Optimizing GPU Memory Transactions for Convolution Operations

Gangzhao Lu
Computer Science and Technology
Harbin Institute of Technology
China
lugangzhao@hit.edu.cn

Weizhe Zhang
Computer Science and Technology
Harbin Institute of Technology
China
wzzhang@hit.edu.cn

Zheng Wang
School of Computing
University of Leeds
United Kingdom
z.wang5@leeds.ac.uk

***Abstract*—Convolution computation is a common operation in deep neural networks (DNNs) and is often responsible for performance bottlenecks during training and inferencing. Existing approaches for accelerating convolution operations aim to reduce computational complexity. However, these strategies often increase the memory footprint with extra memory accesses, thereby leaving much room for performance improvement. This paper presents a novel approach to optimize memory access for convolution operations, specifically targeting GPU execution. Our approach leverages two optimization techniques to reduce the number of memory operations for convolution operations performed on the width and height dimensions. For convolution computations on the width dimension, we exploit shuffle instructions to exchange the overlapped columns of the input for reducing the number of memory transactions. For convolution operations on the height dimension, we multiply each overlapped row of the input with multiple rows of a filter to compute multiple output elements to improve the data locality of row elements.**

We apply our approach to 2D and multi-channel 2D convolutions on an NVIDIA 2080Ti GPU. For 2D convolution, our approach delivers over 2× faster performance than the state-of-the-art image processing libraries. For multi-channel 2D convolutions, we obtain up to 1.3× speedups over the quickest algorithm of cuDNN.

***Index Terms*—Performance Optimization, Convolution, Memory Optimization, GPUs**

I. Introduction

Convolution is a fundamental building block for many application tasks, including image and video processing and machine learning models. However, convolution operations are computation and memory intensive for representative image and machine learning processing tasks. Therefore, there is a critical need for accelerating convolution operations.

A wide range of techniques have been proposed to accelerate convolution operations [1], [2], [3], [4], [5], [6], [7], [8]. Among these methods, general matrix multiplication (GEMM) [6], [7], fast fourier transform (FFT) [2] and winograd [3] methods are the broadly adopted ones. However, these methods can incur many GPU global memory transactions (or memory accesses) during the transformation phase due to the involvement of matrix multiplications and duplicate elements of the transformed matrices.

In this work, we introduce two novel optimization techniques for operations performed on columns and rows to improve the memory performance of convolution operations. The first technique exploits column reuse by utilizing shuffle instructions (supported by both CUDA and OpenCL and hence is applicable to mainstream GPUs) to exchange elements among threads within the same GPU warp (or working group). In this way, we can avoid reloading the same elements shared among different threads. We further extend the shuffle instructions to facilitate dynamic indexing. The second technique targets row reuse by multiplying one input row with multiple rows of a convolutional kernel (or filter) to compute multiple output elements. 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.

We apply our optimization techniques to 2D and multi-channel 2D convolution operations and evaluate them on an NVIDIA 2080Ti GPU. We compare our approach against a range of highly optimized convolution libraries, including cuDNN [9]. Experimental results show that our approach delivers over 2× faster performance over the best-performing competitive strategy.

This paper makes the following technical contributions:

- It presents a novel algorithm for column reuse (Section II-A), which has a better generalization ability over prior work.
- It presents a novel row reuse algorithm to improve the data locality and reduce the number of global memory transactions when performing convolution in the row direction (Section II-B).
- It describes a novel method for transforming dynamic indices into static indices. Our approach enhances register promotion, leading to better performance (Section IV).

II. Our Approach

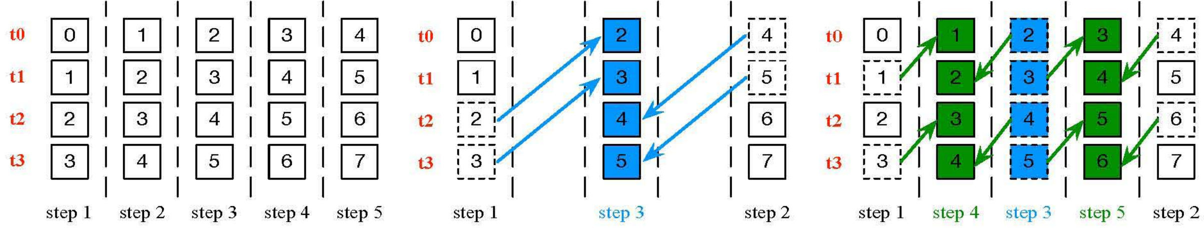
In this section, we describe our two optimizations, column reuse (Section II-A) and row reuse (Section II-B), for reducing GPU memory transactions for convolution operations.

A. Column Reuse Optimization

1) Standard convolution: Figure 1a shows a standard 2D convolution operation, operating on a single-channel input. Here, each thread loads the first corresponding input elements

978-1-7281-6677-3/20/$31.00 ©2020 IEEE
DOI 10.1109/CLUSTER49012.2020.00050

Authorized licensed use limited to: University of Leeds. Downloaded on January 06,2025 at 21:38:52 UTC from IEEE Xplore. Restrictions apply.



(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. 1. Illustration of direct and optimized convolution. 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.

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. After completing step 5, each pair of adjacent threads will have four duplicate input elements.

2) An optimized version: To eliminate the redundant loads, we could use the shuffle instructions to exchange input elements among different threads. Figure 1b depicts such an optimization. Specifically, in steps 1 and 2 of Figure 1b, 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.

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) Our approach: Our column reuse approach (Figure 1c) is described in 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. 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 the threads (Line 6).

Algorithm 1: RetrieveThirdElement

// $iTemp$: Buffer for storing input elements loaded from memory or generated through *shuffle* instructions.

Input: $iTemp$

Output: $iTemp$

1 $tid \leftarrow threadIdx.x;$

2 $mov\ exchange, \{iTemp[0], iTemp[4]\};$

3 $shift \leftarrow ((tid + 2)\&2) << 4;$

4 $exchange \leftarrow exchange >> shift;$

5 $mov\ \{iTemp[1], iTemp[2]\}, exchange;$

6 $iTemp[2] \leftarrow shfl_xor(iTemp[1], 2);$

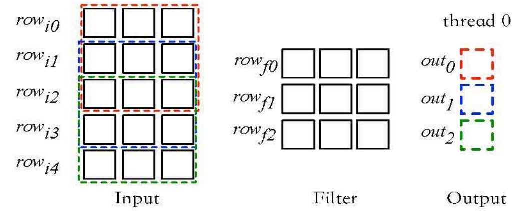


Fig. 2. A 3×3 filter is used to slide over the input image along height dimension, which produces a column of output elements.

B. Row Reuse Optimization

1) Standard convolution: Assume we use one thread to calculate one column of output elements. For the working example given in Figure 2, the convolution will be computed as follows:

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

The above equations suggest that row_{i1} and row_{i3} are loaded twice, and row_{i2} is loaded three times; nine rows should be loaded in total. The redundant loads to the same read-only row thus incur extra memory transactions and additional overhead.

2) Our optimization: To remove redundant loads to the same row, we redesign the execution flow of the convolution. Specifically, after loading a row from the input, we compute the number of output elements that depend on the loaded row. Our approach translates the execution flow of the working example presented in Figure 2 to:

Authorized licensed use limited to: University of Leeds. Downloaded on January 06,2025 at 21:38:52 UTC from IEEE Xplore. Restrictions apply.

Algorithm 2: RowReuse

```
   Input: row, index, filter, Out
   Output: Out
1  if index<F_H − 1 then
2  |  for i ← 0 to index + 1 do
3  |  |  Out[i] ← Out[i] + row · filter[index − i];
4  |  end
5  end
6  else if index ≥ F_H − 1 and index<I_H − F_H + 1 then
7  |  for i ← 0 to F_H do
8  |  |  o_index ← index − F_H + 1 + i;
9  |  |  Out[o_index] ← Out[o_index] + row · filter[F_H − 1 − i];
10 |  end
11 end
12 else
13 |  for i ← F_H − 1 to 0 do
14 |  |  o_index ← I_H − F_H + 1;
15 |  |  Out[o_index] ← Out[o_index] + row · filter[F_H − i];
16 |  end
17 end
```

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

In this new implementation, we would only issue loads to five rows to calculate the output elements. We note that although the number of accesses to the output column out is increased, the overhead is negligible because out is much smaller than multiple rows and hence can be stored in registers.

We describe a general solution for row reuse in Algorithm 2, 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 code at Lines 1-5 process the first $F_H - 1$ rows (row_{i0} and row_{i1} in Figure 2) that are needed by less than F_H output elements. Code at lines 6-11 process the rows needed by exact F_H output elements (e.g., row_{i2} in Figure 2). Finally, code 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 Figure 2).

III. Experimental Setup

We evaluate our approach on an NVIDIA RTX 2080Ti GPU, which integrates 4350 CUDA cores for floating point computation and 4350 CUDA cores for integer operations. We use CUDA Toolkit version 10.2.

We compare our approach against the following state-of-the-art image and convolution libraries: (1) cuDNN version 7.6.4. cuDNN is a state-of-the-art convolution implementation that supports 2D and multi-channel 2D convolutions on GPU. (2) ArrayFire [10], version 3.6.4. ArrayFire is a popular image and signal processing library. (3) NVIDIA Performance Primitives (NPP). This is an image and signal processing library. (4) GEMM-im2col. We extract the implementation of the GEMM-im2col from Caffe [11]. We apply our approach to 2D and multi-channel 2D convolutions.

TABLE I
Layer configurations used for multi-channel 2D convolutions†.

	I_N	$I_C = F_C$	$I_H \times I_W$	F_N	$F_H \times F_W$
CONV1	128	1,3	28×28	128	3×3
CONV2	128	1,3	56×56	64	3×3
CONV3	128	1,3	12×12	64	5×5
CONV4	128	1,3	14×14	16	5×5
CONV5	128	1,3	24×24	256	5×5
CONV6	128	1,3	24×24	64	5×5
CONV7	128	1,3	28×28	16	5×5
CONV8	128	1,3	28×28	512	3×3
CONV9	128	1,3	56×56	256	3×3
CONV10	128	1,3	112×112	128	3×3
CONV11	128	1,3	224×224	64	3×3

† We use I, F, and O to represent the input, the filter, and the output respectively, N, C, H, and W to denote the batch size, the channel, the height, and the width, respectively.

IV. Experimental Results

A. 2D Convolution

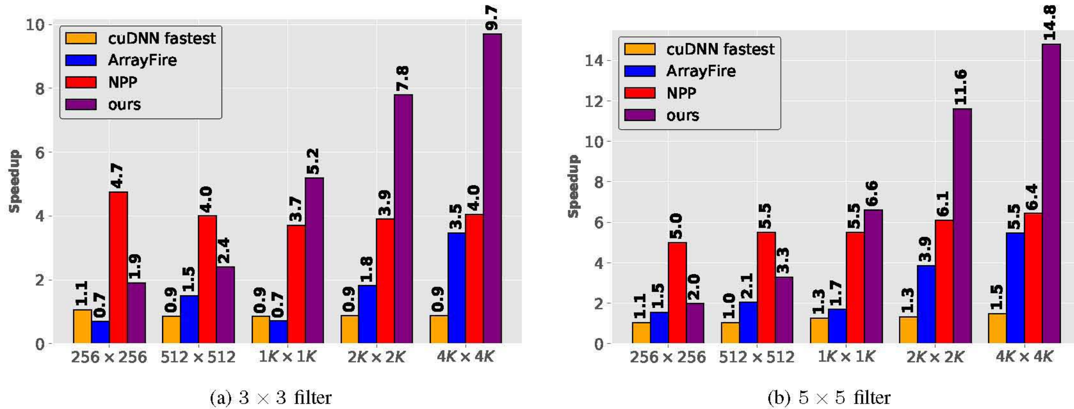
1) Setup: In this experiment, we compare our approach against the 2D convolution implementations from cuDNN, GEMM-im2col, ArrayFire, and NPP. As cuDNN provides multiple implementations, we empirically choose the fastest version, denoted as cuDNN-fastest, for evaluation. We apply each method to images with sizes ranging from 256×256 to $4K \times 4K$.

2) Overall results: Figure 3 reports the speedups of cuDNN, ArrayFire, NPP and our approach over GEMM-im2col. While cuDNN has been heavily optimized for NVIDIA GPUs, it does not show a notable performance advantage. When using a 3×3 filter, our approach gives the best overall speedup of 5.4× (up to 9.7× for the largest input), which translates to an improvement of more than 30% over the second-best method, NPP. We note that our approach is based on the standard 2D direct convolution by applying the column and row reuse algorithms. Therefore, the performance gain is mainly attributed to the reduction of the number of memory transactions. When using a 5×5 filter, our approach achieves a better overall speedup of 7.7×.

B. Multi-channel 2D Convolution

1) Setup: In this experiment, we compare our approach against the multi-channel 2D convolution implementations in cuDNN and use GEMM-im2col as the baseline. Since our work focuses on optimizing memory transactions of convolutions but not operations on input channels, we apply our approach to convolutions with one and three input channels, which are typically used in the first layer of a CNN. We use the

Authorized licensed use limited to: University of Leeds. Downloaded on January 06,2025 at 21:38:52 UTC from IEEE Xplore. Restrictions apply.



(a) 3×3 filter

(b) 5×5 filter

Fig. 3. Speedups of 2D convolutions of four implementations over GEMM-im2col when using a 3×3 (a) and a 5×5 filter (b).

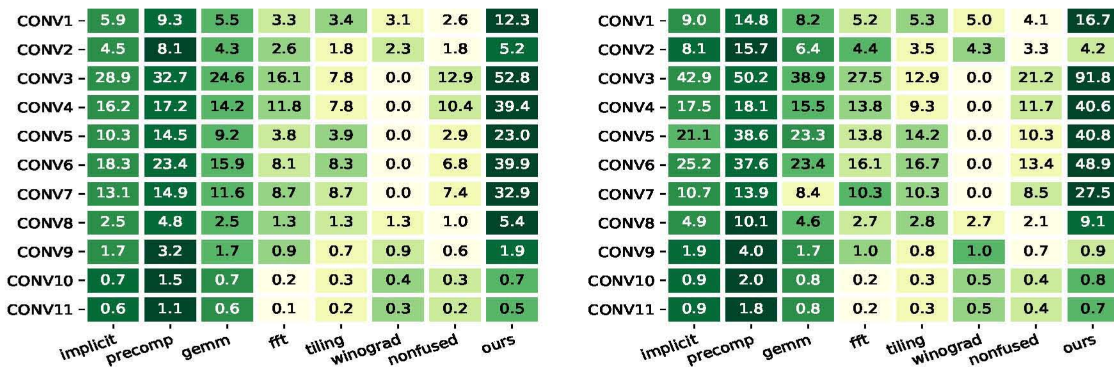


Fig. 4. Speedups of our approach and cuDNN over GEMM-im2col for one (left) and three (right) input channels.

layer configurations from four popular CNN models, namely, AlexNet [12], VGG [13], ResNet [14] and GoogleLeNet [15]. We use 3×3 and 5×5 filters with a batch size of 128. Table I lists the layer configurations used in this experiment.

2) Overall results: Figure 4 shows that our implementation achieves an average speedup of $19.5\times$ and $25.6\times$ over GEMM-im2col for one and three input channels, respectively. This translates to an improvement of $1.3\times$ and $1.1\times$ over the fastest algorithm in cuDNN, for one and three input channels, respectively. Since our approach does not optimize for input channels, it does not give performance improvement for layer configurations that have a large number of channels. This can be improved by careful optimizations on input channels. Nonetheless, our approach improves the performance of convolution layers with a small number of channels.

V. 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. Chellapilla et al. [7] developed an unrolling convolution algorithm. Mathieu et al. [16] proposed an FFT-based convolution to compute convolutions as pointwise products in the Fourier domain. Lavin et al. [3] 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.

Recent studies have looked into minimizing the memory overhead of the transformation phases. Cho et al. [4] 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. Iandola et al. [1] reduced memory communication of 2D convolutions on GPU. They also prefetched the image regions to the registers.

VI. Conclusion

Our approach improves the data locality for convolutional operations performed on the row and column directions to reduce the memory access. We evaluate our approach by applying it to 2D and multi-channel 2D convolutions and evaluate it on an NVIDIA RTX 2080Ti GPU platform. 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 the convolution tasks.

Authorized licensed use limited to: University of Leeds. Downloaded on January 06,2025 at 21:38:52 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

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

[8] W. Zhang, A. M. Cheng, and J. Subhlok, "Dwarfcode: a performance prediction tool for parallel applications," *IEEE Transactions on Computers*, vol. 65, no. 2, pp. 495–507, 2015.

[9] 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.

[10] P. Yalamanchili, U. Arshad, Z. Mohammed, P. Garigipati, P. Entschev, B. Kloppenborg, J. Malcolm, and J. Melonakos, "ArrayFire - A high performance software library for parallel computing with an easy-to-use API," Atlanta, 2015. [Online]. Available: https://github.com/arrayfire/arrayfire

[11] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[12] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *International Conference on Neural Information Processing Systems*, 2012.

[13] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[14] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, 2016, pp. 770–778.

[15] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, 2015, pp. 1–9.

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

Authorized licensed use limited to: University of Leeds. Downloaded on January 06,2025 at 21:38:52 UTC from IEEE Xplore. Restrictions apply.