Article:
Lu, G, Zhang, W and Wang, Z orcid.org/0000-0001-6157-0662 (2021) Optimizing Depthwise Separable Convolution Operations on GPUs. IEEE Transactions on Parallel and Distributed Systems. p. 1. ISSN 1045-9219

https://doi.org/10.1109/tpds.2021.3084813

Optimizing Depthwise Separable Convolution Operations on GPUs

Gangzhao Lu, Weizhe Zhang, *Senior Member, IEEE,* and Zheng Wang

Abstract—The depthwise separable convolution is commonly seen in convolutional neural networks (CNNs), and is widely used to reduce the computation overhead of a standard multi-channel 2D convolution. Existing implementations of depthwise separable convolutions target accelerating model training with large batch sizes with a large number of samples to be processed at once. Such approaches are inadequate for small-batch-sized model training and the typical scenario of model inference where the model takes in a few samples at once. This paper aims to bridge the gap of optimizing depthwise separable convolutions by targeting the GPU architecture. We achieve this by designing two novel algorithms to improve the column and row reuse of the convolution operation to reduce the number of memory operations performed on the width and the height dimensions of the 2D convolution. Our approach employs a dynamic tile size scheme to adaptively distribute the computational data across GPU threads to improve GPU utilization and to hide the memory access latency. We apply our approach on two GPU platforms: an NVIDIA RTX 2080Ti GPU and an embedded NVIDIA Jetson AGX Xavier GPU, and two data types: 32-bit floating point (FP32) and 8-bit integer (INT8). We compared our approach against cuDNN that is heavily tuned for the NVIDIA GPU architecture. Experimental results show that our approach delivers over $2\times$ (up to $3\times$) performance improvement over cuDNN. We show that, when using a moderate batch size, our approach averagely reduces the end-to-end training time of MobileNetV2 and EfficientNet-B0 by 9.7% and 7.3% respectively, and reduces the end-to-end inference time of MobileNet and EfficentNet by 12.2% and 13.5% respectively.

Index Terms—Performance Optimization, Convolution, Depthwise, Pointwise, Memory Optimization, GPU Utilization

1 Introduction

In recent years, deep neural networks (DNNs) have made astonishing success in solving a wide range of tasks [1], [2], [3], [4], [5], [6]. One of the most successful DNN architectures is the convolutional neural network (CNN) that is widely used in tasks like image classification [1], [2], object detection [3], [4] and semantic segmentation [5], [6]. CNN models are typically trained and run on GPUs due to their computation requirements.

The depthwise separable convolution (DSC) is widely used in modern CNN models for accelerating model computation time [7], [8], [9], [10]. This operation can process both the spatial dimensions (e.g., the width and the height of an image) and the depth dimension (e.g., the RGB channels of an image) of an input. It achieves this by splitting a convolution kernel into two separate kernels that perform two convolutions: a depthwise convolution and a pointwise convolution. The former applies a single convolutional filter for each input channel, and the latter uses a 1×1 kernel to iterate through every single point of the input (e.g., the kernel has a depth of however many channels the input image has). Compare to a classical 2D convolution that operates on a 2D input of $channels \times height \times width$, the DSC reduces the number of multiplication operations as well as the number of parameters needed for the convolution filter (and hence the chance of model over-fitting) as well as computation time by having fewer arithmetic operations. For this reason, the DSC is widely used in latency-sensitive scenarios, such as using a trained CNN on embedded devices or performing distributed, on-device learning on resource-constrained systems [11].

A wide range of optimization techniques have been proposed to perform convolutions [12], [13], [14], [15], [16], [17], [18], [19]. Among these techniques, the fast fourier transform (FFT) [15], winograd (Winograd) [16] and general matrix multiplication (GEMM) [17], [18], [19] are broadly adopted. However, FFT and Winograd offer little benefit for depthwise convolutions compared to standard 2D convolution. This is because FFT and Winograd are designed to optimize arithmetic computation [20], [16], but not memory accesses. However, the memory access latency often dominates the execution time of depthwise convolution [21] due to its lower arithmetic operations compared to a standard 2D convolution. Both methods are also ill-suited for pointwise convolutions (that apply a 1×1 kernel) because FFT is designed to operate on a large filter and Winograd works best when the filter size is 3×3.

While GEMM is a good fit for pointwise convolution (that is also adopted by cuDNN [22]), the current implementation of GEMM for CNNs can lead to poor GPU performance during model deployment. A typical GEMM implementation uses a fixed tile size to distribute work across parallel threads, without taking into consideration the amount of computation required. As we will show later in the paper, such a strategy cannot make effective use of the GPU parallelism when the batch size (i.e., the number of samples to be processed at once) is small (e.g., $<= 128$). The ineffective use of GPU resources leads to low GPU utilization and sub-optimal performance. This is a particular

- *G. Lu and W. Zhang are with the School of Cyberspace Science at Harbin Institute of Technology, Harbin 150000, China. E-mail:{lugangzhao,wzzhang}@hit.edu.cn*
- *Z. Wang is with the School of Computing at University of Leeds, United Kingdom. E-mail: z.wang5@leeds.ac.uk*

problem for two real-life scenarios: when running a trained model for inferencing - where the model typically only takes in one or a few samples (and hence a small batch size) - or performing on-device distributed training on an embedded device - where the number of training samples is likely to be small due to resource constraints.

Our work addresses the memory latency and work distribution issues identified above. By addressing these two issues together, our approach enables efficient DSC because it accelerates not only depthwise convolution by reducing the GPU memory access latency and also pointwise convolution for model inference and small-batch-sized training.

To improve the memory performance of depthwise convolution, we introduce two novel optimization techniques for operations performed on rows and columns. Our approach reduces the number of memory accesses required by reusing data. To improve column data reuse, we use the shuffle instructions (supported by both CUDA and OpenCL and hence is applicable to mainstream GPUs) to exchange elements among threads within a GPU warp (or working group). In this way, we can avoid reloading the same elements shared among different threads. We also apply shuffle instructions to converting dynamic indices to static indices to assist register promotion, an optimization strategy that is not exploited in previous studies [23], [24]. To increase row data reuse, we multiply one input row with multiple rows of a convolutional kernel (or filter) to compute multiple output elements at the same time. This strategy improves the data locality of elements within a row, reducing the number of memory transactions compared with that of the existing convolution processing pipeline. By reducing the number of memory accesses, our approach improves the performance of depthwise convolution.

To overcome the drawback of fixed tile size work partition of a GEMM kernel for pointwise convolution, we employ a dynamic tile size scheme. Our approach first adjusts the work assigned to each GPU thread so that we have a sufficient number of tiles to be distributed to GPU threads to improve the GPU utilization. A challenge here is how to assign the right amount of work to GPU threads so that the global memory access latency can be adequately hidden through computation. Having too little work per GPU thread will make the GPU memory access dominates the execution while having too large work assignment will lead to low GPU utilization (as only a small number of GPU threads will receive a tile to work on). To this end, our dynamic scheme distributes input or filter channels across threads within a warp to minimize memory latency with improved GPU parallelism. Recent works [18], [25] employ a heuristic method to maximize parallelism for GEMM. They achieve this by trying to combine multiple convolutions that can be computed concurrently into one GEMM kernel. Such a strategy assumes multiple parallel convolutions can be performed within a single GEMM kernel. However, this strong assumption is only valid for some special CNN structures like the inception layer in GoogleNet [26]. As a result, these prior methods are not applicable to the more general case of CNNs where convolution operations must be performed sequentially due to dependence. Our dynamic work distribution strategy does not rely on this assumption and hence is more generally applicable compared to these prior approaches.

We evaluate our approach by applying it to both depthwise and pointwise convolutions with FP32 and INT8 on two GPU platforms: an NVIDIA RTX 2080Ti GPU and an embedded NVIDIA Jetson AGX Xavier GPU. We compared our approach against cuDNN, an industry strengthened DNN library that is heavily optimized for the NVIDIA GPU architecture. Experimental results show that our approach delivers over $3\times$ and $2\times$ faster performance than cuDNN for depthwise and pointwise convolutions, respectively. We show that, when the batch size is $<= 128$, our approach averagely improves the end-to-end training performance of MobileNetV2 [27], [7] and EfficientNet-B0 [8] by 11.5% and 7.3% respectively, and improves the end-to-end inference performance of MobileNetV2 and EfficentNet-B0 by 9.7% and 11.6% respectively.

This paper makes the following technical contributions:

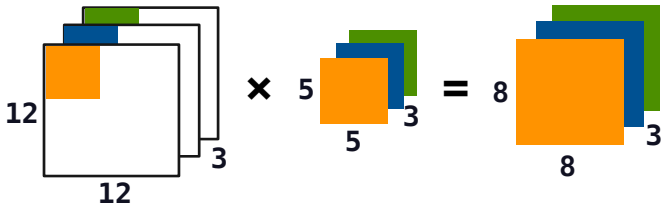
- It presents two novel algorithms for column and row reuse (Section 3) for depthwise convolution, improving the data locality and the memory access latency for depthwise convolution.
- It describes a novel method for transforming dynamic indices into static indices to assist register promotion for performance optimization (Section 3.1.3).
- It presents a dynamic tile size scheme for pointwise convolution, increasing GPU utilization while minimizing the global memory access latency (Section 4).

This work extends our prior work [28] by proposing a new dynamic tile size scheme to optimize pointwise convolution, which is a key component of depthwise separable convolution. We also added new experiments performed on embedded devices and using 8-bit integers for the neural networks. The new experiments demonstrate the robustness of the proposed approach, showing that it consistently outperforms cuDNN by delivering the overall best performance.

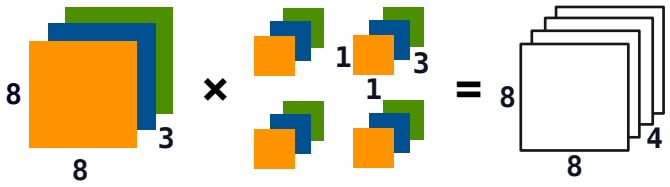
2 Background

2.1 GPU Architecture

Deep learning models are often trained and executed on the GPU. Modern GPUs employ a complex execution pipeline and memory hierarchy to support concurrent execution of parallel threads. A typical GPU consists of multiple Streaming Multiprocessors (SMs). Each SM includes multiple Single-Instruction-Multiple-Thread (SIMT) units, each of which has multiple lanes of execution. Threads scheduled in the same SIMT unit are called a warp, which is the smallest scheduling unit in GPU. Like a modern CPU, a GPU consists of multiple memory hierarchies. The thread-local registers are the fastest memory component, having the lowest access latency (1-2 cycles). The SM local L1 caches and shared memory provide a larger storage capacity over the thread-local registers but have modestly higher accessing latency of around 30 cycles [29], [30]. All the SMs share a unified L2 cache that provides an accessing latency of about 200 cycles. The off-chip global memory, similar to the RAM in a CPU system, provides the largest memory storage capacity on the GPU but has the longest accessing latency of around 500 cycles measured through running micro-benchmarks on



(a) Depthwise convolution: three 5×5 2D filters are used to convolve with one 3-channel 12×12 input and generate one 3-channel 8×8 output.



(b) Pointwise convolution: four 3-channel 1×1 filters are used to convolve with one 3-channel 8×8 input and generate one 4-channel 8×8 output.

Fig. 1. Demonstration of depthwise and pointwise convolutions.

NVIDIA RTX 2080Ti GPU used in this work. Local memory resides in global memory and is used to hold variables with dynamic indexing or too large to fit into registers. It has the same access latency as global memory. The key to optimizing memory performance is to make use of the fast memory sub-systems (i.e., registers and shared memory) and reduce the number of memory accesses to slower memory. Our work is designed to provide such capabilities for depthwise separable convolution operations.

2.2 Depthwise Separable Convolution

Our work targets depthwise separable convolution (DSC) that is widely used by CNN models to reduce the number of multiplication operations needed for doing convolution (a standard operation in CNN). The DSC splits a standard (e.g., multi-channeled) 2D convolution kernel into two individual kernels: a depthwise convolution kernel and a pointwise convolution kernel.

The depthwise convolution kernel processes one input channel at a time, and stacks the outputs of all channels together to form an $c \times n \times n$ matrix, where $n \times n$ is the output of a depthwise convolution kernel and c is the number of channels to be processed. Specifically, it takes as input a feature map and applies a bank of 2D filters (e.g., an $N \times N$ kernel) on the width and height directions of the input. We iteratively apply the depthwise convolution kernel to all channels. Fig. 1a shows an example of depthwise convolution, where three 5×5 2D filters are used to convolve with the corresponding channel of a $3 \times 12 \times 12$ feature map and generate one $3 \times 8 \times 8$ output.

The output of the depthwise convolution kernel is fed into a pointwise convolution kernel which uses a 1×1 filter to iterate through every single point. This kernel has a depth of the number of input channels (i.e., c). The DSC reduces the computation by reducing the number of input transformations needed when compared to a standard depthwise convolution. Fig. 1b shows an example of pointwise convolution, where four $3 \times 1 \times 1$ filters are used to convolve with the $3 \times 8 \times 8$ feature map iteratively and each filter generates one channel of the $4 \times 8 \times 8$ output.

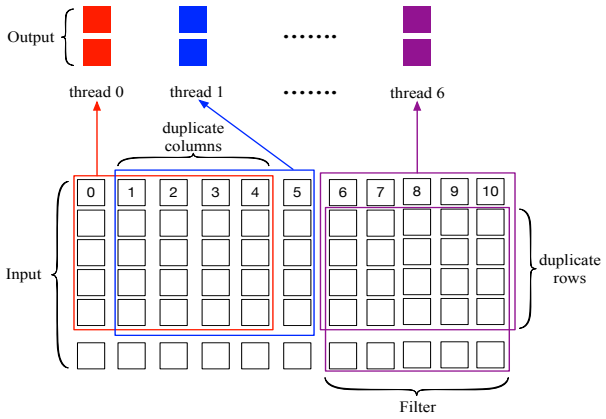


Fig. 2. A working example of performing a depthwise convolution using a GPU. Here, the filter size is 5×5, the input image size is 6×11 and the output size is 2×7.

2.3 Roadmap and Notations

We present our approach for optimizing the two convolutional kernels of DSC in Sections 3 and 4. We start by introducing our methods for improving data locality of depthwise convolution in Section 3 and then presenting our approach for using dynamic work distribution to accelerate small-batch-sized pointwise convolution in Section 4.

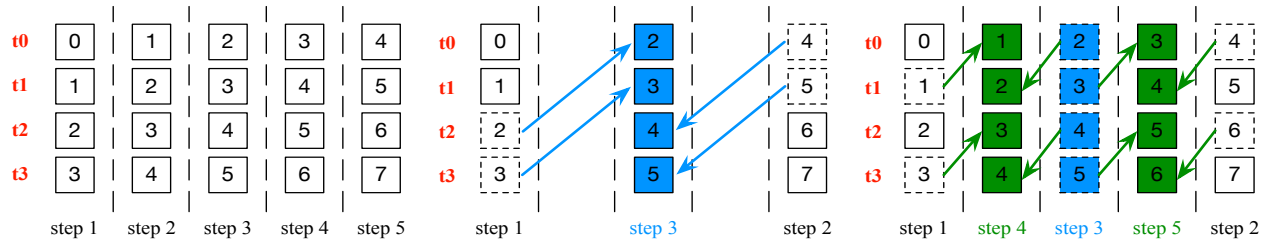
Notations. Throughout the paper, we use I, F, and O to represent the input, the filter, and the output respectively; we also use N, C, H, and W to denote the batch size, the channel, the height, and the width, respectively.

3 Optimizing Depthwise Convolution

In this section, we describe our two optimizations, column reuse (Section 3.1) and row reuse (Section 3.2), for reducing the number of GPU memory accesses for depthwise convolution.

3.1 Column Reuse Optimization

Working example. We use the depthwise convolution with only one channel shown in Fig. 2 as a working example to explain our column reuse method. In practice, we iterate the depthwise convolution kernel on each channel in turn (e.g., the R, G, and B channels of an image). Without loss of generality, we slide a 5×5 filter over a 6×11 input with stride 1 to produce a 2×7 output. Our column reuse method can also be applied to depthwise convolutions with other stride settings. In this example, each thread calculates one column of the output. Two parallel threads 0 and 1 will execute code to slide the filter along the width dimension, where both threads load two overlapped regions from the input image (thereby generating four duplicate columns). Similarly, there will be another thread (thread 6 in this example) to slide the filter along the height dimension, which will load two overlapped regions and generates four duplicate rows.



(a) Direct convolution: Each thread loads 5 input elements from global memory.

(b) Optimized convolution: each thread retrieves its third element from the corresponding thread.

(c) Our approach: each thread retrieves its second and fourth elements from corresponding threads.

Fig. 3. Illustration of direct and optimized convolutions. We use a 5×5 filter and each thread calculates the convolution for one output element. This example shows how a thread processes the first 5 corresponding input elements.

3.1.1 Standard convolution

Fig. 3a shows a standard depthwise convolution operation, operating on a single-channel input on the example shown in Fig. 2. Here, each thread loads the first corresponding input element from the GPU global memory. Given that the indices of these elements are contiguous, i.e., 0, 1, 2, and 3 in this example, concurrent access to these elements will be coalesced to form a single memory transaction. As a result, each step will incur one memory access, five for the five steps (steps 1-5) as shown in Fig. 3a. After completing step 5, each pair of adjacent threads will have four duplicate input elements, corresponding to the duplicate columns in Fig. 2. Specifically, input elements 1, 2 and 3 loaded in step 2 would have already been loaded by threads $t1$, $t2$ and $t3$ in the previous step (Fig. 3a). The repeated load to these elements leads to redundant memory accesses and unnecessary memory access latency. Even the elements may be prefetched to the L1 cache before the next step, access to the L1 cache still takes around 30 cycles on a 2080Ti GPU. To reduce the memory overhead, we would like to avoid such redundant memory accesses.

3.1.2 An optimized implementation

To eliminate the redundant loads, we could use the shuffle instructions supported by both CUDA and OpenCL to exchange input elements among different threads. To this end, we adopt the optimization developed in our prior work [28]. Fig. 3b depicts such an optimization. Specifically, in steps 1 and 2 of Fig. 3b, each thread loads the corresponding first and fifth input elements from the global memory. In step 3, each thread utilizes the shuffle instruction to retrieve the third element from another thread. For example, threads $t0$ and $t1$ could retrieve the third element from threads $t2$ and $t3$, respectively, and provide the fifth element (dashed squares in step 2) for both threads. Similarly, threads $t2$ and $t3$ retrieve the third element from threads $t0$ and $t1$, respectively, and provide the first element (dashed squares in step 1) for threads $t0$ and $t1$. Using the CUDA shuffle instruction, this exchange process can be implemented as $shfl\_xor(iTemp[i], 2)$, where $iTemp$ is a thread-local array used to store the five input elements, and i is the location in the local array. For our working example, threads $t0$ and $t1$ will supply the fifth element, hence $i = 4$. Similarly, threads $t2$ and $t3$ will provide the first element, thus $i = 0$.

While this version reduces the redundant memory accesses compared to a standard convolution implementation, there is still room for improvement. The problem is that the shuffle instruction $shfl\_xor(iTemp[i], 2)$ now becomes a bottleneck because $iTemp$ is accessed through dynamic indexing. Since the indices and the access pattern to $iTemp$ are not available at compile-time, the compiler cannot decide which of the elements in $iTemp$ will be frequently accessed and has to place $iTemp$ in the local memory which would still incur an access latency of around 500 cycles. If we can promote register allocation for $iTemp$, we can then further improve the performance of convolution.

3.1.3 Our approach

Our column reuse scheme (Fig. 3c) converts dynamic indexing to static array accesses to promote register allocation. This strategy is described in Algorithms 1 and 2, where the first algorithm is used for step 3, and the latter is used for steps 4 and 5. Note that these two algorithms can be used for different sized convolution kernels, for which we will discuss in Section 3.1.4.

Fig. 4 gives a working example of Algorithm 1. Here, we first load the corresponding first and fifth input elements into $iTemp$ before passing it to Algorithm 1. Then, we pack two 32-bit elements into a 64-bit variable, $exchange$, where $iTemp[4]$ and $iTemp[0]$ are the high and low 32 bits, respectively (Line 2). As threads $t0$ and $t1$ will provide the fifth element of the data they load, which are the high 32 bits of $exchange$, we right shift $exchange$ for both threads by an offset of 32 to place $iTemp[4]$ in the low 32 bits. Now turning our attention to threads $t2$ and $t3$ that will provide the first element of the data they load. Since the elements are the low 32 bits of $exchange$, we right shift $exchange$ in both threads by an offset of 0. The number of places to be shifted for each thread is calculated based on the thread ID (Line 3). Next, we unpack $exchange$ into $iTemp[2]$ (high 32 bits) and $iTemp[1]$ (low 32 bits) (Line 5). By doing so, we can retrieve the element a thread needs to supply from a fixed location, $iTemp[1]$. Finally, we use the shuffle instruction to exchange the elements among threads (Line 6).

Using Algorithm 1, we can replace dynamic index i in $shfl\_xor(iTemp[i], 2)$ (whose value is unknown at compile time) with a static index, 1 in $shfl\_xor(iTemp[1], 2)$, in our working example. By doing so, we promote register allocation by allowing the compiler to put all the thread-local variables into the fast GPU registers (that have access

Algorithm 1: RetrieveThirdElement

```
// iTemp: Buffer for storing input
   elements loaded from memory or
   generated through shuffle
   instructions.
Input: iTemp
Output: iTemp
1 tid ← threadIdx.x;
2 mov exchange, {iTemp[0], iTemp[4]};
3 shift ← ((tid + 2)&2) << 4;
4 exchange ← exchange >> shift;
5 mov {iTemp[1], iTemp[2]}, exchange;
6 iTemp[2] ← shfl_xor(iTemp[1], 2);
```

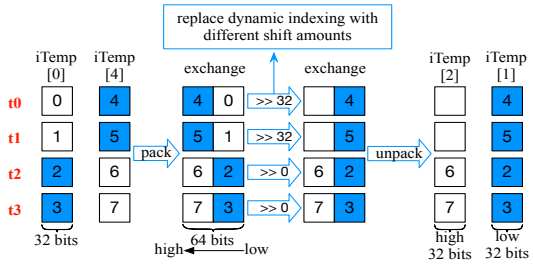


Fig. 4. Convert dynamic indexing of array $iTemp$ into static indexing, allowing $iTemp$ to be allocated in registers instead of the local memory.

latency of 1 to 2 cycles as opposed to 500 cycles when the data are stored in the local memory). Note that this approach will not improve the register usage. Since steps 4 and 5 of our implementation (Fig. 3c) adopt a similar procedure as step 3 in Fig. 3b, we can adapt Algorithm 1 to derive Algorithm 2 with minor modifications. The main distinction between the two algorithms comes from how we process steps 3-5 shown in Fig. 3c. Specifically, we use four threads to exchange the elements in step 3 for Algorithm 1, but use only two adjacent threads in steps 4 and 5 for Algorithm 2. To adapt to the change of the number of threads used, we recalculate the shift offset for each thread (Line 3 of both algorithms) and change the arguments of the shuffle instruction (Line 6 in both algorithms).

For our working example, Algorithms 1 and 2 respectively reduce the number of memory accesses from 5 to 2 and 25 to 10 when 5 and 25 input elements are loaded. This reduction greatly improves the performance of the depthwise convolution.

3.1.4 Generalize to other filter sizes

So far we have described our approach using a concrete working example with a pre-defined filter size, but our algorithms can be generalized to filters with an arbitrary size. To apply our approach to a filter of size $n \times n$, we will first divide the filter into several $n \times 5$ sub-filters. Next, we divide the remaining columns into several $n \times 3$ sub-filters with some overlap columns. Each $n \times 5$ and $n \times 3$ filters can then be directly processed by Algorithm 1 and Algorithm 2.

3.2 Row Reuse Optimization

Working example. Consider now the standard convolution example shown in Fig. 5 as a working example for our row reuse algorithm. When sliding the filter over the 2D input along the height dimension, it produces a column of elements as the output.

Algorithm 2: RetrieveSecondElement

```
Input: iTemp
Output: iTemp
1 tid ← threadIdx.x;
2 mov exchange, {iTemp[0], iTemp[2]};
3 shift ← ((tid + 1)&1) << 5;
4 exchange ← exchange >> shift;
5 mov {iTemp[0], iTemp[1]}, exchange;
6 iTemp[1] ← shfl_xor(iTemp[0], 1);
```

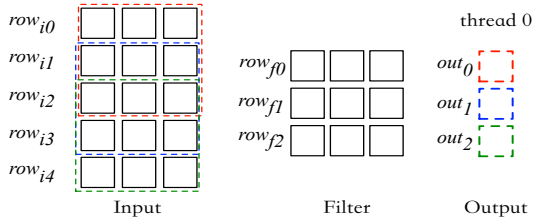


Fig. 5. A 3×3 filter is used to slide over the input image along the height dimension, which produces a column of output elements.

3.2.1 Standard convolution

Assume we use one thread to calculate one column of output elements. For the working example given in Fig. 5, the convolution will be computed as follows:

$$out_0 = row_{i0} \cdot row_{f0} + row_{i1} \cdot row_{f1} + row_{i2} \cdot row_{f2}$$
$$out_1 = row_{i1} \cdot row_{f0} + row_{i2} \cdot row_{f1} + row_{i3} \cdot row_{f2}$$
$$out_2 = row_{i2} \cdot row_{f0} + row_{i3} \cdot row_{f1} + row_{i4} \cdot row_{f2}$$

As can be seen from the above equations, row_{i1} and row_{i3} are loaded twice, and row_{i2} is loaded three times; nine rows are being loaded in total. These redundant loads to the same read-only row thus incur extra memory accesses and additional overhead.

3.2.2 Our optimization

To remove redundant loads to the same row, we redesign the execution flow of the standard depthwise convolution. Specifically, after fetching a row from the input, we compute the number of output elements that depend on the loaded row. With this information in place, we use the loaded row to perform inner products with corresponding rows of the filter to calculate the output elements whose outcomes depending on the loaded row. Our approach translates the execution flow of the working example presented in Fig. 5 to:

$$\begin{aligned}
load\ row_{i0} &: out_0 = row_{i0} \cdot row_{f0} \\
load\ row_{i1} &: out_0 = out_0 + row_{i1} \cdot row_{f1} \\
&\quad\ out_1 = row_{i1} \cdot row_{f0} \\
load\ row_{i2} &: out_0 = out_0 + row_{i2} \cdot row_{f2} \\
&\quad\ out_1 = out_1 + row_{i2} \cdot row_{f1} \\
&\quad\ out_2 = row_{i2} \cdot row_{f0} \\
load\ row_{i3} &: out_1 = out_1 + row_{i3} \cdot row_{f2} \\
&\quad\ out_2 = out_2 + row_{i3} \cdot row_{f1} \\
load\ row_{i4} &: out_2 = out_2 + row_{i4} \cdot row_{f2}
\end{aligned}$$

Algorithm 3: RowReuse

Input: row, $index$, $filter$, Out
Output: Out

1 **if** $index < F_H - 1$ **then**
2 **for** $i \leftarrow 0$ **to** $index + 1$ **do**
3 $Out[i] \leftarrow Out[i] + row \cdot filter[index - i]$;
4 **else if** $index \geq F_H - 1$ ***and*** $index < I_H - F_H + 1$ **then**
5 **for** $i \leftarrow 0$ **to** F_H **do**
6 $o_{index} \leftarrow index - F_H + 1 + i$;
7 $Out[o_{index}] \leftarrow$ $Out[o_{index}] + row \cdot filter[F_H - 1 - i]$;
8 **else**
9 **for** $i \leftarrow F_H - 1$ **to** 0 **do**
10 $o_{index} \leftarrow I_H - F_H + 1$;
11 $Out[o_{index}] \leftarrow Out[o_{index}] + row \cdot filter[F_H - i]$;

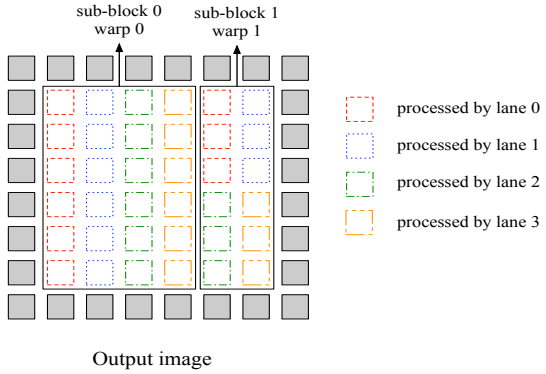


Fig. 6. The output is produced by sliding a 3×3 filter over an 8×8 input with one pad. Here, we assume that the warp size is 4 and thus having $laneid = threadid\%4$.

In this new implementation, we would only issue loads to five rows to calculate the output elements of our working example. Compared to the nine loads required by the standard convolution, we reduce the number of loads to row elements by nearly half. We note that although the number of accesses to the output column out is increased, the overhead is negligible because out is smaller than the size of multiple rows and often can be stored in registers.

We describe a general solution for row reuse in Algorithm 3, where row denotes the row loaded from the input, $index$ denotes the index of row, $filter$ denotes the vector of filter rows and $filter[i]$ means the ith row of the filter. Pseudo codes at Lines 1-5 process the first $F_H - 1$ rows (row_{i0} and row_{i1} in Fig. 5) that are needed by less than F_H output elements. Codes at Lines 6-11 process the rows needed by exact F_H output elements (e.g., row_{i2} in Fig. 5). Finally, codes at Lines 12-17 process the last $F_H - 1$ rows, which are needed by less than F_H output elements (e.g., row_{i3} and row_{i4} in Fig. 5).

Algorithm 3 is designed to eliminate redundant loads to the same row introduced by sliding a filter over the input along the height dimension. By loading each row of the input just once, our approach greatly reduces the number of memory transactions for convolution operations.

3.3 Putting Together

We now take the widely used 2D convolution as an example to illustrate how to apply both reuse algorithms on convolution operations.

Algorithm 4: Optimized Depthwise Convolution

Input: I, F, $subBlockHeight$
Output: O

1 Load the filter into shared memory;
2 Divide columns of the filter into a combination of 3-column and 5-column sub-filters;
3 $\_\_syncthreads()$;
4 **if** $blockIdx.x < gridDim.x - 1$ **then**
5 Init thread local register array sum to zero;
6 Calculate the index of the first input element this thread needs, denoted as $inputIndex$;
7 **for** $i \leftarrow 0$ **to** $subBlockHeight$ **do**
8 **foreach** *sub-filter* **do**
9 Load corresponding input elements from $inputIndex$ of global memory into $iTemp$;
10 Call $RetrieveThirdElement(iTemp)$ or $RetrieveSecondElement(iTemp)$;
11 Call $RowReuse(iTemp, i$,*sub-filter*$, sum)$;
12 Write completed element of sum into O;
13 **else**
14 Divide columns of the last sub-block into multiple partitions and try to evenly assign those partitions to threads of a warp. Each thread uses a direct method to calculate elements of O.;
15 The same method is adopted when processing the edge elements of O;

To apply our approach to depthwise convolution that works on a 2D matrix, we first divide the output into sub-blocks. Each sub-block contains exactly n columns (in this work, $n = 32$, which is the default warp size of our GPU platform). The only exception is the last sub-block, which may contain less than n columns. If a sub-block contains more than k rows ($k = 56$ in this work), we then further break down the sub-block along the height dimension. The blocking method implies that our approach can handle arbitrary input sizes. Each GPU thread block will process one or multiple sub-blocks, and each warp will compute one sub-block.

3.3.1 *Example*

Fig. 6 shows the mapping process of GPU threads to output elements. In this example, we slide a 3×3 filter over an 8×8 input. To apply a square filter at the edge of the image, we need to pad the input. To reduce the memory pressure, we do not allocate GPU memory space for the padded elements. Instead, we use different methods to calculate the edge and inner elements of the output. The edge and inner elements are represented by the shaded and dashed squares in Fig. 6, respectively.

In this example, we assume each GPU warp contains four threads. Therefore, we will divide the inner elements into multiple sub-blocks and each sub-block contains four columns so that a column can be processed by one of the four GPU threads within a wrap. In our case, we will have two sub-blocks, where sub-block 0 contains four columns, but sub-block 1 only contains two columns. To utilize the threads within a warp, we divide elements of the last two columns evenly among the four threads.

3.3.2 *Generalization*

In Algorithm 4, we describe our generalized solution. Here, we process the sub-blocks with exactly 32 columns (i.e., the

default wrap size of our evaluation GPU) and the last sub-block in Lines 4-15 and 16-19, respectively. In this way, each GPU thread calculates one column of the output elements. This is done through several steps. First, each thread block loads the filter into shared memory and divides the filter into a combination of 3-column and 5-column sub-filters. Next, each thread calculates the address of the first input element it needs (Line 6). For each output element and sub-filter, each thread loads corresponding input elements into $iTemp$ and passes $iTemp$ to Algorithms 1 and 2 to fill the row vector $iTemp$ (Line 10). Then, each thread passes the filled vector $iTemp$ to Algorithm 3 to calculate multiple output elements and store results in the register array sum (Line 11). Finally, when the calculation of one output element is completed, we write the corresponding result in sum into the result array O (Line 13).

4 Optimizing Pointwise Convolution

In this section, we explain the workflow of our dynamic tile size scheme for pointwise convolution. This approach extends the optimization for convolution operations in our prior work [28] to pointwise convolution. Our approach consists of three stages, described as follows.

In the first and second stages, we identify parameters related to the tile size and determine candidate values for each parameter (Section 4.1 and Section 4.2). The first and second stages process input dependent and independent parameters respectively. In the third stage, as detailed in Algorithm 5, we iterate over all combinations of parameters and search for the combination that achieves optimal SM utilization and data reuse (Section 4.3).

We note that previous studies [31], [32], [33], [34], [35], [36], [37], [38] have exploited tiling and autotuning for convolution and GEMM operations. However, these prior methods are inadequate for pointwise convolutions on GPUs due to two main drawbacks: they do not consider SM utilization when choosing the optimal tile size and are not designed for pointwise convolutions with small inputs. Our dynamic tile size scheme avoids these two drawbacks. To improve SM utilization, our approach searches for the optimal tile size for the output based on the input size to generate a proper number of tiles to saturate GPU and maximize data reuse. To optimize pointwise convolution with small inputs, we distribute channels across threads within a warp to increase the arithmetic intensity for each thread.

4.1 Determine Tiling Parameters

In our design, we use a 2-level tiling scheme, as shown in Fig. 7, to partition the output into block tiles and warp tiles. Each thread block processes one block tile and each warp processes one warp tile. The height dimension of the warp tile is shared among 32 threads of a warp and the width dimension of the warp tile is distributed across 32 threads of a warp. Hence, we have two input dependent parameters, namely the height and width of the warp tile, denoted as $Warp_H$ and $Warp_W$ respectively. Now we introduce how to use the 2-level tiling scheme to determine candidate values for $Warp_H$ and $Warp_W$.

4.1.1 A two-level tiling scheme

To divide the output into block tiles, we utilize two logical layouts of the output, ***L1*** and ***L2***, as shown in Fig. 7. F_N and $I_N \times I_H \times I_W$ represent the filter and input dimensions of the output respectively. Notice that our 2-level tiling can handle arbitrary input sizes since we do not require $I_H = I_W$. Before partitioning the output, we first select the layout of the output based on the size of the filter dimension. The rationale behind choosing the filter dimension instead of the input dimension can be described as follows. The number of filters, F_N, is fixed once the structure of a CNN is determined. But the size of the input dimension will be affected by the batch size, I_N, during inference and training. Therefore, it is easier to design our approach based on the size of the filter dimension. When $F_N > 48$, we choose layout ***L1*** and distribute filter channels across threads within a warp. Otherwise, we choose layout ***L2*** and distribute input channels. The boundary $F_N = 48$ is determined as follows. Fig. 7 demonstrates that in layout ***L2***, the maximal value of F_N is $4 \times Warp_H$ and $Warp_H \leq 12$ (explain later in this section), therefore we have $F_N \leq 48$ for layout ***L2***.

Since both layouts have the similar procedure, we take layout ***L1*** as an illustration example and give a brief description of layout ***L2*** at the end of this section. After choosing the layout based on F_N, we partition the output along the filter dimension. First, we halve the filter dimension if $F_N \geq 512$. The reason is that if we let each thread block process a large number of filters, then each thread needs to issue more than 15 global memory load instructions, which may cause MIO (Memory Input Output) instruction queue throttle and leads to performance degradation. Then, we halve both dimensions of each block tile and generate 2×2 warp tiles.

4.1.2 Determine candidates for $Warp_H$ and $Warp_W$

Based on the partition method, we know that $Warp_W$ can be calculated with $Warp_W = F_N/4$ or $Warp_W = F_N/2$. Thus, we only need to determine candidate values for $Warp_H$ based on the size of the input dimension. In our design, when $Warp_H > 12$, we need assembly level optimizations like the work in [16], [39] for some configurations of pointwise convolutions to avoid register spills. But in this work, we focus on higher level rather than assembly level optimizations, and thus set $Warp_H \leq 12$. If the size of the input dimension is large, we prefer to choose a large $Warp_H$ because using small $Warp_H$ will generate many thread blocks and results in multiple loads of shared filters [40], [41]. If the size of the input dimension is small, we prefer to choose a small $Warp_H$ because using a large $Warp_H$ will generate a few thread blocks and result in SM underutilization. Since each thread loads at most 12 input elements ($Warp_H \leq 12$), we set the upper limit of large $Warp_H$ to 12 and the lower limit to $12/2 = 6$. Therefore, the candidates for large $Warp_H$ are $Warp_H = \{6, 7, 8, 9, 10, 11, 12\}$. The candidates for small $Warp_H$ are $Warp_H = \{2, 3, 4, 5, 6, 7, 8\}$. In our experiments, there is no clear boundary between large and small candidate sets of $Warp_H$, therefore we let both sets overlap in the middle values. The boundary between the large and small size of the input dimension is experimentally determined as $I_N \times I_H \times I_W = 16 \times 14 \times 14$.

Compared to layout ***L1***, layout ***L2*** swaps the input and filter dimensions. Hence, $Warp_H$ can be calculated with $Warp_H = F_N/4$ or $Warp_H = F_N/2$. The candidate values for large $Warp_W$ are $Warp_W = \{6, 7, 8, 9, 10, 11, 12\}$ and for small $Warp_W$ are $Warp_W = \{2, 3, 4, 5, 6, 7, 8\}$.

4.2 Determine Candidates for Input Independent Parameters

There are three input independent parameters we need to consider, namely the number of warps in a thread block ($Warp_{num}$), the number of thread blocks that can run concurrently on an SM ($Block_{num}$) and the number of channels to be distributed (C_{num}).

4.2.1 Determine candidates for $Warp_{num}$ and $Block_{num}$

When determining candidates for $Warp_{num}$, we need to consider (1) a small warp number will decrease the opportunity to hide the memory access latency at the warp level, (2) a large warp number will decrease the number of thread blocks and may lead to SM underutilization. We empirically set the warp number to be four ($Warp_{num} = 4$), which gives good performance on our pilot study using microbenchmarks of hand-written pointwise convolution kernels. For the number of thread blocks, $Block_{num}$, we use two values, 2 and 4, on our evaluation platforms. These choices are justified as follows. For Nvidia GPUs, each GPU thread can use up to 255 registers, and each SM has 65,536 registers. If we set $Block_{num} = 1$ and $Warp_{num} = 4$ (per our discussion above), each SM will have $Block_{num} \times Warp_{num} = 4$ wraps. This allows a thread block to use up to just half of the available registers of an SM because a thread block under this setting can use at most $4\ (warps\ in\ an\ SM) \times 32\ (threads\ per\ warp) \times 255\ (registers\ per\ thread) = 32,640$ registers. Therefore, to utilize the available hardware register, one should set $Block_{num}$ to be greater than one. We also found that setting $Block_{num} > 4$ during searching offers little benefit and hence we set the $Block_{num}$ to be either 2 or 4 ($Block_{num} = \{2, 4\}$).

4.2.2 Determine candidates for C_{num}

When searching for the optimal combination of parameters, a small tile size may be generated, which may lead to low arithmetic intensity and can not hide global memory access latency. For example, we assume that the warp tile size is $Warp_H \times Warp_W = 8 \times 64$ and has 56 channels, which means that one warp needs to convolve 8 input elements with 64 filter elements and accumulates results 56 times to generate $8 \times 64 = 512$ elements. Since the height dimension is shared among 32 threads of the warp, each thread loads 8 input elements, and the width dimension is distributed across 32 threads, each thread loads 2 filter elements. Therefore, each thread accumulates 56 channels of $8 \times 2 = 16$ elements. Now we can estimate the arithmetic intensity of each thread for one iteration as $\frac{number\ of\ multiplications}{number\ of\ elements} = \frac{8\times2}{8+2} = 1.6$. We can improve arithmetic intensity by distributing channels across threads, as shown in Fig. 7. We distribute eight channels ($C_{num} = 8$) of each filter element across 32 threads of the warp. In that case, each warp can process $F_{num} = 32/C_{num} = 32/8 = 4$ filter elements and each thread processes $T_{num} = Warp_W/F_{num} = 64/4 = 16$ filter elements. The arithmetic intensity can be estimated as $\frac{Warp_H \times T_{num}}{Warp_H + T_{num}} = \frac{8\times16}{8+16} = 5.3$. Higher arithmetic intensity increases the chance to hide global memory access latency. To fully utilize a warp, candidate values for C_{num} should be a power of 2. Thus, candidates for C_{num} are $C_{num} = \{1, 2, 4, 8, 16, 32\}$.

4.3 Search For the Optimal Combination

4.3.1 Hardware resources constraints

When searching for the optimal combination of tiling and input independent parameters, we focus on combinations that can meet the hardware resources constraints, including registers and shared memory. In the rest of this section, we take layout ***L1*** as an illustration example. Based on $Block_{num}$, we calculate the number of registers each thread can use ($Limit_R$) and the size of shared memory each thread block can use ($Limit_S$) with formulas $Limit_R = Total_R/(Block_{num} \times Warp_{num} \times 32)$ and $Limit_S = Total_S/Block_{num}$ respectively. $Total_R$ and $Total_S$ represent the number of registers and the size of shared memory of an SM, respectively. On RTX 2080Ti, $Total_R = 65536$ and $Total_S = 64KB$ while on Jetson AGX Xavier, $Total_R = 65536$ and $Total_S = 48KB$.

In our approach, each warp processes one warp tile which contains $Warp_H \times Warp_W$ output elements. Each thread calculates $Warp_H \times T_{num}$ elements and thus needs $R_{result} = Warp_H \times T_{num}$, $R_{operand} = Warp_H + T_{num}$ registers to store results and operands respectively. The constraints can be formulated as follows:

$$R_{tmp} = \frac{C_{num} \times 2 \times Warp_W}{128} + \frac{C_{num} \times 2 \times Warp_H}{128}$$

$$R_{result} + R_{operand} + R_{tmp} + extraR \leq Limit_R \quad (1)$$

$$(2 \times Warp_H + 2 \times Warp_W) \times C_{num} \times 4 \times 2 \leq Limit_S \quad (2)$$

where R_{tmp} is the number of temporary registers used to store filter and input elements loaded from global memory. $2 \times Warp_H$ and $2 \times Warp_W$ represent the height and width of the block tile respectively. 128 means each thread block has $Warp_{num} \times 32 = 4 \times 32 = 128$ threads to load data from global memory. In Formula 1, $extraR$ is the number of additional registers allocated by the compiler and its value is determined through an off-line method. In our experiments, we set $extraR = 40$ because the NVIDIA CUDA compiler, on average, allocates 40 additional registers for each kernel on our evaluation platforms. These additional registers are usually used to store temporary variables for utilizing GPU arithmetic pipelines. In Formula 2, 4 means each element has 4 bytes and 2 means we use a double buffer method [33], [42].

4.3.2 Searching workflow

To guide the search for the optimal combination of parameters, we use two metrics named SM utilization (SM_{util}) and arithmetic intensity (AI). Two metrics can be calculated as follows:

$$Block_{count} = \frac{F_N}{2 \times Warp_W} \times \frac{I_N \times I_H \times I_W}{2 \times Warp_H}$$

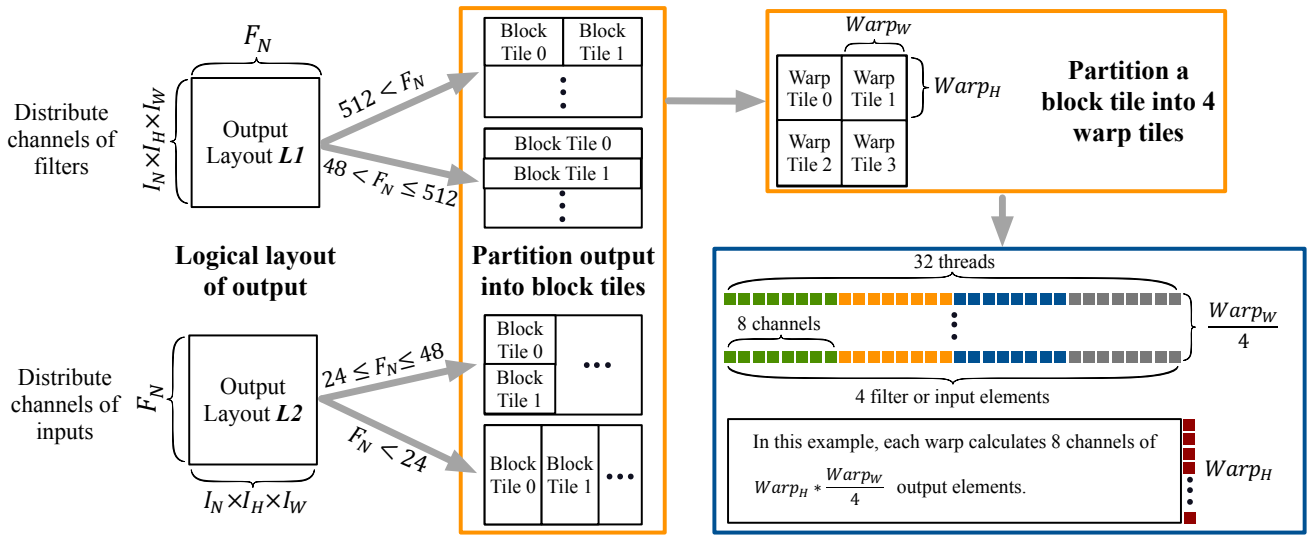


Fig. 7. Workflow of our 2-level tiling and channel distribution methods.

Algorithm 5: Optimized Pointwise Convolution

Input: I, F
Output: O
// below codes are executed on CPU
1 Determine candidates for relevant parameters;
2 **foreach** *parameter combination* **do**
3 **if** *not satisfy constraints of Formula 1 and 2* **then**
4 continue;
5 Calculate SM_{util} and AI with Formula 3 and 4;
6 Choose combinations whose SM_{util} is close to 1;
7 Among chosen combinations, choose the combination with the maximal AI;
8 Choose the kernel based on the chosen combination.;
// below codes are executed on GPU
9 Load C_{num} channels of a block tile into shared memory array $sharedBuf1$;
10 $\_\_syncthreads()$;
11 **for** $iter \leftarrow 0$ **to** I_C *By* $2 \times C_{num}$ **do**
12 Load next C_{num} channels into R_{tmp};
13 Load channels from $sharedBuf1$ into $R_{operand}$;
14 Accumulate output elements into R_{result};
15 Write R_{tmp} into $sharedBuf2$;
16 $\_\_syncthreads()$;
17 Repeat above steps but swap $sharedBuf1$ and $sharedBuf2$;
18 Use segmented parallel reduction to get the final output elements and write the result to O;

$$SM_{util} = \frac{Block_{count}}{Block_{num} \times SM_{num}} \tag{3}$$

$$AI = \frac{Warp_H \times T_{num}}{Warp_H + T_{num}} \tag{4}$$

where $Block_{count}$ is the number of generated thread blocks, SM_{num} is the number of SMs on a GPU. For RTX 2080Ti and Jetson AGX Xavier, $SM_{num} = 68$ and $SM_{num} = 8$ respectively.

The whole workflow is described in Algorithm 5. We first determine candidates for relevant parameters, including $Warp_H$, $Warp_W$, $Warp_{num}$, $Block_{num}$ and C_{num}, based on the size of the input and filter (Line 1). Then we iterate over all combinations of parameters (Line 2), and keep the combinations that satisfy the constraints $Limit_R$ (Formula 1) and $Limit_S$ (Formula 2) (Line 3).

Next, we calculate values of SM_{util} (Formula 3) and AI (Formula 4) for all satisfied combinations (Line 5) and select the optimal combination with following steps (Line 6-7):

Step 1 If $SM_{util} \geq 1$ is true for all combinations, we select the combinations that possess the smallest and close to the smallest SM_{util}. The reason is that when $SM_{util} \geq 1$, all SMs are utilized, in which case we want to reduce the number of thread blocks to reduce the number of loads of shared filters or inputs between multiple thread blocks.

Step 2 If there exists combinations such that $SM_{util} < 1$, we first collect these combinations. Then, among collected combinations, we select the ones that possess the biggest and close to the biggest SM_{util}. The reason is that when $SM_{util} < 1$, there are idle SMs, in which case we want to increase SM_{util} to fully utilize SMs. We do not want SM_{util} to exceed 1 because that will incur more memory operations.

Step 3 Among candidate combinations selected in Step 1 and Step 2, we select the combination with the maximum value of AI because higher arithmetic intensity can hide more global memory access latency.

Last, we choose the pointwise convolution kernel based on the selected combination (Line 8). In this kernel, each thread block first loads C_{num} channels of the corresponding block tile into shared memory array $sharedBuf1$(Line 9). Meanwhile, the thread block loads the next C_{num} channels of the block tile into temporary registers (Line 12). Then, we load data from $sharedBuf1$ into registers (Line 13) and accumulate output elements into registers (Line 14). Next, we write data in temporary registers into $sharedBuf2$ (Line 15). The kernel repeats the process until all channels have been accumulated to output elements. Finally, we use a warp level segmented parallel reduction to reduce results of different channels into the final result and write results to global memory (Line 18).

5 Experimental Setup

5.1 Evaluation Platforms

We apply our approach to two GPU platforms. The first platform has an NVIDIA RTX 2080Ti GPU (2080Ti), which integrates 4350 CUDA cores for floating point computation and 4350 CUDA cores for integer operations. The GPU has 64KB of shared memory. The host machine has a 2.30GHz Intel Xeon E5-2697 CPU with 252GB memory, running Linux kernel v4.15.0. We use CUDA Toolkit 11.0 and cuDNN 7.6.5. The second platform is an embedded GPU platform. It has an NVIDIA Jetson AGX Xavier GPU (Xavier), which integrates 512 Volta cores and 48KB shared memory. The host machine has an 1.2GHz 8-core ARM CPU with 32GB memory, running Linux kernel v4.9.140-tegra. We use CUDA Toolkit 10.0 and cuDNN 7.6.3.

5.2 Competing Methods

We compare our approach against cuDNN [22] which supports a wide range of convolution operations, including depthwise and pointwise convolutions optimized for GPUs. Moreover, cuDNN can execute GEMM-, FFT- and Winograd-based convolutions, allowing us to compare our techniques with mainstream convolution methods. TensorFlow [43] is one of the mainstream machine learning frameworks. We also compare our approach against TensorFlow implementations of depthwise and pointwise convolutions.

5.3 Performance Report

We apply our approach to depthwise and pointwise convolutions of DSC. We run each test case ten times with batch sizes of 1, 8, 16, 32, 64 and 128 on an unloaded machine and report the averaged running time. We found little variance during execution runs, less than 2%. We run convolutions with two data types, 32-bit floating point (FP32) for normal CNNs and 8-bit integer (INT8) for quantized CNNs [44]. In our experiments, we utilize data layouts $NCHW$ and $NHWC$ for FP32 and INT8, respectively, where N, C, H, W respectively denote the batch size, the number of channels, the height and the width. CUDA [45] provides 8-bit integer 4-element vector dot product (DP4A) instruction that performs the vector dot product between two 4-element vectors and accumulates the result in a 32-bit integer. Utilizing the DP4A instruction, we can group four contiguous channels of the INT8 data type into a 4-element vector to perform convolution. Therefore, we utilize $NHWC$ data layout for the INT8 data type due to its better performance over $NCHW$.

In this work, we first test depthwise convolution with two filter sizes, 3×3 and 5×5, because these are commonly used filter sizes. Then, we report the performance of pointwise convolution. Lastly, we apply our optimized depthwise and pointwise convolutions on the standard and quantized MobileNetV2 and EfficientNet-B0 to report the performance of both inference and training.

6 Experimental Results

In this section, we report results for depthwise convolution (Section 6.1) and pointwise convolution (Section 6.2), as well as inference and training of MobileNetV2 and EffcientNet-B0 (Section 6.3), showing that our approach consistently outperforms alternative methods by delivering the overall best performance.

TABLE 1
Layer configurations of depthwise convolutions.

LAYER	I_N	I_C	$I_H \times I_W$	$F_H \times F_W$	S
CONV1	1,8,16,32,64,128	16	112×112	$3 \times 3, 5 \times 5$	2
CONV2	1,8,16,32,64,128	72	56×56	$3 \times 3, 5 \times 5$	2
CONV3	1,8,16,32,64,128	88	28×28	$3 \times 3, 5 \times 5$	1
CONV4	1,8,16,32,64,128	96	28×28	$3 \times 3, 5 \times 5$	2
CONV5	1,8,16,32,64,128	96	14×14	$3 \times 3, 5 \times 5$	1
CONV6	1,8,16,32,64,128	120	14×14	$3 \times 3, 5 \times 5$	1
CONV7	1,8,16,32,64,128	192	14×14	$3 \times 3, 5 \times 5$	1
CONV8	1,8,16,32,64,128	240	14×14	$3 \times 3, 5 \times 5$	2
CONV9	1,8,16,32,64,128	432	7×7	$3 \times 3, 5 \times 5$	1

TABLE 2
Average speedups of four depthwise convolution implementations with FP32 over GEMM.

	3×3, 2080Ti	3×3, Xavier	5×5, 2080Ti	5×5, Xavier
IMPLICIT	1.1	32.8	1.1	20.0
PRECOMP	1.1	1.2	1.0	1.4
ours	2.2	42.8	3.9	39.4
TensorFlow	1.8	34.6	2.2	25.3

6.1 Depthwise Convolution

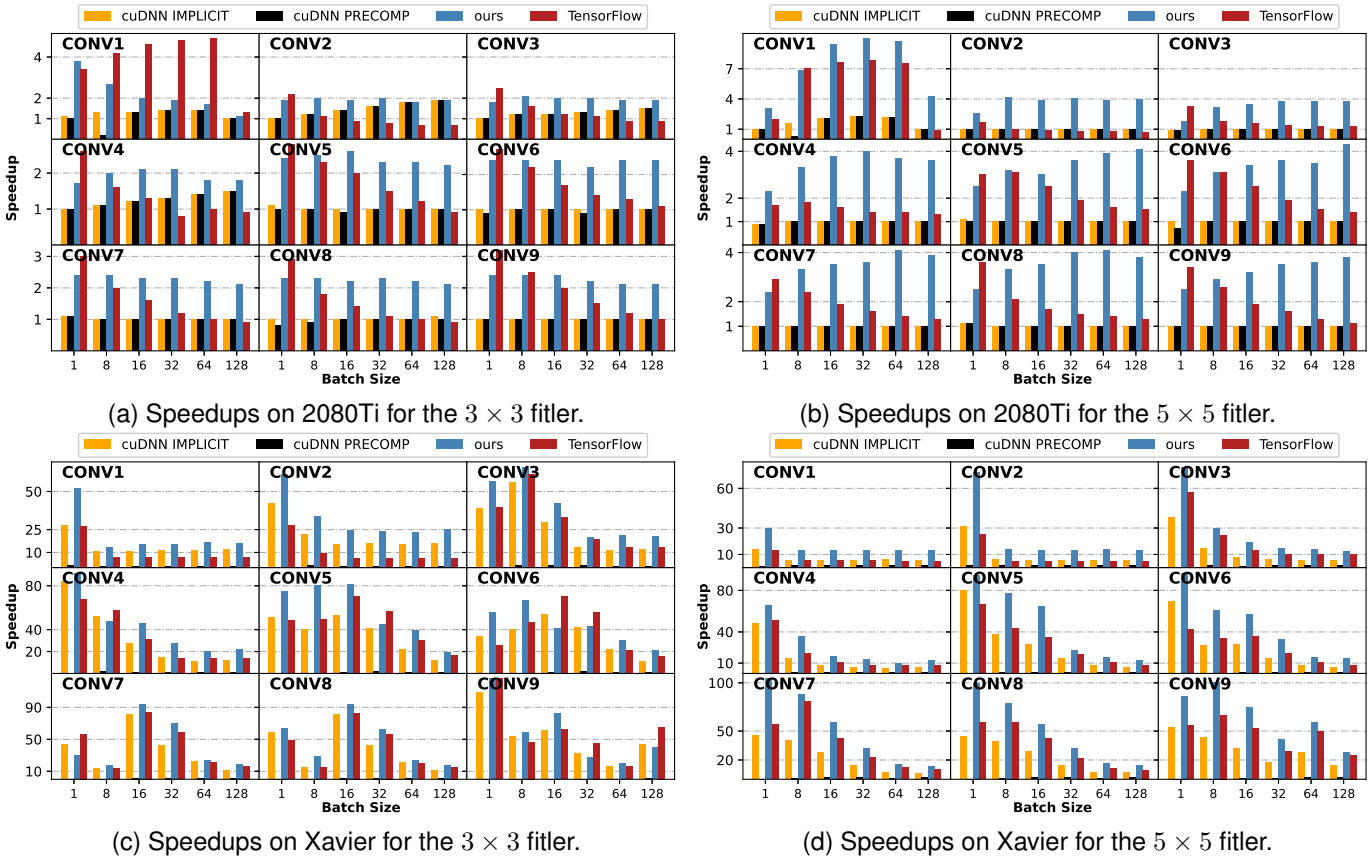
6.1.1 Setup

In this experiment, we compare our approach against the depthwise convolution implementations of cuDNN and TensorFlow. During the experiments, we have compared our approach to seven algorithms in cuDNN, including IMPLICIT_GEMM (IMPLICIT), IMPLICIT_PRECOMP_GEMM (PRECOMP), GEMM, FFT, FFT_TILING (TILING), WINOGRAD and WINOGRAD_NONFUSED (NONFUSED). We found IMPLICIT and PRECOMP give the best performance in all our test cases. We report the results by comparing our approach against IMPLICIT, PRECOMP and TensorFlow, and take GEMM as the baseline in this evaluation. Table 1 gives the layer configurations used in this experiment where the notations were defined earlier in Section 2.3.

6.1.2 Overall results

FP32 implementation. Fig. 8 shows that our approach gives the best speedup in nearly all test cases. Table 2 presents average speedups of IMPLICIT, PRECOMP, our approach and TensorFlow over GEMM for 3×3 and 5×5 filter sizes on 2080Ti and Xavier.

PRECOMP and GEMM algorithms need extra memory operations to compute output elements. Consequently, both algorithms are not suitable for depthwise convolution. TensorFlow achieves better speedups than IMPLICIT because it employs several specially designed kernels to increase the GPU utilization for different input sizes. However, both TensorFlow and IMPLICIT do not optimize memory performance for depthwise convolution. Compared to TensorFlow, our approach achieves an average speedup of $1.5\times$ and $1.6\times$ when using a 3×3 filter on 2080Ti and Xavier respectively, and $2.2\times$ and $1.7\times$ when using a 5×5 filter



(a) Speedups on 2080Ti for the 3×3 fitler.

(b) Speedups on 2080Ti for the 5×5 fitler.

(c) Speedups on Xavier for the 3×3 fitler.

(d) Speedups on Xavier for the 5×5 fitler.

Fig. 8. Speedups of IMPLICIT, PRECOMP, our approach and TensorFlow over the baseline implementation (GEMM) for FP32 depthwise convolution with filters of size 3×3 and 5×5 on two platforms.

on 2080Ti and Xavier respectively. Since IMPLICIT is closed source, we analyze its performance through CUDA Nsight Compute [46] and present the results in Section 6.1.3. Overall, our approach improves IMPLICIT by $2.0\times$ and $1.4\times$ when using a 3×3 filter on 2080Ti and Xavier respectively, and $3.5\times$ and $2.1\times$ when using a 5×5 filter on 2080Ti and Xavier respectively.

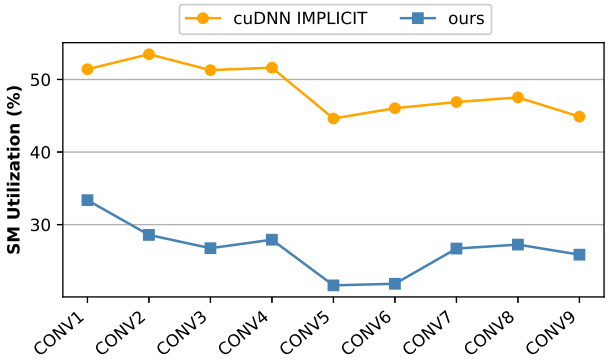
INT8 implementation. We found using FP32 gives a speedup of more than $10\times$ over the INT8 version for depthwise convolution in cuDNN. This is because the INT8 version has the overhead of dequantization (i.e., converting the results from INT8 to FP32 after convolution) and can not fully utilize DP4A instruction to accelerate INT8 convolution. We note that TensorFlow does not optimize depthwise convolution on INT8. Nonetheless, our approach gives over $10\times$ speedups when using INT8 over cuDNN and TensorFlow.

6.1.3 *Further analysis*

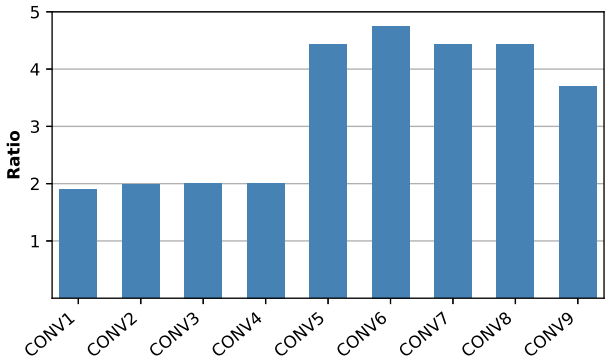
Our performance gain is mainly attributed to the reduced number of memory accesses offered by our column and row reuse algorithms.

Fig. 9 reports the measured LDG (load from global memory) instruction counts and SM utilization for the fast IMPLICIT algorithm and our approach when using a 3×3 filter and a batch size of 32 on 2080Ti. Other configurations follow a similar performance trend. We can see in Fig. 9a that the IMPLICIT algorithm has an average of $2\times$ higher SM utilization compared to our approach. The reason our approach leads to lower SM utilization is explained as follows. Our row reuse algorithm performs better when a thread operates on more rows of the output. However, the more rows a thread computes on, the fewer warps and thread blocks we can generate. Without enough warps running on SMs, the SM utilization will degrade. Though IMPLICIT has high SM utilization, it does not result in good performance for depthwise convolution. The reason is that depthwise convolution possesses a low computational requirement and is more sensitive to memory performance; hence the focus of performance optimization should be reducing the memory access latency. If we now look at Fig. 9b, we see that row and column reuse techniques reduce memory operations with up to $4.5\times$ lower LDG instructions to be executed when compared to IMPLICIT. By reducing the memory access overhead, which dominates the execution time of depthwise convolution, our approach thus can lead to better overall performance compared to cuDNN, despite lower SM utilization.

From Fig. 8 we can observe that speedups of our approach over IMPLICIT fluctuate in a small range as batch size increases. Both IMPLICIT and our approach can not benefit from higher GPU utilization because depthwise convolution is memory bound, thus IMPLICIT and our approach grow at the same rate.



(a) SM utilizations of IMPLICIT and our approach.



(b) The ratio of executed LDG (load from global memory) instruction counts given by IMPLICIT over our approach ($ratio = \frac{LDG\ inst\ counts\ of\ cuDNN}{LDG\ inst\ counts\ of\ ours}$).

Fig. 9. SM utilizations and ratios of executed LDG instruction counts for depthwise convolutions with a batch size of 32 and a filter size of 3×3 on the NVIDIA 2080Ti GPU.

6.1.4 Summary

By reducing the number of memory accesses, our approach leads to faster memory access time and overall quick computation time when performing depthwise convolutions. Compared to the fastest available algorithms in cuDNN, our approach achieves an average speedup of $2.8\times$ and $1.8\times$ when performing depthwise convolutions on 2080Ti and Xavier, respectively.

6.2 Pointwise Convolution

6.2.1 Setup

In this experiment, TensorFlow uses cuDNN implementations as their backend. Therefore, we only compare our approach against all available pointwise convolution implementations in cuDNN. The reported execution time of our approach includes the code running on both the CPU and the GPU, as described in Algorithm 5. We use the layer configurations from MobileNetV2 and EfficientNet-B0 in this experiment. Across different layers of the MobileNetV2 and EfficientNet-B0 models, there are 30 different configurations for pointwise convolution. We test all these configurations and report the performance of 20 selected layers. The other 10 layers exhibit similar performance as the selected ones and hence are omitted for clarity. We report the performance when batch sizes are set to 1, 8, 16, 32, 64 and 128.

To aid clarify, we compare our approach to the best-performing alternative scheme - IMPLICIT and PRECOMP for FP32 and PRECOMP for INT8. When using data type INT8, PRECOMP performs better than IMPLICIT in 180 out of 180 test cases on 2080Ti and 127 out of 180 test cases on Xavier. For FP32, we normalize the speedup over GEMM. For INT8, because GEMM does not support this data type, we show the speedup over IMPLICIT. Table 3 lists the layer configurations and parameter values generated for $Warp_H$, $Warp_W$, $Block_{num}$ and C_{num} ($Warp_{num} = 4$). The notations can be found at Section 2.3.

Normally, for a given convolution layer configuration, when the width of the logical layout of the output (Fig. 7) is small, our scheme tends to choose a small C_{num}. This allows one to generate more warps to utilize the GPU. On the other hand, our scheme tends to choose a large C_{num} to reduce the number of warps because there are already enough warps to maximize the utilization of the GPU. A special parameter tuple (we take parameter tuples generated for 2080Ti as examples) is the layer configuration CONV9 with $I_N = 1$. The width of the logical layout of CONV9 is small. Hence we would like to search for a small C_{num}. However, in this case, $C_{num} = 32$ is large. The reason is that our scheme finds that different values of C_{num} gives similar GPU utilization, then it tries to maximize AI (Formula 4) and then choose $C_{num} = 32$ (the relationship of AI and C_{num} is detailed in Section 4.2.2).

6.2.2 Overall results

FP32 implementation. Fig. 10 shows speedups of IMPLICIT, PRECOMP and our approach for pointwise convolutions on two platforms. The baseline is GEMM. Average speedups of IMPLICIT, PRECOMP and our approach on 2080Ti and Xavier are shown in Table 4. The performance delivered by our approach translates to an improvement of $2\times$ and $1.5\times$ over IMPLICIT on 2080Ti and Xavier respectively.

INT8 implementation. Fig. 11 shows speedups of PRECOMP and our approach over IMPLICIT for pointwise convolution. Table 4 presents average speedups of PRECOMP and our approach over IMPLICIT on 2080Ti and Xavier. Overall, our approach obtains $1.3\times$ and $1.5\times$ improvement over PRECOMP on 2080Ti and Xavier respectively.

6.2.3 Further analysis

Fig. 12 reports the measured SM utilizations and ratios of executed LDG instruction counts of IMPLICIT to our approach when using a batch size of 32 on 2080Ti. Other configurations have a similar performance.

For a specific layer configuration, our approach tries to find a suitable number of thread blocks to utilize the GPU. While using a higher number of thread blocks can improve the GPU utilization, doing so can also incur frequent reloads of filters or inputs shared between thread blocks. As shown in Fig. 12b, our approach leads to $2\times$ more LDG instructions than IMPLICIT in some cases. Although using more LDG instructions incurs extra memory load overhead, our approach still gives $2\times$ faster execution time over IMPLICIT due to our schemes for improving the SM utilization and hiding memory, elaborating as follows.

Our approach exhibits a much higher SM utilization than IMPLICIT. Unlike depthwise convolution, improving SM utilization is key for optimizing pointwise convolution

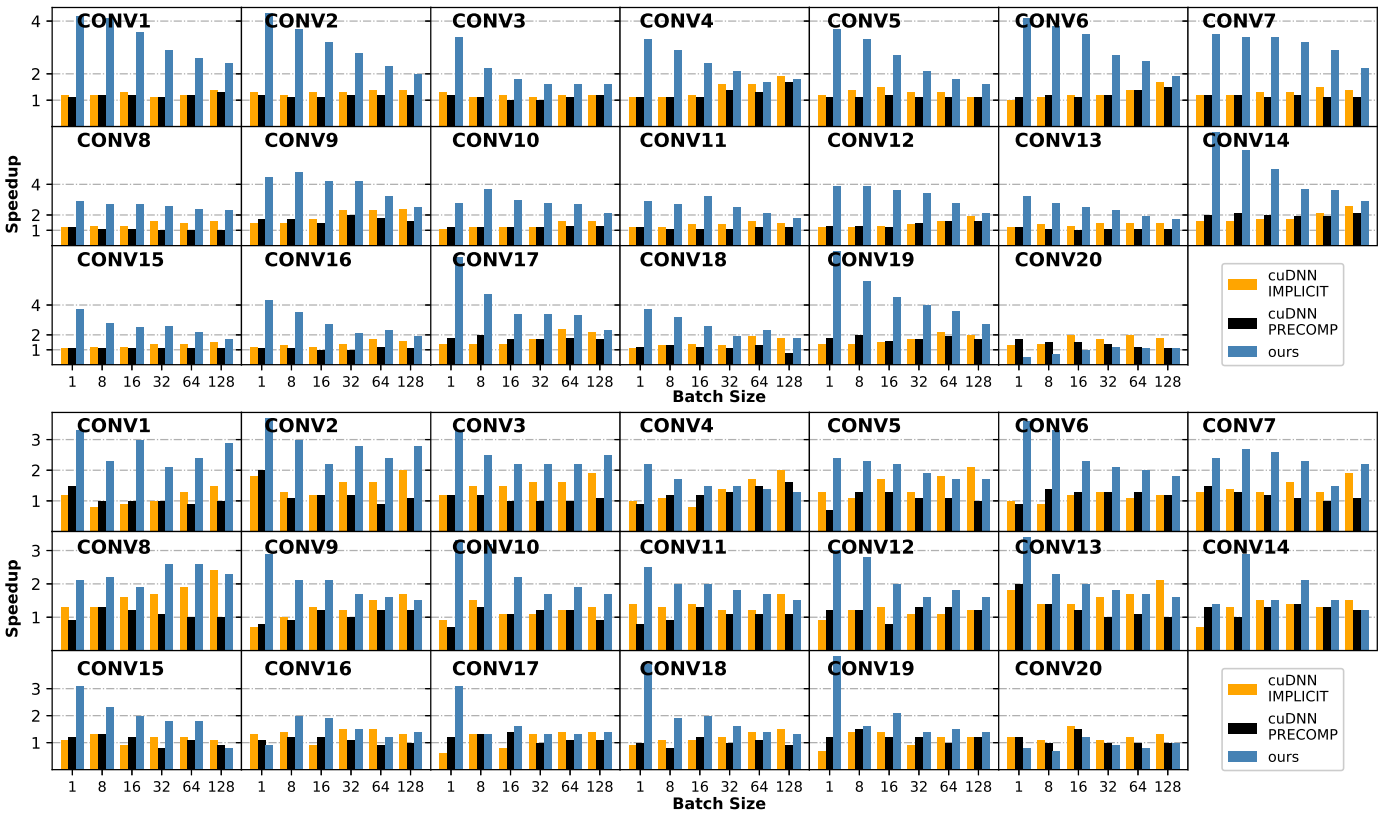


Fig. 10. Speedups of IMPLICIT, PRECOMP and ours over GEMM for pointwise convolutions with FP32 on 2080Ti (top) and Xavier (bottom).

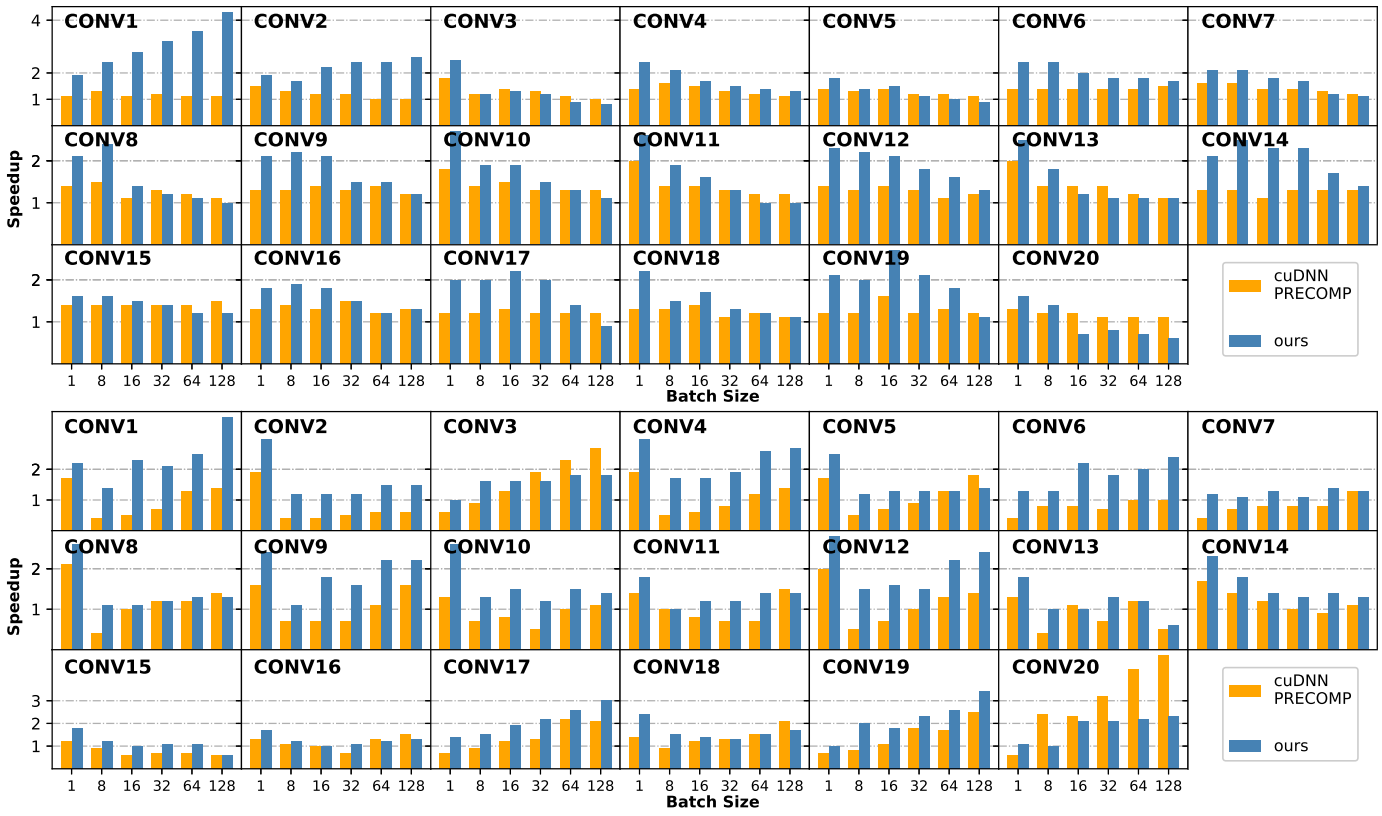


Fig. 11. Speedups of PRECOMP and ours over IMPLICIT for pointwise convolutions with INT8 on 2080Ti (top) and Xavier (bottom).

TABLE 3
Layer configurations of pointwise convolutions ($F_C = I_C$, $I_W = I_H$ and $F_W = F_H = 1$) and parameter values of $Warp_H$, $Warp_W$, $Block_{num}$ and C_{num}. We use the tuples ($Warp_H$, $Warp_W$, $Block_{num}$, C_{num}) and [$Warp_H$, $Warp_W$, $Block_{num}$, C_{num}] to represent parameter values generated for 2080Ti and Xavier respectively.

LAYER	I_C	I_H	F_N	$I_N = 1$	$I_N = 8$	$I_N = 16$	$I_N = 32$	$I_N = 64$	$I_N = 128$
CONV1	16	56	8	(4, 12, 2, 8) [4, 50, 4, 2]	(4, 47, 4, 2) [4, 480, 4, 1]	(4, 480, 4, 1) [4,1216, 2, 32]	(4, 480, 4, 1) [4,1216, 2, 32]	(4, 480, 4, 1) [4,1216, 2, 32]	(4,1216, 2, 32) [4,1216, 2, 32]
CONV2	8	56	16	(8, 12, 2, 8) [8, 50, 4, 4]	(8, 47, 4, 4) [8, 672, 2, 32]	(8, 256, 4, 1) [8, 672, 2, 32]	(8, 256, 4, 1) [8, 672, 2, 32]	(8, 672, 2, 32) [8, 672, 2, 32]	(8, 672, 2, 32) [8, 672, 2, 32]
CONV3	16	56	72	(12, 36, 2, 8) [12, 36, 4, 4]	(12, 36, 4, 4) [12, 36, 4, 4]	(12, 36, 4, 4) [12, 36, 4, 4]	(12, 36, 4, 4) [12, 36, 4, 4]	(12, 36, 4, 4) [12, 36, 4, 4]	(12, 36, 4, 4) [12, 36, 4, 4]
CONV4	72	28	24	(6, 6, 2, 32) [6, 50, 2, 4]	(6, 47, 2, 4) [6, 320, 4, 1]	(6, 47, 4, 4) [6, 864, 2, 32]	(6, 320, 4, 1) [6, 864, 2, 32]	(6, 320, 4, 1) [6, 864, 2, 32]	(6, 864, 2, 32) [6, 864, 2, 32]
CONV5	24	28	96	(3, 48, 2, 2) [12, 48, 4, 2]	(12, 48, 4, 2) [12, 48, 4, 2]	(12, 48, 4, 2) [12, 48, 4, 2]	(12, 48, 4, 2) [12, 48, 4, 2]	(12, 48, 4, 2) [12, 48, 4, 2]	(12, 48, 4, 2) [12, 48, 4, 2]
CONV6	96	14	24	(6, 2, 2, 32) [6, 13, 2, 16]	(6, 12, 2, 16) [6, 50, 4, 4]	(6, 24, 2, 8) [6, 320, 4, 1]	(6, 47, 2, 4) [6, 320, 4, 1]	(6, 47, 4, 4) [6, 864, 2, 32]	(6, 320, 4, 1) [6, 864, 2, 32]
CONV7	24	14	96	(2, 48, 2, 1) [7, 48, 2, 4]	(6, 48, 2, 4) [12, 48, 4, 2]	(12, 48, 2, 8) [12, 48, 4, 2]	(12, 48, 4, 2) [12, 48, 4, 2]	(12, 48, 4, 2) [12, 48, 4, 2]	(12, 48, 4, 2) [12, 48, 4, 2]
CONV8	32	14	192	(2, 96, 2, 1) [7, 96, 2, 2]	(6, 96, 2, 2) [12, 96, 4, 1]	(12, 96, 2, 4) [12, 96, 4, 1]	(12, 96, 4, 1) [12, 96, 4, 1]	(12, 96, 4, 1) [12, 96, 4, 1]	(12, 96, 4, 1) [12, 96, 4, 1]
CONV9	192	14	48	(12, 2, 2, 32) [12, 13, 2, 32]	(12, 12, 2, 32) [12, 50, 4, 2]	(12, 24, 2, 16) [12, 160, 4, 1]	(12, 47, 2, 8) [12, 480, 2, 32]	(12, 47, 4, 2) [12, 480, 2, 32]	(12, 160, 4, 1) [12, 480, 2, 32]
CONV10	96	14	40	(10, 2, 2, 32) [10, 13, 2, 16]	(10, 12, 2, 32) [10, 50, 4, 2]	(10, 24, 2, 16) [10, 192, 4, 1]	(10, 47, 2, 8) [10, 544, 2, 32]	(10, 47, 4, 4) [10, 544, 2, 32]	(10, 192, 4, 1) [10, 544, 2, 32]
CONV11	40	14	120	(2, 60, 2, 1) [7, 60, 2, 4]	(6, 60, 2, 2) [12, 60, 4, 2]	(12, 60, 2, 4) [12, 60, 4, 2]	(12, 60, 4, 2) [12, 60, 4, 2]	(12, 60, 4, 2) [12, 60, 4, 2]	(12, 60, 4, 2) [12, 60, 4, 2]
CONV12	120	14	32	(8, 2, 2, 32) [8, 13, 2, 16]	(8, 12, 2, 16) [8, 50, 4, 4]	(8, 24, 2, 8) [8, 256, 4, 1]	(8, 47, 2, 4) [8, 256, 4, 1]	(8, 47, 4, 4) [8, 672, 2, 32]	(8, 256, 4, 1) [8, 672, 2, 32]
CONV13	40	14	240	(2, 120, 2, 1) [7, 120, 2, 2]	(6, 120, 2, 1) [12, 120, 4, 1]	(12, 120, 2, 4) [12, 120, 4, 1]	(12, 120, 4, 1) [12, 120, 4, 1]	(12, 120, 4, 1) [12, 120, 4, 1]	(12, 120, 4, 1) [12, 120, 4, 1]
CONV14	240	7	64	(2, 32, 2, 2) [2, 32, 2, 2]	(2, 32, 2, 2) [7, 32, 4, 8]	(3, 32, 2, 2) [12, 32, 4, 4]	(6, 32, 2, 4) [12, 32, 4, 4]	(12, 32, 2, 8) [12, 32, 4, 4]	(12, 32, 4, 4) [12, 32, 4, 4]
CONV15	64	7	240	(2, 120, 2, 1) [2, 120, 2, 1]	(2, 120, 2, 1) [7, 120, 4, 2]	(3, 120, 2, 1) [12, 120, 4, 1]	(6, 120, 2, 1) [12, 120, 4, 1]	(12, 120, 2, 4) [12, 120, 4, 1]	(12, 120, 4, 1) [12, 120, 4, 1]
CONV16	72	7	432	(2, 216, 2, 1) [2, 216, 2, 1]	(2, 216, 2, 1) [7, 216, 4, 1]	(3, 216, 2, 1) [9, 216, 4, 32]	(6, 216, 2, 1) [9, 216, 4, 32]	(12, 216, 2, 2) [9, 216, 4, 32]	(9, 216, 4, 32) [9, 216, 4, 32]
CONV17	432	7	112	(2, 56, 2, 1) [2, 56, 2, 1]	(2, 56, 2, 1) [7, 56, 4, 4]	(3, 56, 2, 1) [12, 56, 4, 2]	(6, 56, 2, 4) [12, 56, 4, 2]	(12, 56, 2, 8) [12, 56, 4, 2]	(12, 56, 4, 2) [12, 56, 4, 2]
CONV18	112	7	432	(2, 216, 2, 1) [2, 216, 2, 1]	(2, 216, 2, 1) [7, 216, 4, 1]	(3, 216, 2, 1) [9, 216, 4, 32]	(6, 216, 2, 1) [9, 216, 4, 32]	(12, 216, 2, 2) [9, 216, 4, 32]	(9, 216, 4, 32) [9, 216, 4, 32]
CONV19	432	7	72	(2, 36, 2, 1) [2, 36, 2, 1]	(2, 36, 2, 1) [7, 36, 4, 4]	(3, 36, 2, 2) [12, 36, 4, 4]	(6, 36, 2, 4) [12, 36, 4, 4]	(12, 36, 2, 8) [12, 36, 4, 4]	(12, 36, 4, 4) [12, 36, 4, 4]
CONV20	432	7	1024	(2, 256, 2, 1) [4, 256, 2, 1]	(3, 256, 2, 1) [8, 256, 4, 32]	(6, 256, 2, 1) [8, 256, 4, 32]	(12, 256, 2, 1) [8, 256, 4, 32]	(8, 256, 4, 32) [8, 256, 4, 32]	(8, 256, 4, 32) [8, 256, 4, 32]

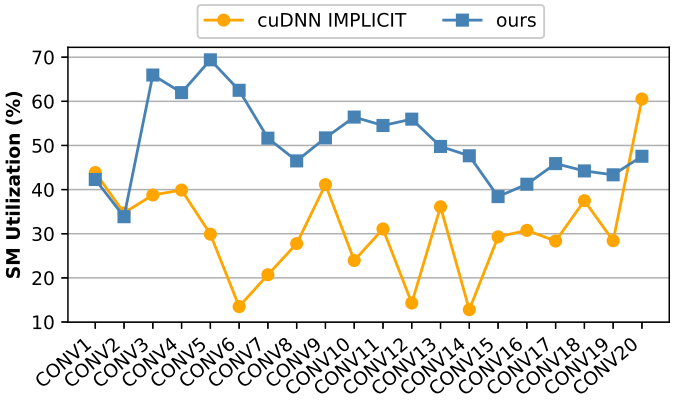
TABLE 4
Average speedups of three pointwise convolution implementations over GEMM and IMPLICIT for FP32 and INT8 respectively.

	FP32, 2080Ti	FP32, Xavier	INT8, 2080Ti	INT8, Xavier
IMPLICIT	1.5	1.3	1.0	1.0
PRECOMP	1.3	1.1	1.3	1.2
ours	3.0	2.0	1.7	1.6

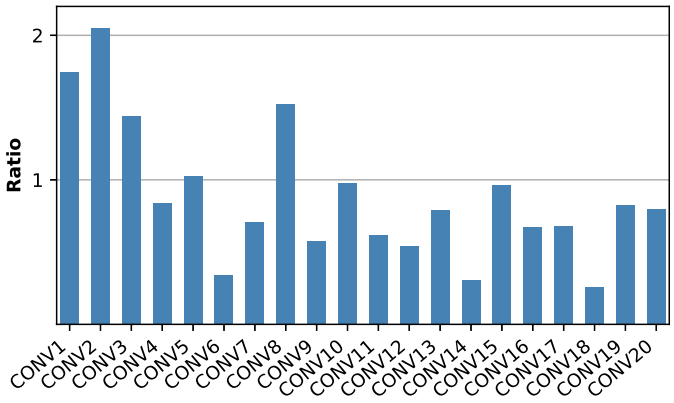
because utilizing more SMs can significantly accelerate the computation. As can be seen from Fig. 12a, our approach has an average of $1.9\times$ higher SM utilization compared to IMPLICIT. IMPLICIT is optimized for training and large batch-sized inference. It uses a fixed tile size work distribution strategy, which fails to utilize SMs efficiently when using a batch size of 128 or smaller. Our dynamic tile size scheme (Section 4.1) overcomes this limitation by adaptively determining the right tile size to use at runtime, which thus leads to better SM utilization and performance improvement.

To hide the global memory access latency, our approach employs double buffering and channel distribution techniques as described in Section 4.2. To quantify the benefit of our memory optimization strategies, consider now Fig. 13 that shows the average number of cycles each GPU warp spends on waiting for the GPU global memory access operation to complete. As a baseline, we implemented a simple pointwise without latency hiding, denoted as simple. We can see that our approach can significantly reduce the memory access latency compared to the simple implementation. Therefore, although our approach incurs a larger number of LDG instructions, much of the memory access overhead can be hidden by our memory optimization strategy.

Furthermore, we observe some performance degradation for pointwise convolutions with the INT8 data type. When performing pointwise convolutions with INT8, we use $NHWC$ data format, and four continuous INT8 channels can be viewed as an INT32 channel. Thus, the size of the channel dimension is reduced to one-fourth of the original size. For small channel sizes ($I_C \leq 96$), the corresponding reduced channel sizes restrict choices of the number of channels distributed, which leads to suboptimal performance compared to original channel sizes. This can be improved by having a better channel size allocation scheme for INT8. We leave this as our future work.



(a) SM utilizations of IMPLICIT and our approach.



(b) Ratios of executed LDG (load from global memory) instruction counts of IMPLICIT to our approach ($ratio = \frac{LDG\ inst\ counts\ of\ cuDNN}{LDG\ inst\ counts\ of\ ours}$).

Fig. 12. SM utilizations and ratios of executed LDG instruction counts for pointwise convolutions with a batch size of 32 on 2080Ti.

From Figs. 10 and 11, we see that our approach is more noticeable on small batch sizes. This is because that pointwise convolution is more sensitive to the GPU utilization. A larger batch size tends to use more warps, which alone can improve the GPU utilization and further improve the performance of cuDNN. For example, the speedups of our approach over cuDNN when $I_N = 128$ are much smaller than the speedups when $I_N < 128$. By contrast, when the batch size is smaller, the resulting warps alone is insufficient in utilizing the GPU where our dynamic tile size scheme can help.

6.2.4 Summary

Our approach uses a dynamic tile size method to improve SM utilization and double-buffering and channel distribution to hide memory access latency. With the help of both methods, we achieve an average speedup of $2\times$ and $1.5\times$ over IMPLICIT on 2080Ti and Xavier, respectively.

6.3 End to End Performance for Inference and Training

6.3.1 Setup

In this experiment, we apply our depthwise and pointwise convolutions to MobileNetV2 and EfficientNet-B0 and report the end-to-end performance of inference and training with ImageNet dataset [47].

Inference. For inference, we test standard and quantized MobileNetV2 and EfficientNet-B0 with batch sizes of 1, 8, 16, 32, 64 and 128 on both platforms and report the respective inference time. For quantization, the input and filter are converting from FP32 to INT8, and the results are converted back to FP32 as the model output. As cuDNN performs poorly for depthwise convolutions with INT8, we do not apply quantization to depthwise convolutions for fair comparisons.

Training. For training, we test MobileNetV2 and EfficientNet-B0 with batch sizes of 16, 32, 64 and 128 on 2080Ti and report the average training time of one training iteration, including the forward and the back-propagation phases.

Workload and performance report. We use the open-source MobileNetV2 and EfficientNet-B0 implemented using the Caffe framework, but we replace the implementations of batch normalization and depthwise convolution layers with the heavily optimized cuDNN implementations. The cuDNN implementation is denoted as **cuDNN** and our implementation is denoted as **Ours**. We report the percentage of performance improvement of our approach compared to cuDNN implementations, denoted as **Improved**.

6.3.2 Overall results

Table 5 reports the measured inference time. For MobileNetV2 with FP32, our approach improves the performance of inference by 12.2% and 13.5% on average compared to IMPLICIT on 2080Ti and Xavier, respectively. For MobileNetV2 with INT8, we obtain 8.5% and 11.7% improvements on average over PRECOMP on 2080Ti and Xavier, respectively. For EfficientNet-B0 with FP32, our approach improves IMPLICIT by 14.4% and 12.3% on average on 2080Ti and Xavier, respectively. For EfficientNet-B0 with INT8, we obtain 9.9% and 9.6% improvements on average over PRECOMP on 2080Ti and Xavier, respectively. Table 6 shows that our approach averagely reduces the training time of MobileNetV2 and EfficientNet-B0 by 9.7% and 7.3% compared to IMPLICIT on 2080Ti, respectively. The results show that our approach can significantly reduce both the model inference and training time by speeding up DSC operations.

7 Related Work

Numerous efforts have been dedicated to optimizing convolution operations. As previously mentioned, GEMM-, FFT- and Winograd-based convolutions are broadly adopted convolution algorithms.

GEMM-based convolution is the first attempt to optimize convolution. Chellapilla et al. [48] developed an unrolling convolution algorithm called the im2col convolution algorithm. Abdelfattah et al. [33] use a simple pruning strategy to search for the optimal tiling size. However, their method is inadequate for depthwise separable convolution because they ignore SM utilization and arithmetic intensity when searching for the optimal tiling size. Our approach avoids this problem by dynamically adjusting the tiling size.

A wide range of techniques on auto-tuning GEMM kernels have been proposed. Among these, the FFT- and

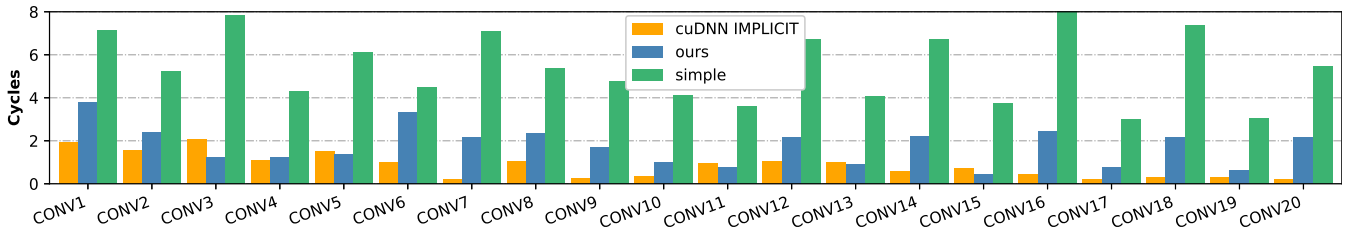


Fig. 13. The average number of cycles each warp spends on waiting for the global memory access to complete.

TABLE 5
Inference time of MobileNetV2 and EfficientNet-B0 with FP32 and INT8 on 2080Ti and Xavier.

		MobileNetV2						EfficientNet-B0					
	Batch	1	8	16	32	64	128	1	8	16	32	64	128
2080Ti (FP32)	**cuDNN (ms)**	7.5	8.8	9.7	14.4	19.1	28.7	10.1	13.7	18.1	25.0	36.4	52.3
	Ours (ms)	6.1	7.1	8.0	12.0	16.9	26.3	7.9	11.3	15.3	21.9	32.6	47.6
	Improved (%)	18.6	19.3	17.5	16.7	11.5	8.4	21.8	17.5	15.5	12.4	10.4	9.0
Xavier (FP32)	**cuDNN (ms)**	16.6	22.3	32.1	52.6	84.2	140.1	19.3	27.4	38.3	57.2	94.0	157.8
	Ours (ms)	13.2	18.9	27.8	44.7	76.1	130.0	15.5	23.2	32.1	50.7	87.3	151.1
	Improve (%)	20.5	15.2	13.4	15.0	9.6	7.2	19.7	15.3	16.3	11.4	7.1	4.2
2080Ti (INT8)	**cuDNN (ms)**	6.3	7.4	7.7	11.2	14.6	20.2	8.0	9.5	13.3	18.7	26.8	38.3
	Ours (ms)	5.5	6.6	6.8	10.3	14.0	19.7	6.8	8.2	11.8	16.9	25.3	36.6
	Improved (%)	12.7	10.8	11.7	8.0	4.1	2.5	15.0	13.7	11.3	9.6	5.6	4.4
Xavier (INT8)	**cuDNN (ms)**	13.3	18.0	27.0	42.6	64.8	103.7	16.1	21.0	33.7	52.8	80.3	127.5
	Ours (ms)	11.7	15.4	22.7	38.8	58.3	94.4	14.2	18.8	30.3	48.2	73.2	117.7
	Improved (%)	12.0	14.4	16.0	8.9	10.0	9.0	11.8	10.5	10.1	8.7	8.8	7.7

TABLE 6
Training time of MobileNetV2 and EfficientNet-B0 with FP32 on 2080Ti.

	MobileNetV2				EfficientNet-B0			
Batch	16	32	64	128	16	32	64	128
cuDNN (ms)	16.6	27.6	43,4	75.4	33.5	49.3	74.7	116.2
Ours (ms)	14.5	24.1	39.9	71.3	30.0	45.1	69.6	112.4
Improved (%)	12.7	12.7	8.1	5.4	10.4	8.5	6.8	3.3

Winograd-based convolutions are the dominating methods because they can reduce computational complexity and improve convolution performance. Mathieu et al. [49] proposed an FFT-based convolution to compute convolutions as pointwise products in the Fourier domain and reuse the transformed input data, which significantly reduces the complexity of the convolution. However, FFT-based convolution is more suitable for large filters than for small ones. Because padding the filters to the same size as the input data is necessary, and the latter (e.g., 3×3 filters) needs more memory than the former. Lavin et al. [50] used Winograd's minimal filtering algorithm to accelerate the convolution on GPU. This algorithm can reduce the arithmetic complexity of convolution by up to four times compared with direct convolution. However, Winograd's algorithm is only suitable for small filters due to its numerical instability. Zhen et al. [12] extended Winograd's algorithm to support any filter size. However, the traditional and extended Winograd-based algorithms need to transform the input and filter before performing matrix multiplication, and both require more operations than the FFT algorithm.

Transforming the input and filter before performing matrix multiplication incurs a large memory overhead, which can outweigh the performance gains obtained through lowering the computational complexity. Therefore, recent studies have looked into minimizing the memory overhead of the transformation phases. Cho et al. [51] reduced the memory overhead of GEMM-based convolutions using a compact lowering scheme to reduce the redundancy in the lowered matrix and then performed multiple small matrix multiplications in parallel. However, this algorithm still needs to transform the input and filter tensors into lowered matrixes to compute the convolution. Iandola et al. [52] reduced memory communication of 2D convolutions on GPU. They also prefetched the image regions to the registers. While their method uses fewer threads, each thread operates on a larger number of data items. As a result, their method does not reduce the number of global memory transactions. Unlike [52], our approach promotes register use and can significantly reduce the number of memory accesses.

The work presented in [53] splits larger batches into smaller batch sizes to mitigate computation resources restrictions of large batch size. Li et al. [54] explore the impact of data layout on convolution operations. Zhang et al. [55] design a method to map computation to FMA (fused multiply-add) units and focus on maximizing arithmetic intensity. Unlike our approach, none of these methods explicitly considers SM utilization, which is vital for achieving good performance for DSC on GPUs. The work presented in [24] also targets column and row reuse. However, there are two main drawbacks in their approach. First, 32 threads of a warp in their approach will generate 28 output elements for a 5×5 filter. Our column reuse method ensures that 32 threads of a warp generate 32 output elements for any filter sizes, thus our approach needs fewer warps and is more efficient than their approach. Second, their approach will load all needed input elements into registers to avoid reloading shared input elements, while our approach loads one input element each time and calculate all output elements that depend on the loaded input element. Thus, our approach uses less registers and can execute more warps on one SM concurrently.

8 Conclusion

We have presented two novel approaches to optimize memory performance and SM utilization for depthwise and pointwise convolutions respectively. Our approach improves the data locality for convolutional operations performed on the row and column directions to reduce the memory access. Our techniques utilize the common GPU shuffle operations supported by mainstream GPU programming models, including CUDA and OpenCL, and do not require hardware modifications. For pointwise convolution, the main problem is low SM utilization because cuDNN uses a fixed tile size for all pointwise convolutions. We design a dynamic tile size method and meanwhile hide the memory access latency. We evaluate our approach for FP32 and INT8 on NVIDIA RTX 2080Ti and Jetson AGX Xavier GPUs. We compare our approach against a wide range of heavily optimized convolution algorithms. Experimental results show that our approach consistently outperforms the competing methods by delivering the best overall performance for depthwise and pointwise convolutions.

Acknowledgments

This work was supported in part by the National Key Research and Development Program of China under grant agreement 2017YFB0202901, the Key-Area Research and Development Program of Guangdong Province under grant agreement 2019B010136001, the National Natural Science Foundation of China (NSFC) under grant agreements 61672186 and 61872294, and the Shenzhen Technology Research and Development Fund under grant agreement JCYJ20190806143418198. Professor Zhang is the corresponding author.

References

[1] D. Zoran, M. Chrzanowski, P.-S. Huang, S. Gowal, A. Mott, and P. Kohli, "Towards robust image classification using sequential attention models," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 9483–9492.

[2] C. Yang, Z. An, H. Zhu, X. Hu, K. Zhang, K. Xu, C. Li, and Y. Xu, "Gated convolutional networks with hybrid connectivity for image classification." in *AAAI*, 2020, pp. 12 581–12 588.

[3] S. Wang, Y. Gong, J. Xing, L. Huang, C. Huang, and W. Hu, "Rdsnet: A new deep architecture forreciprocal object detection and instance segmentation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 07, 2020, pp. 12 208–12 215.

[4] B. Chen, G. Ghiasi, H. Liu, T.-Y. Lin, D. Kalenichenko, H. Adam, and Q. V. Le, "Mnasfpn: Learning latency-aware pyramid architecture for object detection on mobile devices," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 13 607–13 616.

[5] Z. Zhong, Z. Q. Lin, R. Bidart, X. Hu, I. B. Daya, Z. Li, W.-S. Zheng, J. Li, and A. Wong, "Squeeze-and-attention networks for semantic segmentation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 13 065–13 074.

[6] H. Tokunaga, Y. Teramoto, A. Yoshizawa, and R. Bise, "Adaptive weighting multi-field-of-view cnn for semantic segmentation in pathology," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 12 597–12 606.

[7] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan *et al.*, "Searching for mobilenetv3," in *Proceedings of the IEEE International Conference on Computer Vision*, 2019, pp. 1314–1324.

[8] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International Conference on Machine Learning*. PMLR, 2019, pp. 6105–6114.

[9] D. Haase and M. Amthor, "Rethinking depthwise separable convolutions: How intra-kernel correlations lead to improved mobilenets," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 14 600–14 609.

[10] R. Zhang, F. Zhu, J. Liu, and G. Liu, "Depth-wise separable convolutions and multi-level pooling for an efficient spatial cnn-based steganalysis," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 1138–1150, 2019.

[11] L. Espeholt, R. Marinier, P. Stanczyk, K. Wang, and M. Michalski, "Seed rl: Scalable and efficient deep-rl with accelerated central inference," 2019.

[12] J. Zhen, A. Zlateski, F. Durand, and L. Kai, "Optimizing n-dimensional, winograd-based convolution for manycore cpus," in *Acm Sigplan Symposium on Principles & Practice of Parallel Programming*, 2018.

[13] Y. Liu, Y. Wang, R. Yu, M. Li, V. Sharma, and Y. Wang, "Optimizing cnn model inference on cpus," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 1025–1040.

[14] M. Winter, D. Mlakar, R. Zayer, H.-P. Seidel, and M. Steinberger, "Adaptive sparse matrix-matrix multiplication on the gpu," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019, pp. 68–81.

[15] Z. Li, H. Jia, Y. Zhang, T. Chen, L. Yuan, L. Cao, and X. Wang, "Autofft: a template-based fft codes auto-generation framework for arm and x86 cpus," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–15.

[16] D. Yan, W. Wang, and X. Chu, "Optimizing batched winograd convolution on gpus," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 32–44.

[17] A. Vasudevan, A. Anderson, and D. Gregg, "Parallel multi channel convolution using general matrix multiplication," in *IEEE International Conference on Application-specific Systems*, 2017.

[18] X. Li, Y. Liang, S. Yan, L. Jia, and Y. Li, "A coordinated tiling and batching framework for efficient gemm on gpus," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019, pp. 229–241.

[19] D. Wu, J. Li, R. Yin, H. Hsiao, Y. Kim, and J. San Miguel, "ugemm: unary computing architecture for gemm applications," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 377–390.

[20] A. Zlateski, Z. Jia, K. Li, and F. Durand, "The anatomy of efficient fft and winograd convolutions on modern cpus," in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 414–424.

[21] NVIDIA, *CUDA C++ Best Practices Guide*. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

[22] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *CoRR*, vol. abs/1410.0759, 2014.

[23] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun, "Fast convolutional nets with fbfft: A GPU performance evaluation," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[24] P. Chen, M. Wahib, S. Takizawa, R. Takano, and S. Matsuoka, "A versatile software systolic execution model for gpu memory-bound kernels," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–81.

[25] B. Pourghassemi, C. Zhang, J. H. Lee, and A. Chandramowlishwaran, "On the limits of parallelizing convolutional neural networks on gpus," in *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, 2020, pp. 567–569.

[26] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[27] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[28] G. Lu, W. Zhang, and Z. Wang, "Optimizing gpu memory transactions for convolution operations," in *2020 IEEE International*

Conference on Cluster Computing (CLUSTER). IEEE, 2020, pp. 399–403.

[29] X. Mei and X. Chu, "Dissecting gpu memory hierarchy through microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 72–86, 2016.

[30] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the nvidia volta gpu architecture via microbenchmarking," *arXiv preprint arXiv:1804.06826*, 2018.

[31] D. E. Tanner, "Tensile: Auto-tuning gemm gpu assembly for all problem sizes," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018, pp. 1066–1075.

[32] V. Kelefouras, A. Kritikakou, I. Mporas, and V. Kolonias, "A high-performance matrix–matrix multiplication methodology for cpu and gpu architectures," *The Journal of Supercomputing*, vol. 72, no. 3, pp. 804–844, 2016.

[33] A. Abdelfattah, S. Tomov, and J. Dongarra, "Fast batched matrix multiplication for small sizes using half-precision arithmetic on gpus," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 111–122.

[34] J. Kurzak, H. Anzt, M. Gates, and J. Dongarra, "Implementation and tuning of batched cholesky factorization and solve for nvidia gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 7, pp. 2036–2048, 2015.

[35] L. Jiang, C. Yang, and W. Ma, "Enabling highly efficient batched matrix multiplications on sw26010 many-core processor," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 1, pp. 1–23, 2020.

[36] P. Tillet and D. Cox, "Input-aware auto-tuning of compute-bound hpc kernels," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–12.

[37] H. Lan, J. Meng, C. Hundt, B. Schmidt, M. Deng, X. Wang, W. Liu, Y. Qiao, and S. Feng, "Feathercnn: Fast inference computation with tensorgemm on arm architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 3, pp. 580–594, 2019.

[38] Y. Zhang and F. Mueller, "Autogeneration and autotuning of 3d stencil codes on homogeneous and heterogeneous gpu clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 3, pp. 417–427, 2012.

[39] D. Yan, W. Wang, and X. Chu, "Demystifying tensor cores to optimize half-precision matrix multiply," in *2020 IEEE International Parallel and Distributed Processing Symposium, IPDPS*, 2020, pp. 20–24.

[40] L. Jia, Y. Liang, X. Li, L. Lu, and S. Yan, "Enabling efficient fast convolution algorithms on gpus via megakernels," *IEEE Transactions on Computers*, 2020.

[41] S. Zheng, Y. Liang, S. Wang, R. Chen, and K. Sheng, "Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 859–873.

[42] D. Nichols, N.-S. Tomov, F. Betancourt, S. Tomov, K. Wong, and J. Dongarra, "Magmadnn: towards high-performance data analytics and machine learning for data-driven scientific computing," in *International Conference on High Performance Computing*. Springer, 2019, pp. 490–503.

[43] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[44] M. Nagel, M. v. Baalen, T. Blankevoort, and M. Welling, "Data-free quantization through weight equalization and bias correction," in *Proceedings of the IEEE International Conference on Computer Vision*, 2019, pp. 1325–1334.

[45] NVIDIA, *CUDA Toolkit Programming Guide*. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[46] ——, *NVIDIA Nsight Compute*. [Online]. Available: https://developer.nvidia.com/nsight-compute

[47] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[48] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," *Tenth International Workshop on Frontiers in Handwriting Recognition*, 2006.

[49] M. Mathieu, M. Henaff, and Y. LeCun, "Fast training of convolutional networks through ffts," *arXiv preprint arXiv:1312.5851*, 2013.

[50] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4013–4021.

[51] M. Cho and D. Brand, "Mec: memory-efficient convolution for deep neural network," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pp. 815–824.

[52] F. N. Iandola, D. Sheffield, M. J. Anderson, P. M. Phothilimthana, and K. Keutzer, "Communication-minimizing 2d convolution in gpu registers," in *IEEE International Conference on Image Processing*, 2014.

[53] Y. Oyama, T. Ben-Nun, T. Hoefler, and S. Matsuoka, "Accelerating deep learning frameworks with micro-batches," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018, pp. 402–412.

[54] C. Li, Y. Yang, M. Feng, S. Chakradhar, and H. Zhou, "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*, 2016, pp. 633–644.

[55] J. Zhang, F. Franchetti, and T. M. Low, "High performance zero-memory overhead direct convolutions," in *International Conference on Machine Learning*, 2018, pp. 5776–5785.

Gangzhao Lu received the B.S. degree in computer science and engineering from Harbin Institute of Technology, China, in 2014. He is currently working toward the Ph.D. degree in the School of Cyberspace Science, Harbin Institute of Technology. His research interests include performance modeling, parallel optimization, auto-tuning.

Weizhe Zhang (Senior Member, IEEE) received B.Eng, M.Eng and Ph.D. degree of Engineering in computer science and technology in 1999, 2001 and 2006 respectively from Harbin Institute of Technology.

He is currently a professor in the School of Computer Science and Technology at Harbin Institute of Technology, China, and director in the Cyberspace Security Research Center, Peng Cheng Laboratory, Shenzhen, China. His research interests are primarily in parallel computing, distributed computing, cloud and grid computing, and computer network. He has published more than 100 academic papers in journals, books, and conference proceedings.

Zheng Wang is an associate professor with the University of Leeds. His research focuses on parallel computing, compilation and systems security.