Xiuhong Li, Cody Hao Yu, Peng Zhang, and Jason Cong. 2018. TGPA: Tile-grained pipeline architecture for low latency CNN inference. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD).* ACM, 1–8.

[43] Haiping Wu, Bin Xiao, Noel Codella, Mengchen Liu, Xiyang Dai, Lu Yuan, and Lei Zhang. 2021. Cvt: Introducing convolutions to vision transformers. In *Proceedings of the IEEE/CVF international conference on computer vision.* 22–31.

[44] Qingcheng Xiao, Yun Liang, Liqiang Lu, Shengen Yan, and Yu-Wing Tai. 2017. Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on FPGAs. In *Proceedings of the 54th Annual Design Automation Conference 2017.* 1–6.

[45] Tete Xiao, Mannat Singh, Eric Mintun, Trevor Darrell, Piotr Dollár, and Ross Girshick. 2021. Early convolutions help transformers see better. *Advances in neural information processing systems* 34 (2021), 30392–30400.

[46] Yu Xing, Shuang Liang, Lingzhi Sui, Xijie Jia, Jiantao Qiu, Xin Liu, Yushun Wang, Yi Shan, and Yu Wang. 2019. Dnnvm: End-to-end compiler leveraging heterogeneous optimizations on fpga-based cnn accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2019), 2668–2681.

[47] Yifan Yang, Joel S Emer, and Daniel Sanchez. 2023. ISOSceles: Accelerating Sparse CNNs through Inter-Layer Pipelining. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA).* IEEE, 598–610.

[48] Qihang Yu, Yingda Xia, Yutong Bai, Yongyi Lu, Alan L Yuille, and Wei Shen. 2021. Glance-and-gaze vision transformer. *Advances in Neural Information Processing Systems* 34 (2021), 12992–13003.

[49] Kun Yuan, Shaopeng Guo, Ziwei Liu, Aojun Zhou, Fengwei Yu, and Wei Wu. 2021. Incorporating convolution designs into visual transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision.* 579–588.

[50] Dan Zhang, Safeen Huda, Ebrahim Songhori, Kartik Prabhu, Quoc Le, Anna Goldie, and Azalia Mirhoseini. 2022. A full-stack search technique for domain optimized deep learning accelerators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems.* 27–42.

[51] Si Zhang, Hanghang Tong, Jiejun Xu, and Ross Maciejewski. 2019. Graph convolutional networks: a comprehensive review. *Computational Social Networks* 6, 1 (2019), 1–23.

[52] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition.* 6848–6856.

[53] 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 (HPCA).* IEEE, 1113–1126.

[54] Shixuan Zheng, Xianjue Zhang, Daoli Ou, Shibin Tang, Leibo Liu, Shaojun Wei, and Shouyi Yin. 2020. Efficient scheduling of irregular network structures on CNN accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3408–3419.

[55] Weihao Zhuang, Tristan Hascoet, Xunquan Chen, Ryoichi Takashima, Tetsuya Takiguchi, Yasuo Ariki, et al. 2021. Convolutional Neural Networks Inference Memory Optimization with Receptive Field-Based Input Tiling. *APSIPA Transactions on Signal and Information Processing* 12, 1 (2021).