FAST CONVOLUTION KERNELS ON PASCAL GPU WITH HIGH MEMORY EFFICIENCY

Qiong Chang

Grad. of Systems and Information Engineering
University of Tsukuba
1-1-1 Ten-nou-dai Tsukuba Japan
cq@darwin.esys.tsukuba.ac.jp

Masaki Onishi

National Institute of Advanced Industrial
Science and Technology (AIST)
1-1-1 Umezono Tsukuba Japan
onishi@ni.aist.go.jp

Tsutomu Maruyama

Grad. of Systems and Information Engineering
University of Tsukuba
1-1-1 Ten-nou-dai Tsukuba Japan
maruyama@darwin.esys.tsukuba.ac.jp

ABSTRACT

The convolution computation is widely used in many fields, especially in CNNs. Because of the rapid growth of the training data in CNNs, GPUs have been used for their acceleration and memory-efficient algorithms have been the focus of attention due to their high performance. In this paper, we propose two convolution kernels for single-channel and multi-channel convolution respectively. Our two methods achieve high performance by hiding the access delay of the global memory efficiently, and achieving high ratio of floating point Fused Multiply-Add operations per fetched data from the global memory. In comparison to the latest Cudnn library developed by Nvidia aimed to accelerate the deep-learning computation, the average performance improvement of our research is 2.6X for the single-channel, and 1.4X for the multi-channel.

Keywords: Convolution, GEMM, GPU

1 INTRODUCTION

Convolution is widely used as a fundamental operation in many applications such as computer vision, natural language processing, signal processing. Especially, the Convolution Neural Network (CNN), a popular model used for deep-learning, is widely used in many applications such as image recognition (Poznanski and Wolf 2016), (Simonyan and Zisserman 2014), video analysis (Yu et al. 2016), natural language processing (Zhang and Wallace 2017), and has yielded remarkable results.

Recently, many CNN models have been proposed, such as AlexNet (Krizhevsky et al. 2017), GoogleNet (Szegedy et al. 2015), VGGNet (Simonyan and Zisserman 2014), ResNet (He et al. 2016), etc. They are used in many areas, and are improved steadily. The sizes of their networks have grown larger, which has led to the increase of their processing time. The CNNs have several layers, but a large portion of the total processing time is used for the convolution layers. Because of the high inherent parallelism of the convolution algorithms, many researchers are exploring to use GPUs to accelerate them. They can be

divided into four categories: 1) *Direct-based* method (Li et al. 2015), 2) *FFT-based* method (Mathieu et al. 2013), 3) *Winograd-based* method (Lavin and Gray 2016) and 4) *General matrix multiplication (GEMM) based* method (Tan et al. 2011). Recently, many of the algorithms and their modified versions have been aggregated into a public library called Cudnn by Nvidia, which aims to accelerate the deep-learning platforms like Chainer (Tokui et al. 2015), Caffe (Jia et al. 2014), etc. As the volume of the processing data used in deep-learning increases, the memory-efficient algorithms play an increasingly more important role, resulting in the constant proposal of many improved versions. Among them, *Implicit-GEMM* (Chetlur et al. 2014) has been included in the Cudnn because of its high efficiency. It divides the feature map and filter data into many sub-blocks and converts them to many sub-matrices only using on-chip memory of GPU. This method is very memory-efficient, and achieved a high ratio of floating point Fused Multiply-Add (FMA) operations per data transferred from the global memory. In (Chen et al. 2017), two memory-efficient methods were proposed. Both of which are faster than the Implicit-GEMM. However, their performances are negatively affected when the feature map size is smaller than 32, because it fixes the amount of the data assigned to each *SM*, which sometimes is not suitable to the small feature map. In modern CNN models as described above, more than half of the convolution layers are used for the calculation of the images smaller than 32 (such as 28, 14, 7), meaning that (Chen et al. 2017) cannot handle the modern CNN models efficiently.

In this paper, we propose two methods to accelerate the convolution kernels for single-channel and multi-channel. In these methods, the memory access is optimized to achieve higher performance. For the single-channel kernel, our approach follows the computation method proposed in (Chen et al. 2017), however, the data is divided and assigned to each *SM* carefully to hide the access delay of the global memory considering the input data size and the hardware features of our target GPUs (Pascal GPUs). For the multi-channel kernel, we propose a *stride-fixed block* method. This method aims to maximize the number of FMA operations per loaded data because the total amount of data that have to be loaded to each *SM* is much larger than the single-channel convolution. This allows the access delay to be hidden by *data prefetching*.

2 OPTIMIZATION METHODS

In this section, we first introduce the CNN models, and then discuss what kind of optimization method is applicable on Pascal GPUs.

2.1 The Convolution Models

The multi-channel convolution can be defined as follows:

$$\begin{aligned} O^m(x,y) &= \sum_{ch=1}^{C}\sum_{i=0}^{K-1}\sum_{j=0}^{K-1} I^{ch}(x+i,y+j)\cdot F^{ch,m}(i,j), \\ \textit{where} \quad & x\in[0,W_x-K+1), y\in[0,W_y-K+1), m\in[1,M]. \end{aligned} \tag{1}$$

Here, I is the input feature map set and F is the input filter set. O is the output feature map set which is generated from I and F. x and y are the coordinates of the pixels of the feature maps. W_x is the width and W_y is the height of the input feature map. i and j are the offsets, and are added to the coordinates. Their upper bound K determines the size of filter. ch represents the channel of the input in the range of $[1,C]$ (C is the number of channels), and all of the convolution results are added along the dimension ch. m represents the filter number, and each filter has C channels. M is the number of filters, which is defined in each convolution layer. When $C=1$, it is called single-channel convolution, and its definition is given the following equation.

$$O^m(x,y) = \sum_{i=0}^{K-1}\sum_{j=0}^{K-1} I(x+i,y+j)\cdot F^m(i,j) \tag{2}$$

2.2 Acceleration models on GPU

Here, we discuss the acceleration methods of the convolution calculation. However, this discussion is not restricted to the convolution, and can be applied to other applications.

In GPUs, the on-chip memory including registers and shared memory is supported, as well as the off-chip memory, in which the global memory is mainly supported. The register is the fastest, while the global memory is the slowest and largest. To fully utilize this hierarchy, many studies such as (Chen et al. 2017) have been proposed. However, throughput computation, the data loading time from the global memory to the on-chip memory is most critical, and hiding the latency of the global memory is the most crucial point for the acceleration.

To hide the latency of the global memory, two methods can be considered:

1. keep the operation units busy (mainly Fused Multiply-Add (FMA) operation units in convolution) by executing more than N_{FMA} operations (the lowest value to make the units busy) in each *SM* for the current data set until the next data set arrive from the global memory by data prefetching, and
2. transfer a large volume of data (V_s) from the global memory continuously.

In most cases, the first approach is preferable, because the data loading overhead from the global memory can be relatively reduced more by executing more operations per loaded data. In the multi-channel convolution, the data size is large enough, and it is possible to find the division of the feature maps and filters that makes it possible to execute more than N_{FMA} operations in each *SM*. However, in the single-channel convolution, when the size of feature maps is small, the number of executable operations becomes less than N_{FMA} even with the data prefetching, and the second approach is required. Thus, it is necessary to make it clear under what conditions which method shows better performance.

Table 1 shows several parameters of GTX 1080Ti and its performance for accessing single precision data. As shown in Table 1, in GTX 1080Ti, 2 FMA operations can be executed in one clock cycle in each core, namely 256 FMA operations in each *SM* (each *SM* has N_{cores} = 128 cores). According to the method proposed in (Mei and Chu 2017), the global memory latency of the GTX 1080Ti is 258 clock cycles. In order to hide these 258 clock cycles, $N_{FMA} = 66,048$ FMA operations ($66,048 = 258 \times N_{cores} \times 2$) are required in each *SM* for the current data set, the set of divided feature maps and filters.

The volume size V_s can be calculated as follows. The Geforce GTX 1080Ti has a base clock of 1480 MHz and the bandwidth of 484 GB/s, which means the transfer rate is roughly 327 bytes per clock cycle. Therefore, the volume size needed to hide the latency (258 clock cycles) becomes 84,366 = 327 $\times$ 258 bytes. To accommodate the data transfer of this size, 21,120 (= 84,366 / 4) threads are required because each thread fetches a 4 byte data in single precision. Thus, in each of 28 *SMs* ($N_{sm} = 28$ is the total number of *SMs* in the GTX 1080Ti), 755 threads are required to fetch one 4-byte word respectively. Here, because 32 threads in a warp execute the same instruction at the same time, the number of threads required is often kept at an integer multiple of 32. Therefore, instead of 755, 768 threads are preferred (in total, it becomes 768 $\times 4 \times 28$= 86,016 > 84,366). This means that the minimum volume size to make the global memory busy is $V_s = 86,016$ bytes. For dividing the feature maps and filters, and assigning them to each *SM*, the following procedure should be taken:

1. Divide the feature maps and filters so the total size of data that are assigned to each *SM* is smaller than the size of the shared memory S_{shared} (96KB in GTX 1080Ti).
2. Evaluate the number of FMA operations that can be executed for the data in each *SM*.
3. If it is larger than N_{FMA}, use the first method which is based on the data prefetching.
4. If not, redivide the feature maps and filters so that the total size of data that are transferred to all *SMs* become larger than V_s, and use the second approach.

Table 1: Parameters to access single precision data

	GTX 1080Ti
Architecture	Pascal
Global Memory Latency (clock cycles)	258
Bandwidth (GB/s)	484
Base clock cycle (MHz)	1480
SM	28
Transmission Rate (Byte/clock cycle)	327
Data Requirement (bytes)	84,366
Thread Requirement/SM	768
Warp Requirement/SM	24
Data Requirement/SM (bytes)	3072
Flops/clock cycle/core	2

Additionally, to access global memory, it is necessary to confirm that the starting address and the size of the sequential accessing segment is a multiple of *32-byte*. In Pascal GPU, although the performance of multiple of *128-byte* shows best (NVIDIA 2018), the performance for *32-byte* and *64-byte* are also acceptable.

2.3 Data Mapping

As shown in Fig.1(a), in the single-channel convolution ($C = 1$), the size of each filter is $K \times K \times 4\text{-}byte$, and they are continuously stored in the global memory. With this data mapping, the filters can be divided only along the dimension m, and the filters can be efficiently loaded from the global memory because they are stored continuously. Three approaches can be considered.

1. Only the filters are divided and are assigned to each *SM*. In each *SM*, the assigned filters are applied to the whole feature maps (the feature map is processed sequentially against each filter).
2. Only the feature maps are divided and are assigned to each *SM*. In each *SM*, the assigned feature maps are processed by all filters sequentially.
3. Both feature maps and filters are divided, and they are assigned to each *SM* (the combination of the first and the second approach).

By using different approach, the amount of data that has to be loaded to the shared memory from the global memory, and the number of FMA operations that can be executed in parallel become different. Therefore, finding a good balance between the size of divided feature maps and filters becomes a key point.

In the multi-channel convolution ($C > 1$), which is the typical case in the convolution layers of the CNN. The data size becomes much larger than the single-channel convolution. Fig.1(b) shows how the filters are stored in the global memory. They are stored along the dimension *ch* first, and then along the dimension m. In this case, the dividing method of the filters along the dimension m used for the single-channel convolution cannot be applied as is, because the data size of each filter is normally not a multiple of *32-byte*. Especially when $K = 1$, the filter size is only 4 bytes and are accessed as *4-byte* segments which causes serious performance reduction due to the *non-coalescing memory access*.To solve this problem, several approaches can be considered. Fig.2(a) shows the whole data structure before the division.

1. In Fig.2(b), both the filters and the feature maps are divided along the dimension *ch*, and the data for $C' = C/N_{sm}$ channels are assigned to each *SM* (N_{sm} is the total number of *SMs*). With this

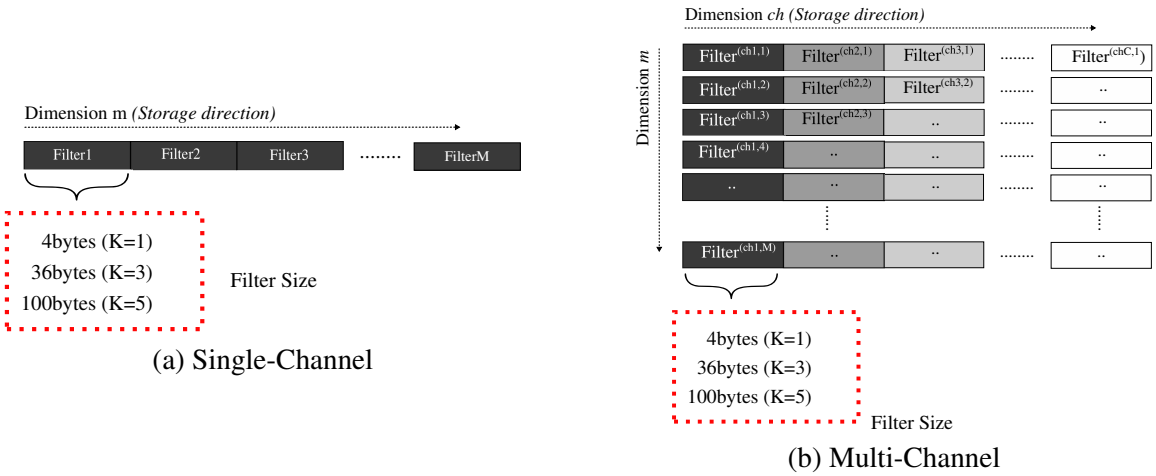


Figure 1: Memory storage form of the Filter

division, the data calculated in each thread have to be summed along dimension *ch*, meaning that add operations among *SMs* are required, and $W_x \times W_y \times C' \times 4\text{-}byte$ byte in the global memory are used for this summation. The global memory accesses to this area and the synchronous operations required for this summation considerably reduce the overall performance.

2. In Fig.2(c), only the filters are divided along the dimension m. $M' \times C = M/N_{sm} \times C$ filters are assigned to each *SM*, and the whole feature map is loaded to *SMs* from the global memory. In this approach, if the total size of the filters are less than the total size of shared memory ($K \times K \times C \times M \times 4\text{-}byte < N_{sm} \times S_{shared}$), the divided filters can be cached in each *SM*, and no additional access to the global memory is required.
3. On the other hand, in Fig.2(d), only the feature maps are divided along dimension y. The divided feature maps are assigned to each *SM*, and the whole of the filters are loaded to *SMs* from the global memory. In this case, if the total size of the feature maps is smaller than the total size of the shared memory, the divided feature maps can be cached in each *SM*, and no additional access to the global memory is required.
4. However, the total size of the filters and feature maps are larger than the total size of the shared memory in general. Thus, as shown in Fig.2(e), both the filters and feature maps have to be divided respectively, and each divided segments are cached in each *SM* or loaded from the global memory. In this case, there are many alternatives for how to divide the filters and features maps.

According to our preliminary evaluation, the performance with the data dividing method along the dimension *ch* (Fig.2(b)) is obviously slower than other dividing methods because of the additional access to the global memory for the addition. For achieving higher performance, it is necessary to choose other dividing methods considering the data size and the hardware features of the target GPUs so that in each *SM* the number of FMA operations that can be executed per loaded data from the global memory is maximized.

3 GPU IMPLEMENTATION

According to the discussion in Section 2, in both the single-channel and multi-channel convolution, it is important to make the number of FMA operations per pre-fetched data higher than N_{FMA} in order to improve performance. However, in some single-channel convolution cases, like when the size of feature maps is small, the number of FMA operations cannot be kept high enough by data prefetching. This means that, in case of single channel convolution, according to the size of input data, we need to choose one of the two methods described in Section 2.2: data prefetching or data transfer larger than V_s.

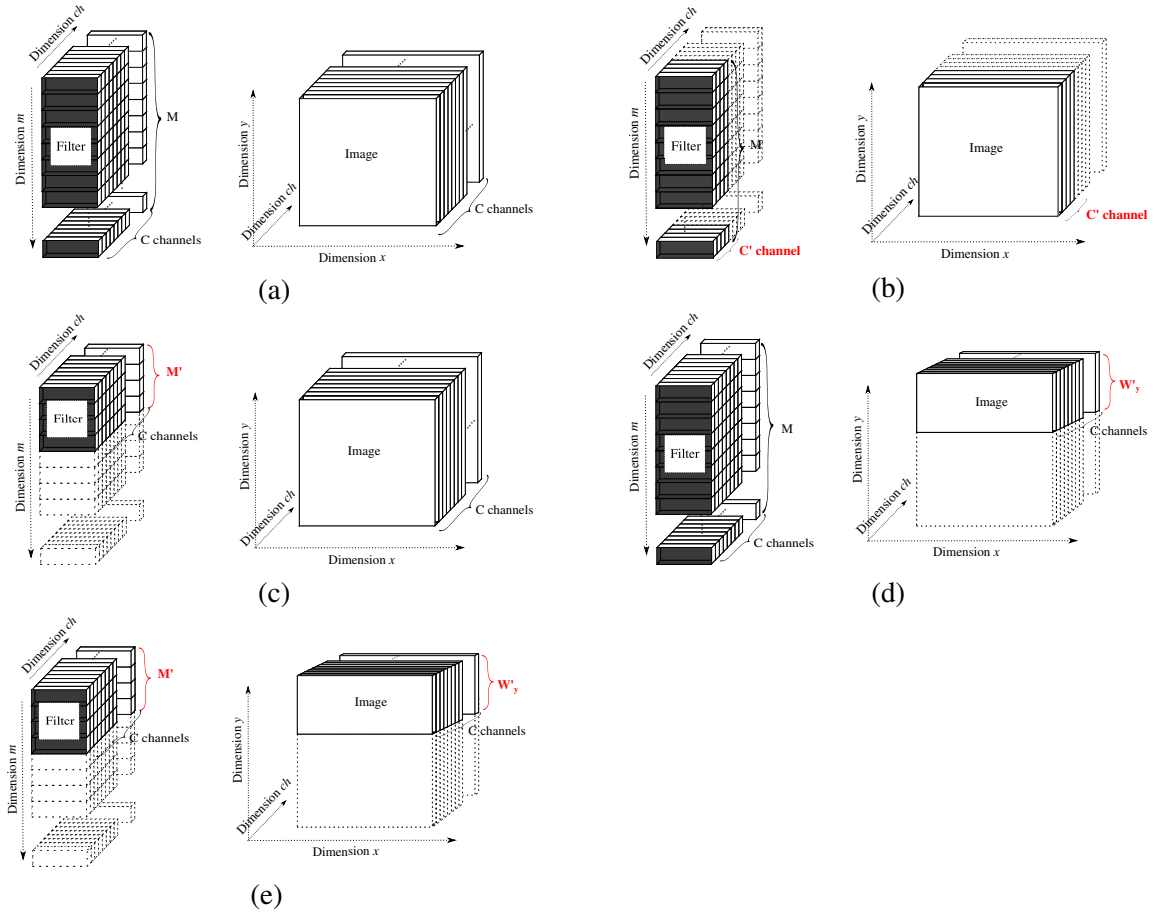


Figure 2: Assignment of the input data

In the multi-channel convolution, the size of input data is large enough, so the number of FMA operations can be kept high enough by data prefetching. However, the performance can be improved more by achieving higher FMA operation ratio for the fetched data, because to fetch the data from the global memory, each thread has to issue the instruction to read the data, and the clock cycles are spent for issuing these read instructions. Therefore, in the multi-channel convolution, to find the data dividing method that maximizes the number of FMA operations for each divided data is the key to achieve higher performance.

3.1 Single-Channel Convolution

Here, we describe how to divide the input data to improve the performance in the single channel convolution. As for the convolution calculation in each SM, we follow the method proposed in (Chen et al. 2017).

The total amount of the input data is given by:

$$D_{input} = D_{filter} + D_{map} = (K \times K \times M + W_x \times W_y) \times 4Bytes. \tag{3}$$

Let N_{sm} be the number of SMs. There are two ways to divide the input data and assign them to each SM. In the first method, the input data is divided along the dimension m of the filter. D_1, the size of input data assigned to each SM, becomes

$$D_1 = \frac{D_{filter}}{N_{sm}} + D_{map} = (K \times K \times \lceil \frac{M}{N_{sm}} \rceil + W_x \times W_y) \times 4Bytes \tag{4}$$

In general, D_{map} is too large to be stored in the on-chip memory of each SM. Thus, D_{map} is divided into P pieces along the dimension y. The size of each piece becomes $D_{Inc1} = D_{map}/P$. Here, for each line of feature map, since the convolution requires additional $K-1$ lines, the amount of data that have to be held in the on-chip memory becomes

$$D_1 = \frac{D_{filter}}{N_{sm}} + D_{Inc1} + (K-1) * W_x = (K \times K \times \lceil \frac{M}{N_{sm}} \rceil + (\lceil \frac{W_y}{P} \rceil + K - 1) \times W_x) \times 4Bytes \tag{5}$$

and the number of FMA operation that can be executed for these data in each SM is given by

$$Th_1 = \frac{D_{filter}}{N_{sm}} \times D_{Inc1} = K \times K \times \lceil \frac{M}{N_{sm}} \rceil \times \lceil \frac{W_y}{P} \rceil \times W_x. \tag{6}$$

In the second method, the input data is divided along the dimension y of the feature map. In this case, D_2, the amount of the input data assigned to each SM, becomes

$$D_2 = D_{filter} + \frac{D_{map}}{N_{sm}} = (K \times K \times M + (\lceil \frac{W_y}{N_{sm}} \rceil + K - 1) \times W_x) \times 4Bytes. \tag{7}$$

D_{filter} is too large to be stored in the on-chip memory in general, and it is divided into Q pieces. The size of each piece becomes $D_{Inc2} = D_{filter}/Q$. Then, D_2 becomes

$$D_2 = D_{Inc2} + \frac{D_{map}}{N_{sm}} = \frac{D_{filter}}{Q} + \frac{D_{map}}{N_{sm}} = (K \times K \times \lceil \frac{M}{Q} \rceil + (\lceil \frac{W_y}{N_{sm}} \rceil + K - 1) \times W_x) \times 4Bytes \tag{8}$$

and the number of FMA operation that can be executed for these data in each SM is given by

$$Th_2 = D_{Inc2} \times \frac{D_{map}}{N_{sm}} = K \times K \times \lceil \frac{M}{Q} \rceil \times (\lceil \frac{W_y}{N_{sm}} \rceil) \times W_x. \tag{9}$$

The values of P and Q are decided considering if D_1 or D_2 is smaller than S_{shared}, and if Th_1 or Th_2 is larger than N_{FMA}. If $P = 1$ or $Q = 1$, the feature maps or the filters are not divided, and they are transferred to the on-chip memory at a time. If $P > 1$ or $Q > 1$, the feature maps or the filters are divided into several pieces, and the pieces are transferred to each SM by using the data prefetching. With smaller P and Q, D_1, D_2 and Th_1, Th_2 become larger. The lower bound of P and Q is given by the requirement that D_1 and D_2 have to be smaller than S_{shared}, and the upper bound is given by the requirement that Th_1 and Th_2 should be larger than N_{FMA}. P and Q should be chosen so that these requirements can be satisfied. In our implementation, P and Q are decided as follows.

1. Th_1 or Th_2 should be larger than the number of FMA operation N_{FMA}.

   $Th_1 \geq N_{FMA}$ and $Th_2 \geq N_{FMA}$

   Thus, the upper bound of P and Q (they must be smaller than W_y and M respectively) is given as follows

   $P \leq \frac{K \times K \times \lceil \frac{M}{N_{sm}} \rceil \times W_y \times W_x}{N_{FMA}}$ *and* $P \leq W_y$, $\qquad Q \leq \frac{K \times K \times M \times \lceil \frac{W_y}{N_{sm}} \rceil \times W_x}{N_{FMA}}$ *and* $Q \leq M$

2. D_1 and D_2 must be smaller than the size of on-chip memory. The lower bound of P and Q is given as follows.

   $P \geq \frac{4 \times W_y \times W_x}{S_{shared} - 4 \times K \times K \times \lceil \frac{M}{N_{SM}} \rceil + (1-K) \times 4 \times W_x}$, $\qquad Q \geq \frac{4 \times M \times K \times K}{S_{shared} - 4 \times W_x \times (\lceil \frac{W_y}{N_{SM}} + K - 1)}$

   Actually, there exist one more requirement to decide this lower bound. The number of required registers for the computation must be smaller than that supported in each SM. Its detail is not shown here, but considering this requirement, the lower bound is calculated.

3. If there exist P and Q (P and Q must be an integer) in the range specified by (1) and (2), any of them can be used. In our current implementation, the minimum ones are chosen as P and Q, because the smaller values means less division, and make the processing sequence simpler. If no value exists, P and Q are set to 1.
4. Using the obtained P and Q, D_1 and D_2 are calculated and compared. If D_1 is smaller than D_2, Q is reset to 1 to use the first dividing method described above, and otherwise, P is reset to 1 to use the second one. Both methods can be used because they both satisfy the requirements, but for safety and leaving more memory space on the on-chip memory, the smaller one is chosen.

Following this procedure, the input data are divided and allocated to each *SM* in the best balance.

3.2 Multi-Channel Convolution

As described above, in the multi-channel convolution, both feature maps and filters are divided, and prefetching is used to transfer them to each *SM* from the global memory. Recent block-based methods show high performance in convolution due to their continuous and simple memory access sequence. Fig.3 shows the data mapping of the filters and feature maps, and how they are divided and calculated in each *SM*. In the block-based method, as shown in Fig.3(a), the following data are loaded to the on-chip memory in each *SM*.

1. S bytes of each filter along the dimension *ch* (called *segment* in the following discussion) of M' filters ($S \times M'$ bytes in total), and
2. a part of feature map, $W'_x \times W'_y \times 4\,bytes$ in the same channel (W'_x is an arbitrary value that is decided by the size of on-chip memory, but W'_y is specified as $\lceil \frac{S}{K \times 4bytes} \rceil$, because when S bytes are fetched along the dimension *ch*, $\lceil \frac{S}{K \times 4bytes} \rceil$ lines in the feature map are required to apply the filter).

Then, the convolution is calculated for these data, and the next data (next $S \times M'$ bytes of filters and $W'_y \times W'_x$ bytes of feature maps are loaded by data prefetching. In (Chen et al. 2017), the filter size is chosen as S ($S = K \times K \times 4\,bytes$), and only the filters of the target channel and a part of feature map of the same *channel* are loaded to the on-chip memory. However, the filter size $K \times K$ is usually odd and often small, and the performance is seriously degraded because of *non-coalescing memory access*. (Tan et al. 2011) tried to solve this problem by extending S to *128-bytes*. By fetching continuous 128 bytes on the global memory, the highest memory throughput can be achieved in GPUs. In this method, the filters of several channels (and a part of the next channel) are fetched at the same time, and are kept in the on-chip memory. First, only the filters of the first channel are used for the computation, then, the filters of the next channel are used. With this larger S, M' has to be kept small because of the limited size of on-chip memory, and smaller M' means less parallelism (M' filters are applied in parallel to the feature map of the same channel). In (Chen et al. 2017), higher parallelism comes first, while in (Tan et al. 2011), lower access delay has a higher priority.

Here, we propose a *stride-fixed block* method not only to maintain the efficient global memory access, but also to achieve high parallelism in each *SM*.

1. S is set to a multiple of 32-bytes. Actually, 32 or 64 is used. Small S allows larger M', namely higher parallelism, under the limited size of on-chip memory. When S is 32 or 64 bytes, the memory throughput from the global memory becomes a bit worse than $S = 128$ bytes (the highest throughput), but it is acceptable. $S = 32$ is the minimum value to maintain efficient global memory access.
2. Next, fixed W'_x. W'_x pixels in the feature map are fetched along dimension x from the global memory. Thus, W'_x should be a multiple of 128-bytes to achieve the highest memory throughput. Larger W'_x is preferable because it increases the Instruction Level Parallelism (ILP), which can improve the performance of the convolution.

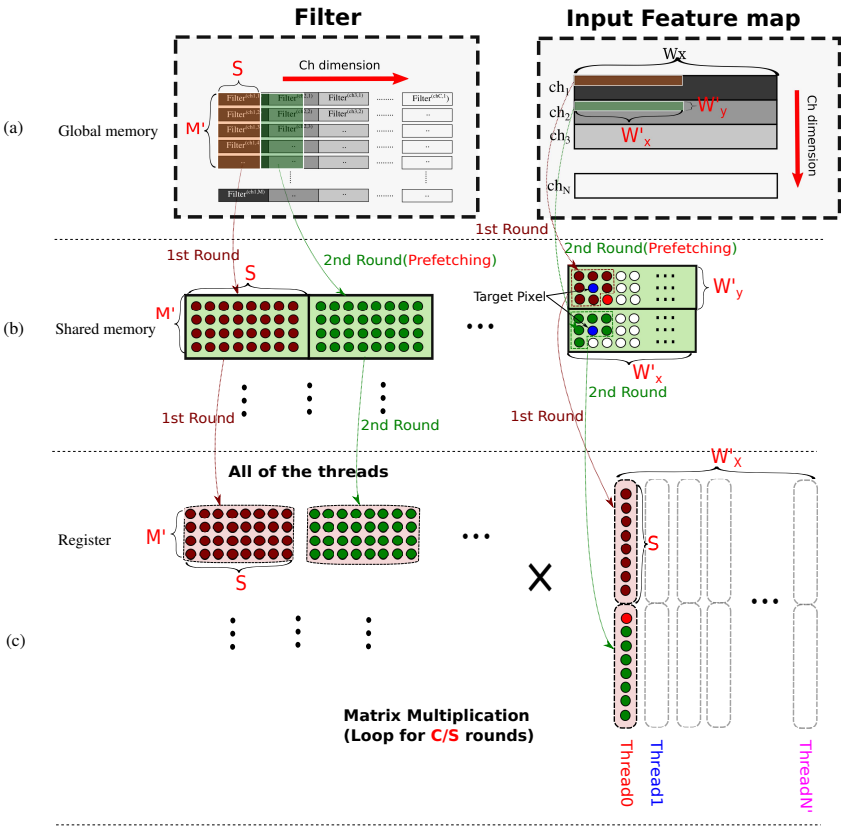


Figure 3: Multi-Channel Convolution Kernel

3. After deciding the values of S and W'_x, the most suitable M' can be found by the requirements of the number of FMA operations.

   $M' \geq \frac{N_{FMA} \times 4\text{-}bytes}{S \times W'_x}$.

4. Because the data prefetching is used to fetch the next data set while the current data set is being used for the current calculation, the size of the data set cannot exceed half of the shared memory. Thus, $(S \times M' + \lceil \frac{S}{K \times 4bytes} \rceil \times W'_x) \leq \frac{S_{shared}}{2}$

   Here, $\lceil \frac{S}{K \times 4bytes} \rceil = W'_y$ is the number of feature maps required for the calculation.

With this approach, for given S, W'_x and M' to improve performance based on *block method* can be obtained.

From here, we describe how the convolution calculation is executed in each SM. As shown in Fig.3(a)(b), first, each SM loads $S bytes$ of M' filters to the shared memory. At the same time, W'_x pixels on $\lceil \frac{S}{K \times 4bytes} \rceil = W'_y$ lines of the feature maps are also loaded. After the first round loading of these data, the same size of data for the next round are pre-fetched: the next $S \times M'$ bytes along the dimension ch, and the next W'_x pixels of the W'_y lines. During the second round loading, the convolutions for the first round data set are calculated on the chip as shown in Fig.3(b)(c). On the chip, each thread corresponds to one target pixel of the feature map as shown in Fig.3(b). Because the accessing speed of registers is faster than that of the shared memory, it is required to transfer each data in the shared memory to the registers in order to achieve high performance. In the convolution computation, the target pixel and its neighbors in the feature map are sent to the corresponding registers by each thread. Here, one important point is that only $S/4bytes$ pixels

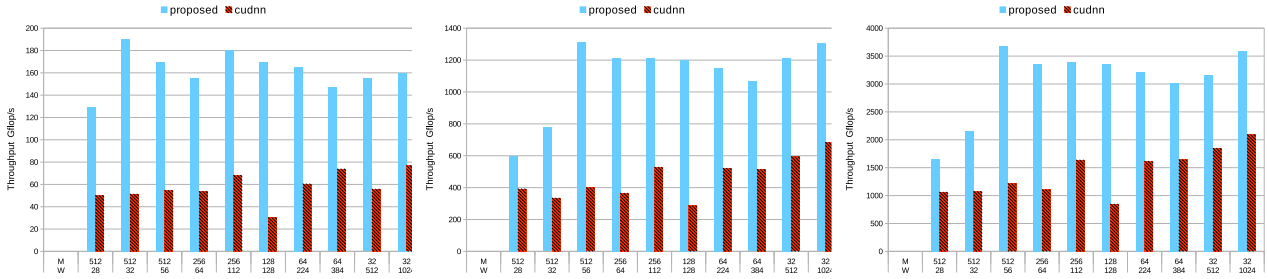
in the feature map have to be loaded onto the registers. The rest pixels, the red ones in Fig.3(b), are just held in the shared memory for the next round. The filter data is also transferred to the registers by the corresponding thread, but in this case, all data is transferred to the registers because all of them are used. After that, each pixel data is multiplied by the corresponding filter data, and their products are added. When all computations for the data stored in on-chip memory has been finished, data prefetching for the third round is started. During this loading, the convolution for the second round data is calculated.

By using this method, the size of S can be kept small, and the number of filters M' can be increased. This ensures that more filters can be applied in parallel to the same feature map. This does not increase the number of data loading of the feature maps, and hides the latency caused by global memory access.

4 EXPERIMENTAL ANALYSIS

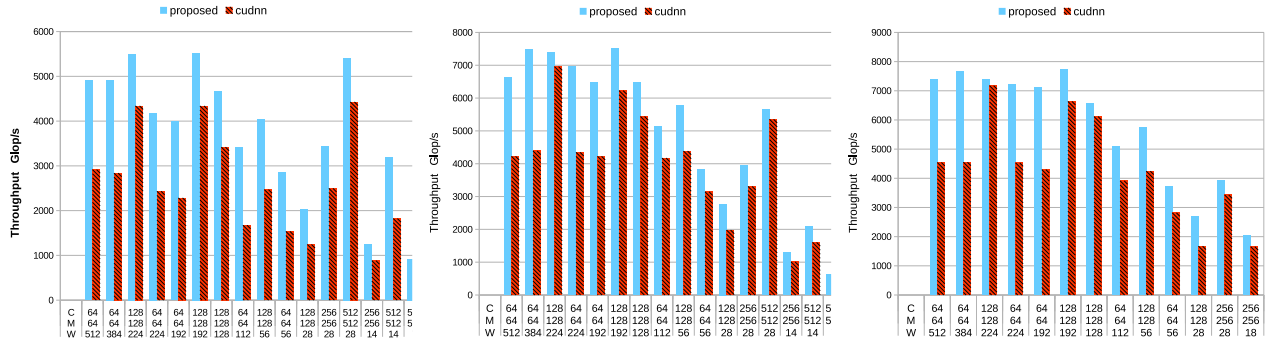
We implemented our two convolution kernels on Pascal series GPU Geforce GTX 1080Ti by using the CUDA 8.0. Their performances were evaluated using many convolutions which are commonly used in popular CNN models, and compared with the latest public library Cudnn v7.1 (Chetlur et al. 2014).

In the single-channel convolution, we changed the sample size of the feature maps from 28 to 1K and the size of the corresponding channels from 512 to 32. The filter size is 1, 3 or 5, which is common in many CNN models. In CUDA programming, by assigning more number of blocks to each SM, the SMs can be kept busy. In our current implementation, $N_{block} = 2 \times N_{SM} = 2 \times 28$ blocks are used. Two blocks are assigned to each SM, and 512 threads are assigned to each block. Thus, the maximum number of registers for each thread is constrained to 128. For each tested case, P and Q are decided following the method



Filter Size = 1 Filter Size = 3 Filter Size = 5

Figure 4: Performance of the Single-Channel Convolution Kernel



Filter Size = 1 Filter Size = 3 Filter Size = 5

Figure 5: Performance of the Multi-Channel Convolution Kernel

described in Section 3.1. Fig.4 shows the results of the single-channel convolution. Our method is faster than Cudnn v7.1 in all tested cases. The performance gain is 1.5X to 5.6X, and its average is 2.6X. In the multi-channel convolution, we changed the sample size of the feature maps from 7 to 512, and the size of the corresponding channels from 64 to 512. The filter size is also 1, 3, or 5. As discussed in Section 3.2, larger M' is preferable for making *data prefetching* more effective. Therefore, we fixed the segment size S as 32 or 64 bytes, and M' and W'_x are decided following the method described in Section 3.2. According to our preliminary evaluation, when $M' = 64$ and $W'_x = 128$, the performance is best, and we used these values for this comparison. As shown in Fig.5, our method is faster than Cudnn in all tested cases, and the throughput has been increased by 1.05X to 2X, with an average increase of 1.39X. In (Chen et al. 2017), a different GPU is used, and a direct comparison is not possible. However, when $K = 3$, our performance is 4X faster and the peak performance of which is 2.4X faster than that used in (Chen et al. 2017).

We also implemented our two kernels on Maxwell series GPU GTX Titan X, and it also showed that our performance is faster than Cudnn on the same GPU by 1.3X to 3.7X in the single-channel convolution and 1.08X to 1.8X in the multi-channel convolution.

5 CONCLUSIONS

In this paper, we proposed two convolution kernels on Pascal series GPUs for single-channel and multi-channel respectively. For single-channel convolution, we introduced an effective method of data mapping, which can hide the access delay of the global memory efficiently. For multi-channel convolution, we introduced a method that not only guarantees the memory access efficiency, but also achieves high FMA operation ratio per loaded data. Performance comparison with the public library Cudnn shows that our approaches are faster in all tested cases: 1.5X to 5.5X in the single-channel convolution and 1.05X to 2X in the multi-channel convolution. Our approaches was designed assuming Pascal architecture, but the performance is also faster than Cudnn on Maxwell architecture. This practice shows that our approaches can be applied to the wide range of CNN models on various GPUs. In our current implementation, the throughput is still lower than the theoretical maximum, meaning that the convolution kernel still has room for improvement, which will be our main future work.

ACKNOWLEDGMENTS

This paper is based on results obtained from a project commissioned by the New Energy and Industrial Technology Development Organization (NEDO).

REFERENCES

Chen, X., J. Chen, D. Z. Chen, and X. S. Hu. 2017. "Optimizing Memory Efficiency for Convolution Kernels on Kepler GPUs". In *54th Annual Design Automation Conference, DAC*, pp. 68:1–68:6.

Chetlur, S., C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. 2014. "cuDNN: Efficient Primitives for Deep Learning". *CoRR* vol. abs/1410.0759.

He, K., X. Zhang, S. Ren, and J. Sun. 2016. "Deep Residual Learning for Image Recognition". In *IEEE Conference on Computer Vision and Pattern Recognition,CVPR*, pp. 770–778.

Jia, Y., E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. B. Girshick, S. Guadarrama, and T. Darrell. 2014. "Caffe: Convolutional Architecture for Fast Feature Embedding". In *ACM International Conference on Multimedia, MM*, pp. 675–678.

Krizhevsky, A., I. Sutskever, and G. E. Hinton. 2017. "ImageNet classification with deep convolutional neural networks". *Commun. ACM* vol. 60 (6), pp. 84–90.

Lavin, A., and S. Gray. 2016. "Fast Algorithms for Convolutional Neural Networks". In *IEEE Conference on Computer Vision and Pattern Recognition,CVPR*, pp. 4013–4021.

Li, S., Y. Zhang, C. Xiang, and L. Shi. 2015. "Fast Convolution Operations on Many-Core Architectures". In *17th IEEE International Conference on High Performance Computing and Communications, HPCC*, pp. 316–323.

Mathieu, M., M. Henaff, and Y. LeCun. 2013. "Fast Training of Convolutional Networks through FFTs". *CoRR* vol. abs/1312.5851.

Mei, X., and X. Chu. 2017. "Dissecting GPU Memory Hierarchy Through Microbenchmarking". *IEEE Trans. Parallel Distrib. Syst.* vol. 28 (1), pp. 72–86.

NVIDIA 2018. "CUDA C Programming Guide V9.1". Http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

Poznanski, A., and L. Wolf. 2016. "CNN-N-Gram for HandwritingWord Recognition". In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pp. 2305–2314.

Simonyan, K., and A. Zisserman. 2014. "Very Deep Convolutional Networks for Large-Scale Image Recognition". *CoRR* vol. abs/1409.1556.

Szegedy, C., W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. 2015. "Going deeper with convolutions". In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pp. 1–9.

Tan, G., L. Li, S. Triechle, E. H. Phillips, Y. Bao, and N. Sun. 2011. "Fast implementation of DGEMM on Fermi GPU". In *Conference on High Performance Computing Networking, Storage and Analysis, SC*, pp. 35:1–35:11.

Tokui, S., K. Oono, S. Hido, and J. Clayton. 2015. "Chainer: a Next-Generation Open Source Framework for Deep Learning". In *Workshop on Machine Learning Systems (LearningSys), NIPS*.

Yu, D., W. Xiong, J. Droppo, A. Stolcke, G. Ye, J. Li, and G. Zweig. 2016. "Deep Convolutional Neural Networks with Layer-Wise Context Expansion and Attention". In *Interspeech 2016, 17th Annual Conference of the International Speech Communication Association*, pp. 17–21.

Zhang, Y., and B. C. Wallace. 2017. "A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification". In *8th International Joint Conference on Natural Language Processing, Volume 1: Long Papers*, pp. 253–263.

AUTHOR BIOGRAPHIES

QIONG CHANG received his Master degree in Engineering from the University of Tsukuba in 2013. He is currently a PhD student at the Graduate School of Systems and Information Engineering, University of Tsukuba. His research interest is in reconfigurable parallel computing systems. His email address is cq@darwin.esys.tsukuba.ac.jp.

MASAKI ONISHI recevied his PhD from the Osaka Prefecture University in 2002. He was a research scientist at the National Institute of Advanced Industrial Science and Tehnology (AIST). His research interests are computer vision, video surveillance, and human robot interaction. His email address is onishi@ni.aist.go.jp.

TSUTOMU MARUYAMA received his PhD in Engineering from the University of Tokyo in 1987. He is currently a professor at the Graduate School of Systems and Information Engineering, University of Tsukuba. His research interest is in reconfigurable parallel computing systems. His email address is maruyama@darwin.esys.tsukuba.ac.jp.