

Efficient Convolution Pooling on the GPU

Shunsuke Suita^a, Takahiro Nishimura^a, Hiroki Tokura^a, Koji Nakano^{a,*},
Yasuaki Ito^a, Akihiko Kasagi^b, Tsuguchika Tabaru^b

^a*Department of Information Engineering, Hiroshima University, Japan*

^b*Fujitsu Laboratories, Japan*

Abstract

The main contribution of this paper is to show efficient implementations of the convolution-pooling in the GPU, in which the pooling follows the multiple convolution. Since the multiple convolution and the pooling operations are performed alternately in earlier stages of many Convolutional Neural Networks (CNNs), it is very important to accelerate the convolution-pooling. Our new GPU implementation uses two techniques, (1) convolution interchange with direct sum, and (2) conversion to matrix multiplication. By these techniques, the computational and memory access cost are reduced. Further the convolution interchange is converted to matrix multiplication, which can be computed by cuBLAS very efficiently. Experimental results using Tesla V100 GPU show that our new GPU implementation compatible with cuDNN for the convolution-pooling is expected 2.90 times and 1.43 times faster for fp32 and fp16 than the multiple convolution and then the pooling by cuDNN, respectively. the most popular library of primitives to implement the CNNs in the GPU.

1. Introduction

The GPU (Graphics Processing Unit) is a specialized circuit designed to accelerate computation for building and manipulating images [1, 2, 3]. Latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. Hence, GPUs have recently attracted the attention of many application developers. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified

*Corresponding author: nakano@cs.hiroshima-u.ac.jp (K. Nakano)

Device Architecture) [4], the computing engine for NVIDIA GPUs. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in NVIDIA GPUs. Application programs running on GPUs can be developed using CUDA C programming language. Further, NVIDIA provides several libraries of primitives to accelerate application programs. For example, cuBLAS [5], a linear algebra library including matrix computations, is optimized for each of GPU architecture generations, such as Kepler, Maxwell, Pascal, Volta, and Turing. So, we can attain the best performance for operations of linear algebra using cuBLAS, and it makes no sense to develop them using CUDA C language by ourselves in most cases.

GPUs have been used for accelerating machine learning by Deep Neural Networks (DNNs). In particular, Convolutional Neural Networks (CNNs), a kind of DNNs for images can be accelerated by GPUs very efficiently [3, 6]. NVIDIA provides cuDNN [7, 8], a GPU-accelerated library of primitives for DNNs such as the convolution and the pooling. Developers can use cuDNN APIs to implement DNN operations in GPUs. Further, popular machine learning frameworks such as TensorFlow, CNTK, PyTorch, and Caffe2 call cuDNN APIs to accelerate operations of DNN using GPUs. Hence, it is very important to improve library calls of cuDNN. The main purpose of this paper is to provide an efficient cuDNN-compatible GPU implementation for the convolution-pooling, in which the pooling follows the convolution as illustrated in Fig. 1. Since the convolution and the pooling are performed alternately in earlier stages of a Convolutional Neural Network (CNN), a kind of DNN for images, training and inference of CNNs can be accelerated by our cuDNN-compatible API. If developers write a CNN program for the convolution-pooling using cuDNNs, cuDNN APIs for the multiple convolution and that for the pooling are called in turn. The main contribution of this paper is to present more efficient GPU implementation for the convolution-pooling compatible with cuDNN. Thus, our GPU implementations enables developers to accelerate the computation of the CNN if two adjacent layers performs the convolution and then the pooling.

Our new GPU implementation for the convolution-pooling uses two techniques, (1) convolution interchange with direct sum, and (2) conversion to matrix multiplication. In (1), the direct sum operation is performed before the convolution to obtain the same results. The computational and memory access cost are reduced by this technique. To further accelerate the convolution-pooling, the computation in (1) is converted to equivalent matrix multiplication, which can be computed by cuBLAS very efficiently. In this

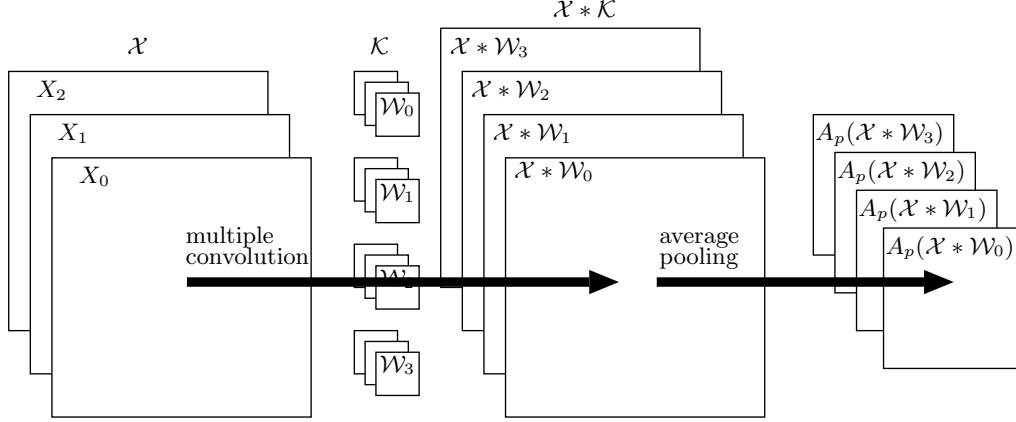


Figure 1: The convolution-pooling for $I = 3$ input channels and $R = 4$ output channels

paper, we show theoretical analysis of the computational cost, which is the total number of operations, of the convolution-pooling. From the theoretical analysis, our convolution-pooling algorithm reduces the computational cost of the convolution-pooling. Thus, our acceleration technique is applicable for any architecture if regular memory access performed by it does not have large memory access penalty.

There are a lot of approaches to accelerate the operations in the DNN [9, 10, 11, 12]. In [13], the convolution interchange technique to accelerate the convolution-pooling has been presented. They use the Summed Area Table (SAT) of the input channels to reduce the computational cost, which is inverse proportional to the pooling size. However, in most DNNs, the pooling of size only 2×2 is used. We have used the direct sum computation, which is more efficient than the SAT when the pool size is small. In addition, we have used the matrix multiplication conversion for the convolution interchange technique for further acceleration. We have used two techniques above and evaluate the performance on Tesla V100 GPU.

Our experimental results for the convolution-pooling for 3×3 kernels and 2×2 pooling for 32, 64, 128, 256, and 512 input/output channels with batch size 64 show that our convolution-pooling is expected 2.90 times and 1.43 times faster for fp32 (32-bit single precision floating point numbers) and fp16 (16-bit single precision floating point numbers) than the multiple convolution and then the pooling by cuDNN, respectively. Note that the convolution pooling for fp16 uses Tensorcore of the GPU, which can compute

the multiplication-addition of 4×4 matrices in one clock cycle [14]. Thus, the convolution-pooling can be accelerated using the two techniques.

This paper is organized as follows. In Section 2, we define the convolution-pooling formally and several acceleration techniques. We then go on to show how we use matrix multiplication to perform the multiple convolution in Section 3. Section 4 shows the details of GPU implementations for computing the convolution-pooling. Finally, we show experimental results using Tesla V100 GPU in Section 5. Section 6 concludes our work.

2. Convolution-pooling in the CNN

The main purpose of this section is to explain the details of convolution-pooling, in which pooling operation follows convolution operation and show *the computational cost*, which is the number of arithmetic operations such as addition and multiplication. Hence, it is almost equal to the running time of a sequential algorithm. In particular, we will discuss *a convolution-pooling layer*, in which a pooling layer follows a convolution layer.

2.1. Convolution-pooling and straightforward implementation

Let X and W be matrices of size $n \times n$ and $k \times k$, respectively. The *convolution* of X and W denoted by $X * W$ is a $(n - k + 1) \times (n - k + 1)$ matrix defined by the following formula:

$$(X * W)[i, j] = \sum_{i'=0}^{k-1} \sum_{j'=0}^{k-1} X[i + i', j + j'] W[i', j'] \quad (0 \leq i, j \leq n - k) \quad (1)$$

Sometimes *zero padding operation*, which expands the size of X or W by padding zero elements, is performed before the convolution to obtain an $n \times n$ resulting matrix. Usually, in the area of image processing and machine learning, $n \gg k$ holds and matrices X and W are called *a channel* and *a kernel*, respectively. For a set $\mathcal{X} = \{X_0, X_1, \dots, X_{I-1}\}$ of I channels and a set $\mathcal{W} = \{W_0, W_1, \dots, W_{I-1}\}$ of I kernels, we write $\mathcal{X} * \mathcal{W}$ to denote the element-wise sum of the pairwise convolutions, that is,

$$(\mathcal{X} * \mathcal{W})[i, j] = \sum_{l=0}^{I-1} (X_l * W_l)[i, j] \quad (0 \leq i, j \leq n - k) \quad (2)$$

Clearly, the computational cost of the multiple convolution is $O(n^2 k^2 I)$. Suppose that a set \mathcal{X} of I channels and R sets $\mathcal{K} = \{\mathcal{W}_0, \mathcal{W}_1, \dots, \mathcal{W}_{R-1}\}$ of I

kernels each are given. *The multiple convolution* is a task to compute R products

$$\mathcal{X} * \mathcal{K} = \{\mathcal{X} * \mathcal{W}_0, \mathcal{X} * \mathcal{W}_1, \dots, \mathcal{X} * \mathcal{W}_{R-1}\}.$$

Clearly, the total computational cost of $\mathcal{X} * \mathcal{K}$ is $O(n^2 k^2 IR)$. The reader should refer to Fig. 1 illustrating multiple convolution for $I = 3$ input channels and $R = 4$ output channels. Note that an input matrix and an output matrix are called *an input channel* and *an output channel* in the CNN.

The (average) pooling of a matrix is a down-sampling by dividing an input matrix into blocks, and computing the average of each block. More specifically, for an $n \times n$ matrix X , the resulting matrix $A_p(X)$ of the average pooling is an $\frac{n}{p} \times \frac{n}{p}$ matrix such that

$$A_p(X)[i, j] = \sum_{i'=pi}^{pi+p-1} \sum_{j'=pj}^{pj+p-1} X[i', j'] / p^2 \quad (0 \leq i, j \leq \frac{n}{p} - 1) \quad (3)$$

where $p \times p$ is *the pooling size*. Since the sum of p^2 input elements is computed for each element of the resulting $\frac{n}{p} \times \frac{n}{p}$ matrix, the computational cost is $p^2 \times (\frac{n}{p})^2 = O(n^2)$.

In the CNN, it is often the case that the pooling follows the multiple convolution as illustrated in Fig. 1. We call these computations combined *the convolution-pooling*, which is a task to output

$$A_p(\mathcal{X} * \mathcal{K}) = \{A_p(\mathcal{X} * \mathcal{W}_0), A_p(\mathcal{X} * \mathcal{W}_1), \dots, A_p(\mathcal{X} * \mathcal{W}_{R-1})\}$$

Clearly, the total computational cost to obtain these R matrices is $(O(n^2 k^2 I) + O(n^2)) \cdot R = O(n^2 k^2 IR)$, and we have,

Lemma 1. *The convolution-pooling can be done in $O(n^2 k^2 IR)$ computational cost.*

We will show that the computational cost can be reduced to $O(\frac{n^2 k^2 IR}{p^2})$ later.

2.2. Fused kernel implementation of convolution-pooling layer

The convolution operation is associative, that is, $(X * Y) * Z = X * (Y * Z)$ holds for any matrices X , Y , and Z . We will show that, using this associative law, the convolution-pooling can be implemented by the convolution with the down-sampling.

Let S_p be a down-sampling operation to pick one element in each $p \times p$ block of a matrix X . More specifically, $S_p(X)$ of size $\frac{n}{p} \times \frac{n}{p}$ is defined as follows:

$$S_p(X)[i, j] = X[pi, pj] \quad (0 \leq i, j \leq \frac{n}{p} - 1).$$

Let α_p be a kernel of size $p \times p$ with every element taking value $\frac{1}{p^2}$. Clearly, the convolution $X * \alpha_p$ corresponds to the average filter for X . Hence, the resulting matrix $A_p(X)$ of the average pooling for X can be computed by evaluating $S_p(X * \alpha_p)$, that is, $A_p(X) = S_p(X * \alpha_p)$ always holds. Thus, each resulting matrix of convolution-pooling can be obtained by the following formula:

$$A_p(\mathcal{X} * \mathcal{W}_r) = S_p\left(\sum_{l=0}^{I-1} (X_l * (W_{r,l} * \alpha_p))\right) \quad (0 \leq r \leq R-1), \quad (4)$$

where $W_{r,l}$ denotes the l -th kernel in \mathcal{W}_r . We can think that each *fused kernel* $W_{r,l} * \alpha_p$ is a fixed matrix of size $(k+p-1) \times (k+p-1)$. After that, $X_l * (W_{r,l} * \alpha_p)$ is computed. However, it is not necessary to compute all matrix elements of $X_l * (W_{r,l} * \alpha_p)$, because the down-sampling S_p is performed; Only one element in every $p \times p$ block is necessary. Thus, we have,

Lemma 2. *The convolution-pooling by fused kernels can be done in $O(\frac{n^2(k+p)^2IR}{p^2})$ computational cost.*

The computational cost is not better than that of the straightforward implementation shown for Lemma 1. However, since convolution operation is performed only once, fused kernel implementation can be faster from the practical point of view.

2.3. Convolution interchange for the convolution-pooling

This section shows the convolution interchange technique to implement the convolution-pooling, and it runs in only $O(\frac{n^2k^2IR}{p^2})$ computational cost by computing the summed area table as illustrated in Fig. 2. We then go on to show that our direct sum technique for the convolution interchange for further acceleration.

Since convolution operation is associative and commutative, we can rewrite formula (4) as follows:

$$A_p(\mathcal{X} * \mathcal{W}_r) = S_p\left(\sum_{l=0}^{I-1} ((X_l * \alpha_p) * W_{r,l})\right) \quad (0 \leq r \leq R-1). \quad (5)$$

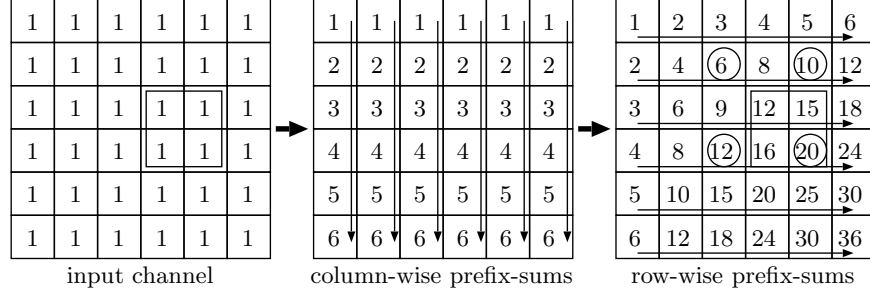


Figure 2: The summed area table (SAT) computed by column-wise prefix-sums and then by row-wise prefix-sums

Clearly, each $X_l * \alpha_p$ can be computed in $O(n^2 p^2)$ computational cost. If we use *the summed area table (SAT)* as presented in [13], the computational cost can be reduced to $O(n^2)$. For an $n \times n$ matrix X , the SAT $S(X)$ is defined as follows:

$$S(X)[i, j] = \sum_{i'=0}^i \sum_{j'=0}^j X[i', j'] \quad (6)$$

1 It is known that $S(X)$ can be obtained by computing the column-wise prefix-sums of X and then computing the row-wise prefix-sums [15, 16]. Hence, $S(X)$ can be computed in $O(n^2)$ computational cost. The sum of any rectangular block can be computed by four elements of $S(X)$. For example, the sum of any $p \times p$ block of X can be computed by four elements of $S(X)$ as follows:

$$\begin{aligned} \sum_{i'=i}^{i+p-1} \sum_{j'=j}^{j+p-1} X[i', j'] &= S(X)[i+p-1, j+p-1] + S(X)[i-1, j-1] \\ &\quad - S(X)[i-1][j+p-1] - S(X)[i+p-1][j-1] \end{aligned} \quad (7)$$

For example, in Fig. 2, the sum of elements in the 2×2 square can be computed by four elements with circles such that $20 + 6 - 10 - 12 = 4$. Thus, each element of $X_l * \alpha_p$ can be computed by $O(1)$ computational cost by computing the sum of each $p \times p$ region and dividing it by p^2 , and so the total computational cost to obtain all $X_l * \alpha_p$ for all l ($0 \leq l \leq I-1$) is $O(n^2 I)$. After that, each element of $S_p(\sum_{l=0}^{I-1} ((X_l * \alpha_p) * W_l))$ is computed in $O(k^2 I)$ computational cost. Since we have $\frac{n^2}{p^2}$ elements, $S_p(\sum_{l=0}^{I-1} ((X_l * \alpha_p) * W_l))$

can be computed in $\frac{n^2}{p^2} \cdot O(k^2 I) = O(\frac{n^2 k^2 I}{p^2})$ computational cost. Since the convolution-pooling is performed for R sets of I kernels each, we have,

Theorem 2.1. *The convolution-pooling by the convolution interchange can be completed in $O(\frac{n^2 k^2 I R}{p^2} + n^2 I)$ computational cost.*

Clearly, if $k\sqrt{R} \geq p$, then the computational cost is $O(\frac{n^2 k^2 I R}{p^2})$. Actually, p is smaller than both k and R in practical implementations of CNNs. Further, in the CNN, most pooling operation is performed with parameter $p = 2$. If this is the case, it makes no sense to compute the SAT to obtain $X_l * \alpha_p$. By computing the sum of each neighboring pair in row direction, and then by computing the sum of each neighboring pair in row direction, we can obtain the sum of every 2×2 block as illustrated in Fig. 3. By dividing each sum by 4, we can obtain $X_l * \alpha_2$ in $O(n^2 I)$. For later reference, we call this computation *direct sum*. After computing the direct sum of each input channel, we can compute $S_p(\sum_{l=0}^{I-1} ((X_l * \alpha_2) * W_{r,l}))$ to complete the convolution-pooling in the same way.

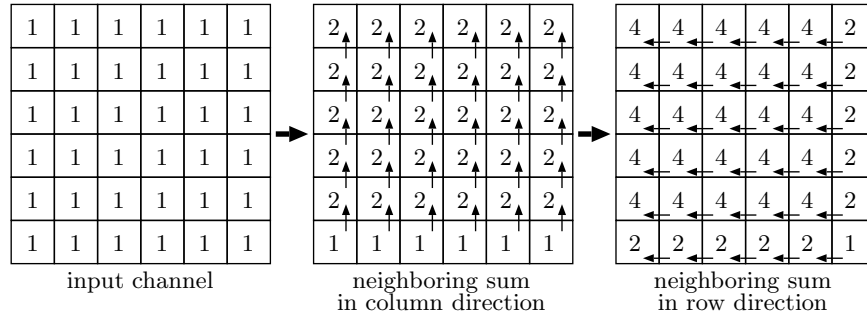


Figure 3: The direct-sum computation to obtain $X_l * \alpha_2$

3. Matrix multiplication conversion for convolution-pooling

This section explains the matrix multiplication conversion technique known as im2col, which is implemented in Python, MATLAB, cuDNN, etc. It converts input channels combined are converted in a single matrix and kernel set combined is also converted in a single matrix so that the product of them equals to the result of the multiple convolution. We apply this technique to the convolution-pooling, and use cuBLAS to multiply two matrices.

We first show that the multiple convolution represented as formula (2) can be computed by a matrix multiplication. First, I input channels $\mathcal{X} = \{X_0, X_1, \dots, X_{I-1}\}$ and R kernel sets $\mathcal{K} = \{\mathcal{W}_0, \mathcal{W}_1, \dots, \mathcal{W}_{R-1}\}$ are converted into two matrices $D(\mathcal{X})$ and $V(\mathcal{K})$ of size $(n - k + 1)^2 \times k^2 I$ and $k^2 I \times R$ as illustrated in Fig. 4. The matrix $D(\mathcal{X})$ has $k^2 I$ columns such that consecutive k^2 columns are copied from an input channel. Each row in consecutive k^2 columns corresponds to a $k \times k$ block. For example, dashed blocks of X_0 and X_1 are arranged in the top row of $D(\mathcal{X})$. Hence, $D(\mathcal{X})$ has $(n - k + 1)^2$ rows. Each column of the matrix $V(\mathcal{K})$ corresponds to a kernel set and the value of I kernels in a kernel set are copied in the corresponding column as illustrated in Fig. 4. From the figure, the reader should have no difficulty to confirm that the product of $D(\mathcal{X})$ and $V(\mathcal{K})$ is equal to the values of R output channels $\mathcal{X} * \mathcal{W}_i$ for all i ($0 \leq i \leq R - 1$). Thus, the multiple convolution can be obtained by the product of $D(\mathcal{X})$ and $V(\mathcal{K})$,

The computational cost for generating $D(\mathcal{X})$ is $O((n - k + 1)^2 \times k^2 I) \leq O(n^2 k^2 I)$. Also, that for $V(\mathcal{K})$ is $O(k^2 I R)$. Their product can be computed in $O((n - k + 1)^2 \cdot k^2 I \cdot R) \leq O(n^2 k^2 I R)$. Hence, the total computing cost is $O(n^2 k^2 I R)$.

We will show that the same technique can be used for the convolution interchange, which computes $S_p(\sum_{l=0}^{I-1} ((X_l * \alpha_p) * W_l))$. Suppose that $\mathcal{X} * \alpha_p = \{X_0 * \alpha_p, X_1 * \alpha_p, \dots, X_{I-1} * \alpha_p\}$ and R kernel sets $\mathcal{K} = \{\mathcal{W}_0, \mathcal{W}_1, \dots, \mathcal{W}_{R-1}\}$ are given. Clearly, by the product of two matrices $D(X_l * \alpha_p)$ and $V(\mathcal{K})$, we can obtain $\sum_{l=0}^{I-1} ((X_l * \alpha_p) * W_{r,l})$. Since we need the down-sample $S_p(\sum_{l=0}^{I-1} ((X_l * \alpha_p) * W_{r,l}))$, which is obtained by selecting one element from every $p \times p$ block of $\sum_{l=0}^{I-1} ((X_l * \alpha_p) * W_{r,l})$, we can remove unnecessary rows from $D(X_l * \alpha_p)$ to obtain $S_p(\sum_{l=0}^{I-1} ((X_l * \alpha_p) * W_{r,l}))$. Let $D_p(X_l * \alpha_p)$ denote the matrix obtained by this down-sampling such that one out of every p^2 rows in $D(X_l * \alpha_p)$ is picked appropriately. The product of $D_p(X_l * \alpha_p)$ and $V(\mathcal{K})$ can be computed in $O(\frac{n^2 k^2 I R}{p^2})$ and so the total computational cost is $O(\frac{n^2 k^2 I R}{p^2} + n^2 I)$.

Also, the same technique can be used for fused kernels. Let $\mathcal{W}'_i = \{W_0 * \alpha_p, W_1 * \alpha_p, \dots, W_{I-1} * \alpha_p\}$ be a set of I fused kernels, and $\mathcal{K}' = \{\mathcal{W}'_0, \mathcal{W}'_1, \dots, \mathcal{W}'_{R-1}\}$ be R sets of I fused kernels. The convolution-pooling by fused kernels can be computed by the product of $D_p(\mathcal{X})$ and $V(\mathcal{K}')$. The total computational cost is $O(\frac{n^2 (k+p)^2 I R}{p^2})$.

X_0						X_1						\mathcal{W}_0			\mathcal{W}_0			\mathcal{W}_0		
0	1	2	3	4	5	6	7	8	9	10	11	a	b	c	j	k	l	s	t	u
6	7	8	9	10	11	12	13	14	15	16	17	d	e	f	m	n	o	v	w	x
12	13	14	15	16	17	18	19	20	21	22	23	g	h	i	p	q	r	y	z	A
18	19	20	21	22	23	24	25	26	27	28	29	a	b	c	j	k	l	s	t	u
24	25	26	27	28	29	30	31	32	33	34	35	d	e	f	m	n	o	v	w	x
30	31	32	33	34	35	36	37	38	39	40	41	g	h	i	p	q	r	y	z	A

$D(\mathcal{X})$																		$V(\mathcal{K})$		
0	1	2	6	7	8	12	13	14	0	1	2	6	7	8	12	13	14	a	j	s
1	2	3	7	8	9	13	14	15	1	2	3	7	8	9	13	14	15	b	k	t
2	3	4	8	9	10	14	15	16	2	3	4	8	9	10	14	15	16	c	l	u
3	4	5	9	10	11	15	16	17	3	4	5	9	10	11	15	16	17			
6	7	8	12	13	14	18	19	20	6	7	8	12	13	14	18	19	20	i	r	A
7	8	9	13	14	15	19	20	21	7	8	9	13	14	15	19	20	21	a	j	s
8	9	10	14	15	16	20	21	22	8	9	10	14	15	16	20	21	22	b	k	t
9	10	11	15	16	17	21	22	23	9	10	11	15	16	17	21	22	23	c	l	u
12	13	14	18	19	20	24	25	26	12	13	14	18	19	20	24	25	26			
																		i	r	A

Figure 4: The multiple convolution by matrix multiplication for $I = 2$ input channels $\mathcal{X} = \{X_0, X_1\}$ with $R = 3$ kernel sets $\mathcal{K} = \{\mathcal{W}_0, \mathcal{W}_1, \mathcal{W}_2\}$.

4. GPU implementations

We have implemented five methods, cuDNN(naive), cuDNN(fused), cuBLAS(fused), cuDNN(direct), and cuBLAS(direct) to compute the convolution-pooling as follows.

cuDNN(naive) The multiple convolution $\mathcal{X} * \mathcal{K}$ is computed by the cuDNN and then the pooling $A_p(\mathcal{X} * \mathcal{K})$ is computed by cuDNN.

cuDNN(fused) All fused kernels $W_{r,l} * \alpha_p$ are computed in advance. The multiple convolution $S_p(\sum_{l=0}^{I-1} (X_l * (W_{r,l} * \alpha_p)))$ is computed for each r -th kernel set by cuDNN.

cuBLAS(fused) A matrix $V(\mathcal{K}')$ is generated from the resulting values of $W_{r,l} * \alpha_p$ in advance. A matrix $D_p(\mathcal{X})$ is generated by our CUDA C program and the product $D_p(\mathcal{X}) \cdot V(\mathcal{K}')$ is computed by cuBLAS.

cuDNN(direct) Each $X_l * \alpha_p$ is computed by the direct sum using our CUDA C program and then the multiple convolution $S_p(\sum_{l=0}^{I-1} ((X_l * \alpha_p) * W_{r,l}))$ is computed by cuDNN.

cuBLAS(direct) Each $X_l * \alpha_p$ is computed by the direct sum and then $D_p(X_l * \alpha_p)$ and $V(\mathcal{K})$ are generated by our CUDA C program. The product of $D_p(X_l * \alpha_p)$ and $V(\mathcal{K})$ are computed by cuBLAS.

If developers implement the convolution-pooling using cuDNN as it is, they will use cuDNN(naive) implementation. Further, if they use DNN frameworks such as Chainer, PyTorch, and TensorFlow, the convolution-pooling is executed on the GPU as cuDNN(naive). Thus, the performance of cuDNN(naive) approximates that using DNN frameworks. If developers know the fused kernel technique, they may use cuDNN(fused) to implement the convolution-pooling. Both cuDNN(direct) and cuBLAS(direct) use the convolution interchange and the direct sum. Their difference is to use cuDNN or cuBLAS to compute the convolution.

Also, please note that the convolution performed for multiple channel sets called *batch* at the same time in most DNNs. More specifically, let B denote the size of batch, *i.e.* the number of channel sets. The multiple convolution of a batch of size B performs the convolution for B channel sets of I channels each with respect to a single kernel set of I kernels. We should evaluate the performance of the running time of the convolution for a batch.

We will explain the details of the five implementations.

4.1. *cuDNN(naive)*

The convolution of cuDNN can have several options of convolution algorithms. We call parameters of the multiple convolution such as data type (double, float, half), the size $n \times n$ and the number I of channels, the size $k \times k$ and the number I of kernels, and the batch size B , *the configuration of the multiple convolution*. We use the function call `cudnnGetConvolutionForwardAlgorithm()`, which returns the best algorithm for the configuration of the multiple convolution. After that, we first execute `cudnnGetConvolutionForwardWorkspaceSize()` with the selected best algorithm and the configuration which allocates memory space in the global memory for multiple con-

volution computation by cuDNN. We call `cudaDnnConvolutionForward()` with the best selected algorithm and the configuration to perform the multiple convolution. Finally, we call `cudaDnnPoolingForward()` with the configuration to perform the pooling. The running time of cuDNN(naive) are evaluated by the sum of the running time of `cudaDnnConvolutionForward()` and `cudaDnnPoolingForward()`. That for `cudaDnnGetConvolutionForwardAlgorithm()` is excluded, because it is executed only once.

4.2. *cuDNN(fused)*

We first compute the fused kernel $W_{r,l} * \alpha_p$ for a kernel set by our CUDA C program in an obvious way. Similarly, `cudaDnnGetConvolutionForwardAlgorithm()` is called to obtain the best algorithm for the configuration. We then executes `cudaDnnGetConvolutionForwardWorkspaceSize()` to allocate the global work memory space, and `cudaDnnConvolutionForward()` with stride p to compute the multiple convolution. Since the computation of fused kernel $W_{r,l} * \alpha_p$ is executed once for the same kernel set, and `cudaDnnGetConvolutionForwardWorkspaceSize()` is called only once, the running time of `cudaDnnConvolutionForward()` is used for evaluating the performance of cuDNN(fused).

4.3. *cuBLAS(fused)*

We first compute the fused filter and convert it to the corresponding matrix $V(\mathcal{K}')$. We then covert input channels of each channel set to the corresponding matrix $D_p(\mathcal{X})$ by our CUDA C program. Since we have B channel sets, the corresponding B matrices are concatenated into one large matrix. Finally, we execute `cublasSgemmStridedBatched()` to complete the convolution-pooling. Since the $V(\mathcal{K}')$ is computed only once, the running time of the computation of the corresponding matrix $D_p(\mathcal{X})$ and `cublasSgemmStridedBatched()` are evaluated.

4.4. *cuDNN(direct)*

Each $X_l * \alpha_p$ is computed by the direct sum using our CUDA C program. Similarly to cuDNN(naive), the best algorithm is obtained by calling `cudaDnnGetConvolutionForwardAlgorithm()` for the configuration of the multiple convolution. We then executes `cudaDnnGetConvolutionForwardWorkspaceSize()` to allocate the global work memory space, and `cudaDnnConvolutionForward()` with stride p to compute the multiple convolution. Since `cudaDnnGetConvolutionForwardWorkspaceSize()` is executed only once, the running time of the computation of $X_l * \alpha_p$ by our CUDA C program and `cudaDnnConvolutionForward()` are used to evaluate the performance.

4.5. *cuBLAS(direct)*

We first convert kernels to the corresponding matrix $V(\mathcal{K})$ by our CUDA C program. We then compute each $X_l * \alpha_p$ by the direct sum and convert it to the corresponding matrix $D_p(X_l * \alpha_p)$ by our CUDA C program. We execute `cublasSgemmStridedBatched()` to compute the product of $D_p(X_l * \alpha_p)$ and $V(\mathcal{K})$. Since $V(\mathcal{K})$ for a kernel set \mathcal{K} is computed only once, the running time of the computation of $D_p(X_l * \alpha_p)$ and `cublasSgemmStridedBatched()` are evaluated.

4.6. *Tensorcore*

Volta architecture of NVIDIA GPUs has Tensorcores, which can compute the multiply-add of 4×4 matrices of fp16s (16-bit half precision floating numbers) as illustrated in Fig. 5. NVIDIA Tesla V100 has 640 Tensorcores, each of which can compute the multiply-add in every clock cycle. We have also developed the five implementations of the convolution-pooling for fp16 using Tensorcores. To use Tensorcore for the five implementations, we have used fp16 option for cuDNN and cuBLAS.

$$\begin{array}{|c|c|c|c|} \hline D_{0,0} & D_{0,1} & D_{0,2} & D_{0,3} \\ \hline D_{1,0} & D_{1,1} & D_{1,2} & D_{1,3} \\ \hline D_{2,0} & D_{2,1} & D_{2,2} & D_{2,3} \\ \hline D_{3,0} & D_{3,1} & D_{3,2} & D_{3,3} \\ \hline \end{array} = \begin{array}{|c|c|c|c|} \hline A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ \hline A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ \hline A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ \hline A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \\ \hline \end{array} \times \begin{array}{|c|c|c|c|} \hline B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ \hline B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ \hline B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ \hline B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \\ \hline \end{array} + \begin{array}{|c|c|c|c|} \hline C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ \hline C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ \hline C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ \hline C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \\ \hline \end{array}$$

Figure 5: Illustrating the computation performed by a Tensorcore

5. Experimental results

Table 1 shows the running time of the convolution-pooling by cuDNN(naive), cuDNN(fused), cuBLAS(fused), cuDNN(direct), and cuBLAS(direct) for input channel size from 8×8 to 64×64 and for the number of input/output channels from 32/32 to 512/512. The data type is fp32 (32-bit single precision floating point number). We have used NVIDIA Tesla V100 with cuDNN v7.1.4 and cuBLAS v9.0. Since kernels of size 3×3 and the pooling for 2×2 are used in the DNNs, we use these parameters for the experiments. The running time is evaluated for 64 sets of the multiple convolution, thus, it corresponds to batch size 64 in the DNN. The running time in the table is the

average of 100 iterations. In the table, the best running time of the five implementations for each parameter set is highlighted. It also shows the speedups of the best result of cuDNN(direct) and cuBLAS(direct) over cuDNN(naive), and that of over the best result of cuDNN(fused) and cuBLAS(fused). From the table, we can see that either cuDNN(direct) or cuBLAS(direct) is always faster than cuDNN(naive) for every case. Also, they are faster than cuDNN(fused) and cuBLAS(fused) in most cases. They are slower for few cases but the difference is quite small. The speedup for cuDNN(naive) is from 1.34 to 9.49 and the average speedup is 2.90. The maximum speedup of 9.49 is achieved for 128 input/output channels of size 64×64 , because cuDNN(naive) does not select an appropriate algorithm and takes a lot of time for the multiple convolution.

Table 2 shows the running time for the same convolution-pooling using fp16. We have used Tensorcore of these five implementations whenever possible. The speedup for cuDNN(naive) is from 0.406 to 3.90 and the average speedup of them is 1.43. Also, we can see that cuDNN can run fastest for channels with small size. This is because cuDNN uses Winograd algorithm [17, 18], which performs the multiple-convolution very efficiently for channels with small size.

Unfortunately, the best algorithm of the five differs depending on configurations. Usually, DNNs have many layers with different configurations. We may choose the best one for each layer to minimize the total computing time.

6. Conclusion

We have presented new GPU implementations for the convolution-pooling based on convolution interchange with direct sum. Experimental results using Tesla V100 GPU show that our new GPU implementation compatible with cuDNN for the convolution-pooling is expected 2.90 times and 1.43 times faster for fp32 and fp16 than the multiple convolution and then the pooling by cuDNN, respectively.

References

- [1] W. W. Hwu, GPU Computing Gems Emerald Edition, Morgan Kaufmann, 2011.

- [2] K. Ogawa, Y. Ito, K. Nakano, Efficient Canny edge detection using a GPU, in: Proc. of International Conference on Networking and Computing, IEEE CS Press, 2010, pp. 279–280.
- [3] N. Matsumura, H. Tokura, Y. Kuroda, Y. Ito, K. Nakano, Tile art image generation using conditional generative adversarial networks, in: Proc. of International Symposium on Computing and Networking Workshops, 2018, pp. 209–215.
- [4] NVIDIA Corporation, NVIDIA CUDA C programming guide version 4.0 (2011).
- [5] NVIDIA Corporation, CUBLAS LIBRARY user guide, <https://docs.nvidia.com/cuda/cublas/index.html> (Feb. 2019).
- [6] K. Chellapilla, S. Puri, P. Simard, High performance convolutional neural networks for document processing, in: Proc. of International Workshop on Frontiers in Handwriting Recognition, 2006.
- [7] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, E. Shelhamer, cuDNN: Efficient primitives for deep learning, CoRR abs/1410.0759 (Aug. 2014).
- [8] NVIDIA Corporation, CUDNN developer guide, <https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html> (Feb. 2019).
- [9] Y. Cheng, D. Wang, P. Zhou, T. Zhang, A survey of model compression and acceleration for deep neural networks, CoRR abs/1710.09282 (Oct. 2017).
- [10] C. Li, Y. Yang, M. Feng, S. Chakradhar, H. Zhou, Optimizing memory efficiency for deep convolutional neural networks on GPUs, in: Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2016.
- [11] V. Sze, Y.-H. Chen, T.-J. Yang, J. S. Emer, Efficient processing of deep neural networks: A tutorial and survey, Proceedings of the IEEE 105 (12) (2017) 2295 – 2329.

- [12] Q. Zhang, M. Zhang, T. Chen, Z. Sun, Y. Ma, B. Yu, Recent advances in convolutional neural network acceleration, *Neurocomputing* 323 (2019) 37–51.
- [13] A. Kasagi, T. Tabaru, H. Tamura, Fast algorithm using summed area tables with unified layer performing convolution and average pooling, in: *Proc. of International Workshop on Machine Learning for Signal Processing*, 2017.
- [14] NVIDIA Corporation, NVIDIA TESLA V100 GPU architecture, <https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf> (Aug. 2017).
- [15] Y. Emoto, S. Funasaka, H. Tokura, T. Honda, K. Nakano, Y. Ito, An optimal parallel algorithm for computing the summed area table on the GPU, in: *Proc. of International Parallel and Distributed Processing Symposium Workshops*, 2018, pp. 763–772.
- [16] A. Kasagi, K. Nakano, Y. Ito, Parallel algorithms for the summed area table on the asynchronous hierarchical memory machine, with GPU implementations, in: *Proc. of International Conference on Parallel Processing (ICPP)*, 2014, pp. 251–250.
- [17] S. Winograd, *Arithmetic Complexity of Computations*, SIAM, 1980.
- [18] A. Lavin, S. Gray, Fast algorithms for convolutional neural networks, in: *Proc. of IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

Table 1: The running time (ms) of the convolution-pooling for fp32: 3×3 kernels and 2×2 pooling for I input channels and R output channels for batch size 64

input channel size: 8×8					
channels I/R	32/32	64/64	128/128	256/256	512/512
cuDNN(naive)	0.104	0.137	0.206	0.651	1.87
cuDNN(fused)	0.105	0.153	0.259	0.425	0.961
cuBLAS(fused)	0.103	0.141	0.281	0.929	3.17
cuDNN(direct)	0.107	0.133	0.340	0.341	0.701
cuBLAS(direct)	0.0473	0.0739	0.154	0.479	1.92
Speed-up:naive	2.20	1.85	1.34	1.91	2.67
Speed-up:fused	2.18	1.91	1.68	1.25	1.37
input channel size: 16×16					
channels I/R	32/32	64/64	128/128	256/256	512/512
cuDNN(naive)	0.186	0.262	0.663	1.87	6.51
cuDNN(fused)	0.102	0.155	0.299	0.824	3.18
cuBLAS(fused)	0.110	0.174	0.335	0.922	3.34
cuDNN(direct)	0.112	0.146	0.342	0.616	1.92
cuBLAS(direct)	0.0488	0.101	0.192	0.577	1.89
Speed-up:naive	3.81	2.59	3.45	3.24	3.44
Speed-up:fused	2.09	1.53	1.56	1.43	1.68
input channel size: 32×32					
channels I/R	32/32	64/64	128/128	256/256	512/512
cuDNN(naive)	0.247	0.42	2.03	5.98	20.9
cuDNN(fused)	0.156	0.254	0.836	2.82	10.7
cuBLAS(fused)	0.294	0.538	1.05	3.3	12.5
cuDNN(direct)	0.163	0.255	0.626	1.85	7.25
cuBLAS(direct)	0.164	0.315	0.610	1.85	7.02
Speed-up:naive	1.52	1.65	3.33	3.23	2.98
Speed-up:fused	0.957	0.996	1.37	1.52	1.52
input channel size: 64×64					
channels I/R	32/32	64/64	128/128	256/256	512/512
cuDNN(naive)	0.936	1.85	20.6	13.9	47
cuDNN(fused)	0.327	1.19	4.03	13.5	49.3
cuBLAS(fused)	0.989	1.96	4.06	12.8	44.3
cuDNN(direct)	0.355	0.685	2.17	7.03	24.2
cuBLAS(direct)	0.548	1.12	2.28	7.01	25.0
Speed-up:naive	2.64	2.70	9.49	1.98	1.94
Speed-up:fused	0.921	1.74	1.86	1.83	1.83

Table 2: The running time (ms) of the convolution-pooling for fp16 using Tensorcore: 3×3 kernels and 2×2 pooling for I input channels and R output channels for batch size 64

input channel size: 8×8					
channels I/R	32/32	64/64	128/128	256/256	512/512
cuDNN(naive)	0.0705	0.0636	0.0750	0.106	0.227
cuDNN(fused)	0.108	0.0974	0.126	0.221	0.509
cuBLAS(fused)	0.0680	0.0979	0.144	0.458	1.65
cuDNN(direct)	0.0773	0.0984	0.204	0.351	0.618
cuBLAS(direct)	0.0341	0.0487	0.0811	0.261	0.940
Speed-up:naive	2.07	1.31	0.925	0.406	0.368
Speed-up:fused	2.00	2.00	1.55	0.845	0.824
input channel size: 16×16					
channels I/R	32/32	64/64	128/128	256/256	512/512
cuDNN(naive)	0.133	0.0967	0.129	0.237	0.542
cuDNN(fused)	0.110	0.190	0.164	0.280	0.576
cuBLAS(fused)	0.0716	0.114	0.187	0.541	1.83
cuDNN(direct)	0.0830	0.145	0.225	0.567	1.91
cuBLAS(direct)	0.0341	0.0579	0.110	0.321	1.05
Speed-up:naive	3.90	1.67	1.18	0.740	0.516
Speed-up:fused	2.10	1.97	1.50	0.872	0.548
input channel size: 32×32					
channels I/R	32/32	64/64	128/128	256/256	512/512
cuDNN(naive)	0.184	0.261	0.362	0.737	1.83
cuDNN(fused)	0.0894	0.122	0.222	0.643	2.03
cuBLAS(fused)	0.146	0.255	0.472	0.973	2.77
cuDNN(direct)	0.128	0.245	0.616	1.88	6.94
cuBLAS(direct)	0.0846	0.159	0.297	0.610	1.61
Speed-up:naive	2.17	1.64	1.22	1.21	1.14
Speed-up:fused	1.06	0.770	0.747	1.05	1.26
input channel size: 64×64					
channels I/R	32/32	64/64	128/128	256/256	512/512
cuDNN(naive)	0.341	0.606	1.48	4.40	15.3
cuDNN(fused)	0.213	0.374	1.05	2.11	7.16
cuBLAS(fused)	0.446	0.857	1.73	3.84	10.5
cuDNN(direct)	0.311	0.676	2.14	6.68	24.4
cuBLAS(direct)	0.282	0.543	1.08	2.34	6.16
Speed-up:naive	1.21	1.11	1.37	1.88	2.48
Speed-up:fused	0.757	0.689	0.971	0.903	1.16