Enabling Efficient Fast Convolution Algorithms on GPUs via MegaKernels

Liancheng Jia, Yun Liang, *Senior Member, IEEE*, Xiuhong Li, Liqiang Lu, and Shengen Yan

Abstract—Modern Convolutional Neural Networks (CNNs) require a massive amount of convolution operations. To address the overwhelming computation problem, Winograd and FFT fast algorithms have been used as effective approaches to reduce the number of multiplications. Inputs and filters are transformed into special domains then perform element-wise multiplication, which can be transformed into batched GEMM operation. Different stages of computation contain multiple tasks with different computation and memory behaviors, and they share intermediate data, which provides the opportunity to fuse these tasks into a monolithic kernel. But traditional kernel fusion suffers from the problem of insufficient shared memory, which limits the performance. In this article, we propose a new kernel fusion technique for fast convolution algorithms based on MegaKernel. GPU thread blocks are assigned with different computation tasks and we design a mapping algorithm to assign tasks to thread blocks. We build a scheduler which fetches and executes the tasks following the dependency relationship. Evaluation of modern CNNs shows that our techniques achieve an average of 1.25X and 1.7X speedup compared to cuDNN's two implementations on Winograd convolution algorithm.

1 Introduction

Convolutional neural networks (CNNs) are the state-of-the-art solution of image classification, detection and many other computer vision tasks [1], [2], [3]. The high accuracy of CNNs comes at the cost of huge computational complexity and tremendous connections in the convolutional layer. In a typical convolutional network, the input feature maps convolve with many weight filters iteratively to yield the output feature maps. For state-of-the-art CNN networks, convolution layers are the most time-consuming part. For example, the convolutional layer occupies more than 90 percent of the total computation in many popular neural networks [4]. Hardware accelerators such as GPUs, FPGAs, and ASICs have been employed to deal with the overwhelming computation pressure [5], [6], [7].

A universal approach to calculate the convolution operation is to apply general-purpose matrix multiplication (GEMM) by flattening features and filters to matrices. One effective alternative is to introduce FFT and Winograd fast algorithms [8], [9] on specific layers to reduce the arithmetic complexity. The fast convolution algorithms contain four stages. First, both input and filter tensors are split into tiles and apply Winograd/FFT transformation respectively. Then each transformed input and filter tiles perform element-wise matrix multiplication (EWMM) operation, which can be converted into batched GEMM operation. The computation amount of GEMM transformed tiles is significantly smaller than the original GEMM-based convolution. Finally, inverse transformation is applied to result matrices to produce the convolution result. Prior work [8] shows that using Winograd algorithm can reduce the number of multiplications more than twice on VGG network, leading to an average speedup of 3.4X in CNN inference.

The conventional GPU implementation of the fast convolution algorithm runs each stage in one GPU kernel. The whole convolution is implemented in four kernels: input and filter transformation, GEMM and output transformation. Since different GPU kernels have different arithmetic intensity and they share intermediate data, kernel fusion potentially provides the opportunity of further speedup. When kernels of different stages are fused into one, GEMM and transformation tasks are able to run in parallel, which can increase resource and bandwidth utilization, therefore, enhance the performance. It's also possible to enable on-chip data reuse without transferring data back and forth with off-chip memory.

cuDNN [10] is the most renowned GPU deep learning library that implements Winograd/FFT fast convolution algorithm. It is used by most deep learning frameworks for efficient convolution processing on GPUs. It supports both fused and non-fused implementation for Winograd convolution. However, its implementation of the fused algorithm is still rudimentary. We observe that the overall performance of the fused kernel can be even slower compared to the non-fused version in many convolution layers. For instance, the fused implementation is 1.6X slower than the non-fused version using the cuDNN library for the *conv*3 layer of YOLOv3.

In this paper, we propose a new kernel fusion technique for Winograd convolution algorithms on GPUs. Different from typical kernel fusion which executes each stage one by one in the kernel function, we assign each thread block with different types of tasks from original kernels. To produce correct results, execution of tasks for every partition of data must

- *Liancheng Jia, Yun Liang, Xiuhong Li, and Liqiang Lu are with Center for Energy-efficient Computing and Applications, Peking University, Beijing 100871, China. E-mail: {jlc, ericlyun, lixiuhong, liqianglu}@pku.edu.cn.*
- *Shengen Yan is with Sensetime Group, Hong Kong, China. E-mail: yanshengen@sensetime.com.*

Manuscript received 15 Oct. 2019; revised 29 Jan. 2020; accepted 2 Feb. 2020. Date of publication 21 Feb. 2020; date of current version 9 June 2020. (Corresponding author: Yun Liang.)
Digital Object Identifier no. 10.1109/TC.2020.2973144

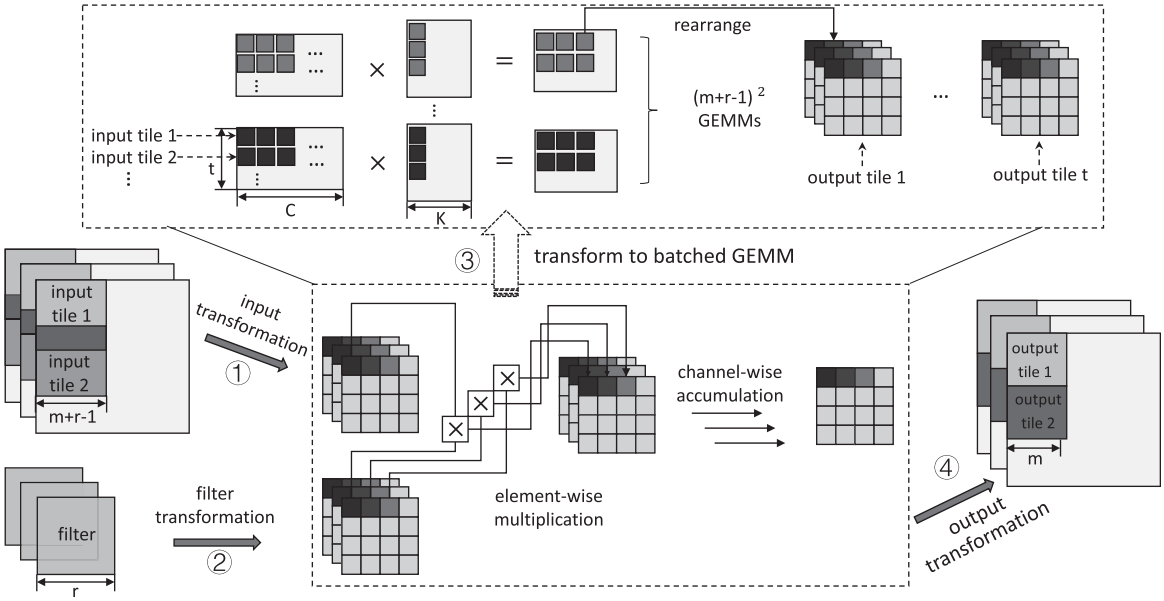


Fig. 1. The computation flow of fast convolution algorithms.

follow the order of *input and filter transformation* → *GEMM* → *output transformation*, which causes dependency between tasks. We develop a task mapping algorithm that maps the tasks from different kernels to the thread blocks in the fused kernel. We optimize for several factors including task dependency, resource balancing, and data reusing to gain the performance speedup. The dependency between tasks can be resolved with little overhead with the proposed task scheduler.

In conclusion, the contribution of this paper includes:

- We propose a novel kernel fusion technique for Winograd convolution algorithm on GPUs based on megakernel.
- We develop an efficient task mapping algorithm based on comprehensive analysis of task dependency, resource balancing and data reusing of the fused kernel.
- We develop a task scheduler for the execution of fused kernel. It handles the dependency between tasks with little overhead.

We evaluate our kernel fusion technique on NVIDIA Tesla V100 GPUs with four modern CNNs. For 3 x 3 convolution layers in VGG-16, ResNet-50, YOLO-v3, and DenseNet-161, our technique achieves an average speedup of 1.25X and 1.7X over cuDNN non-fused and fused implementation of Winograd convolution, respectively, and 1.13X faster than the fastest cuDNN algorithm.

2 Background

In this section, we first introduce the computation flow of fast convolution algorithms and then present how the computation kernel is implemented on GPUs.

2.1 Winograd Fast Convolution Algorithm

Consider a normal 2D convolution layer, we have an $H \times W$ input image I and a $r \times r$ filter F with stride 1. Convolution operation generates the output O of size $(H - r + 1) \times (W - r + 1)$ with the following formula:

$$O_{x,y} = \sum_{a=1}^{r} \sum_{b=1}^{r} F_{a,b} I_{x+a,y+b}. \tag{1}$$

In practice, there are often multiple input channels and multiple filters. Images are also batched into groups for higher performance. For convolution operation with C input channels, K filters, and images are batched with size N, the output tensor O is given by the following formula. Here j is the index of the image in the batch, and k is the index of the filter.

$$O_{j,k,x,y} = \sum_{c=1}^{C} \sum_{a=1}^{r} \sum_{b=1}^{r} I_{j,c,a,b} * F_{k,c,x+a,y+b}. \tag{2}$$

Two representative fast algorithms to accelerate the spatial convolution process are Winograd and FFT. The fast algorithms can generate a tile of output feature map instead of computing them individually. Fig. 1 illustrates the convolution process of a single tile in following four stages:

- *Input Transformation (ITrans).* First, input tiles are transformed into size $(m + r - 1) \times (m + r - 1)$, with $(r - 1)$ rows of overlapping elements between neighbouring tiles.
- *Filter Transformation (FTrans).* Filter is also transformed to the same size $(m + r - 1) \times (m + r - 1)$ as transformed input tile.
- *Element-Wise Matrix Multiplication.* Element-wise multiplication-and-add are performed for the transformed tiles. In current GPU implementation, element-wise multiplication is batched into GEMMs for better parallelism. In Fig. 1, every t tiles are reorgnaized into $(m + r - 1)^2$ matrices, and they process GEMMs respectively.
- *Output Transformation (OTrans).* After doing the Element-wise multiplication, the result applies output transformation to generate the $m \times m$ convolution result. There is no overlap tile in the output tensor.

Given an input tile of size $(m+r-1)\times(m+r-1)$ and filter of size $r\times r$, the fast algorithm is applied to generate the output feature map of size $m\times m$,which can be represented as the following formula:

$$O=\sum_{j=0}^{C} OTrans[ITrans(I_j)\odot FTrans(F_j)], \tag{3}$$

where $\odot$ refers to the EWMM operation. I, O, F refers to input tile, output tile and filter, respectively. r is filter size and m is output tile size. *ITrans*, *FTrans*, *OTrans* refer to the special transformation functions for input feature map, filter and output feature map.

We apply the Winograd transformation function into formula (3), and the 1-D winograd convolution algorithm [8] can be described as follow:

$$O=A^T[(G\times I)\odot(B^T\times F)]. \tag{4}$$

Winograd transformations are linear transformations which can be expressed by matrix multiplication. A, B, G are fixed matrices which are irrelevant to input data. Common neural networks on computer vision use two-dimensional convolution, it can be computed by nesting one-dimensional convolution. The formula is as follow:

$$O=A^T[(GIG^T)\odot(B^TFB)]A. \tag{5}$$

The element-wise matrix-matrix multiplication only needs $(m+r-1)\times(m+r-1)$ multiplications. Compared to conventional GEMM solution, which needs $m\times m\times r\times r$ multiplication, Winograd algorithm can reduce arithmetic complexity by

$$\frac{(m+r-1)\times(m+r-1)}{m\times m\times r\times r}. \tag{6}$$

However, when m gets larger, the cost of transformation process grows rapidly, and the numerical accuracy of Winograd algorithm also decreases. In practical, $m=4$ is the most commonly used version.

2.2 Converting EWMM to GEMM

Element-wise matrix multiplication has low compute intensity, since one multiply-add operation needs three memory accesses (load two operands, and store the result), and GPU can't perform well to this kind of computation workload. EWMM can be transformed into batched GEMMs by transposition and reorgnaization, which will be efficiently executed on GPUs.

As shown in Fig. 1, the input transformation generates t tiles. Each tile's size is $(m+r-1)\times(m+r-1)$ and it has C channels. It can be transposed to a $((m+r-1)\times(m+r-1))\times C$ matrix. Similarly, we apply transposition to the filter, and it becomes a $((m+r-1)\times(m+r-1))\times C\times K$ tensor. The convolution of a tile then becomes batched vector-matrix multiplication (GEMV) with vector size C, matrix size $C\times K$ and batch size $(m+r-1)\times(m+r-1)$. Since we have t tiles, the GEMV operation is further combined into batched GEMM of $(t\times C)\times(C\times K)$, and the batch size is $(m+r-1)\times(m+r-1)$.

TABLE 1
Arithmetic Intensity(AI) for Different Stages of Convolution (FP32 Data Type)

Kernels	IN trans	GEMM	Out trans
Compute	14.18NHWC	9NHW(C+K)	11.6NHWK
Memory	13NHWC	4.5NHWCK	13NHWK
AI	1.08	$16\sim128$	0.89

2.3 Workload Partition for GPUs

To implement the fast convolution algorithm on GPUs, each stage is processed in one GPU kernel. For the transformation kernels, one GPU thread performs the transformation of one tile in one channel, and one TB performs the transformation on $blockDim$ channels of the same tile, where $blockDim$ is the number of thread in one TB. If the tile size is smaller than $blockDim$, the TB continues to process the next tile. For GEMM tasks, one or several TBs perform the multiplication of a $t\times C$ transformed input matrix and a $C\times K$ transformed filter matrix, and each thread computes one submatrix with 32 to 128 elements.

3 Motivation

We can observe two important benefits of kernel fusion for fast convolution algorithms: balanced resource usage and reduction of data movement. However, although GPU deep learning library cuDNN supports both fused and non-fused implementation for Winograd algorithm, the fused version often underperforms the non-fused version. For cuDNN library, the fused version is only faster than non-fused when channel sizes are less than 128. This motivates our work which targets a better kernel fusion technique. Here we introduce three motivations for our proposed method.

Balanced Computation and Memory Resource. As discussed before, Winograd/FFT convolution is composed of four kernels and they exhibit diverse computation and memory behaviors. We use arithmetic intensity (AI) to model the computation and memory transaction amount.

$$arithmetic\ intensity=\frac{compute\ instructions}{global\ memory\ transaction\ size}. \tag{7}$$

We measured the number of computation and memory transactions of Winograd convolution kernels and present them in Table 1. The balanced AI for Tesla V100 is close to 16 (Computation speed:15TFLOP/s, Memory bandwidth: 900 GB/s). If AI is greater than 16, the performance is likely to be limited by computation resources, and vice versa. Thus, the performance of the transformation kernels is limited by memory bandwidth, and the GEMM kernel is limited by computation resources. The different resource requirement of each kernel motivates the benefits of kernel fusion. Kernel fusion reorganizes the workload from multiple kernels to run in one kernel. It enables different types of workloads to run concurrently on GPU. When compute-intensive and memory-intensive workloads are running together, the arithmetic intensity will be balanced.

Reduction of Data Movement. In non-fused kernels, intermediate results (transformed input, filter and output tensors) are transferred back and forth within GPU registers and global memory. For example, transformed input data generated by input transformation kernel is stored to global memory, and loaded again by GEMM kernel. Therefore, kernel fusion can be applied to reduce some of the unnecessary data movement. Data reuse can be achieved via the on-chip shared memory, which reduces global memory transactions.

Insufficient Shared Memory. The existing fused implementation suffers from a huge disadvantage which hurts the performance greatly. Ideally, intermediate results can be stored in the shared memory of each thread block (TB) instead of global memory. However, the intermediate results are often too large to fit in the shared memory. Consider a convolution layer with 128 input and output channel for 16 6×6 tiles(the number of tiles must be big enough for efficient matrix multiplication). The workload will be transformed into $(16 \times 128) \times (128 \times 128)$ matrix multiplication with batch size of 36. If we compute Input Transformation, GEMM, and Output Transformation stages in one TB, the whole transformed tile should be stored in shared memory. The transformed input and output tile has $128 \times 16 \times 36$ numbers, which requires 288 KB shared memory. The weight coefficients also consume some shared memory capacity. Since current GPUs only have 48 to 96 KB shared memory in one SM, the capacity is far from enough. To solve the problem, the intermediate data has to either recomputed or stored into global memory, which introduces great overhead for large layers. We profiled fused and non-fused cuDNN Winograd implementations with NVIDIA profiler and found that the number of floating-point instructions in the fused kernel is 1.5 times of the non-fused kernel. It is possibly due to the recomputation of intermediate data. As a result, the advantage of kernel fusion cannot make up the disadvantage, and the performance becomes slower.

In this paper, we propose a new kernel fusion technique for fast convolution algorithms which avoids the problem of memory capacity while keeping the advantages of kernel fusion. In the following of this paper, Section 4 introduces our kernel fusion technique based on heterogeneous megakernels. Performance optimization of the fused kernel depends on an task mapping algorithm, which is analyzed in Section 5. The implementation of task scheduler is explained in Section 6. We evaluate our technique and compare with existing solutions in Section 7.

4 MegaKernel-Based Fusion for Convolution

We use Megakernel-based approach [11], [12] to effectively fuse the kernels of convolution algorithm. Listing 1 shows the difference of the proposed megakernel-based fusion between the conventional sequential fusion. Unlike the sequential fusion which assigns each stage of convolution in the same TB and executes them one by one(Listing 1(a)), our fusion technique schedules the computation of different stages to different TBs (Listing 1(b)). Each TB switches to different device function according to its block index and only processes the computation of a single stage.

Listing 1. Pseudo Code of Two Fusion Methods

```
kernel_a(){
    stage1();

    stage2();

    stage3();
}
```

(a) Sequntial Fusion

```
kernel_b(){
  item=task_map[blockIdx];
    switch (item.type){
      case 1:
        stage1();break;
      case 2:
        stage2();break;
      case 3:
        stage3();break;
  }
}
```

(b) MegaKernel

4.1 Problems and Solution of MegaKernel Fusion

However, the advantages of kernel fusion are also destroyed in the megakernel-based execution model. First, across-stage data reuse inside TB is not preserved. Each TB can only execute one stage, and different stages are executed in different TBs which must be transferred with global memory. Second, resource balancing is not guaranteed. GPU resource can be imbalanced when TBs of same stages execute together. If all concurrent running TBs are performing transformation stages, the memory bandwidth will be exhausted while some computation resources are idle. Moreover, there exists data dependency between stages, which must be preserved in the fused kernel. For example, for each partition of data, GEMM can only start after the corresponding inputs and filters are transformed.

The solution of these problems is inspired from the behaviour of NVIDIA GPU thread block scheduler. Although not stated in the official documents, researchers observed that the execution order of TBs is positivly correlated to the TB index (blockIdx) [13], [14]. Based on this observation, we find that the time interval of two TBs is related to their blockIdx distance. The transaction speed of L2 cache is much faster than off-chip memory. Since the global memory transaction is cached with L2, if the time interval of two TBs is small, they have more chance to share data with L2 cache. This solves the first problem of data reusing. For the resource balancing problem, it can be optimized by running types of workloads together, and can also be controlled with the index of TBs. For the dependency problem, the dependency between stages becomes the dependency between TBs. In the fused megakernel, we must preserve the execution order of particular TBs.

4.2 Proposed Design

Fig. 2 shows our solution of kernel fusion that aims to solve the problems. It consists of two parts: (a) a static task mapping algorithm and (b) a task scheduler. We define *task* as each TBs in original non-fused kernels. Task mapping is the process to map each task to a TB of fused kernel specified by TB index. The mapping process is invoked before kernel launching. With the task mapping process, we can optimize for L2 cache data reusing and GPU resource balancing by adjusting the tasks assigned to each TB. The task scheduler is used to execute the task and ensure the tasks are executed without unresolved dependency.

Fig. 2b shows the brief structure of the task scheduler. The execution of every TB consists of four steps: obtain task,

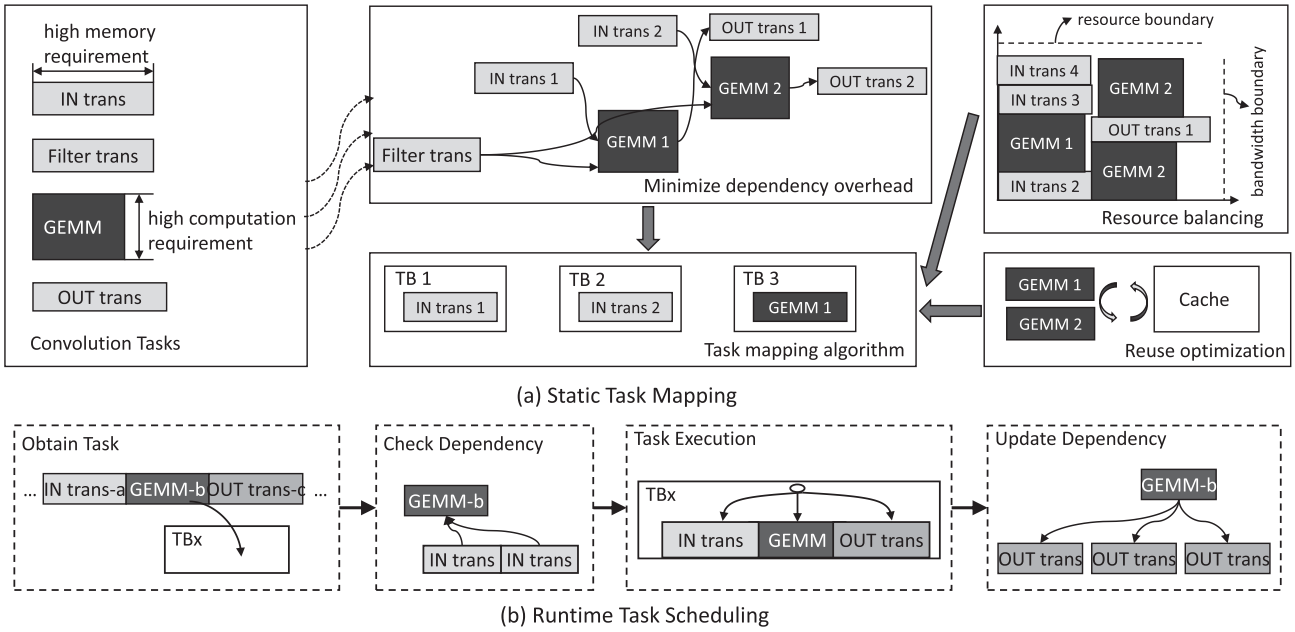


Fig. 2. Overview of kernel fusion.

check dependency, execute tasks and update task dependency. Tasks are pre-assigned to TBs in the task mapping step, so TBs can just access the mapping array to obtain its task. It checks its dependency relationship before execution. After making sure that the dependency is resolved, the TB switches to its device functions to perform the actual calculation. Finally, it updates the dependency status for other tasks. The detail of task scheduler will be explained in Section 6.

The advantage of the task scheduling system comes from three aspects. First, the scheduling overhead is very small. It only requires one extra memory transaction (read the task mapping array) and two atomic operations (check and update dependency). Second, it's very flexible to adjust to different task mapping solutions. Several tasks can run concurrently if they are assigned with adjacent TBs, and vice versa. Finally, static task mapping does not affect the load balancing among GPU streaming multiprocessor(SM)s, because TBs will be scheduled to GPUs with GPU's internal TB scheduler once there is free space on any SM.

5 Task Mapping for Megakernel

The key problem of Megakernel fusion is to map the tasks from the original kernels to one specific TB in the fused kernel. We define it as a *task mapping* problem. In this section, we analyze the effect of different factors for performance, and present a task mapping solution for best timing performance. Fig. 2a shows our task mapping optimization to minimize the overall execution time of the fused kernel. We first formulate the task mapping problem in Section 5.1. To balance the computation and memory utilization, we map the tasks with different arithmetic intensity adjacently to make them running concurrently on GPU (Section 5.2). Then, we model the distance between the dependent tasks to minimize the dependency overhead (Section 5.3). We try to maximize the data reuse by mapping tasks with data reuse to adjacent TBs, which reduces global memory transactions (Section 5.4). Finally, we use an algorithm to generate the task mapping sequence by combining all the optimization techniques (Section 5.5).

5.1 The Task Mapping Problem

As shown in Fig. 1, the execution of Winograd convolution algorithm can be separated into many groups, and each group is formed with t tiles. So the total number of group is $N_G = Batch\_Size \times Tiles\_per\_batch/t$. Inside each group, there are three types of task dependency orders which must be preserved: input transformation → GEMM, filter transformation → GEMM and GEMM → output transformation.

Definition 1 Task Group. *We first define* task *as the thread block from four original kernels. Apparently there are four types of task according to their original kernel. We define* task group *as a set of tasks of the same type to process each stage of convolution for t tiles. The convolution of t tiles involves different number of TBs from each original kernels. We define the number of tasks in each task group as S_I, S_G, S_O for input transformation, GEMM and output transformation, respectively. Filter transformation is used by every partition, so it is considered as a whole task group. The number of tasks for filter transformation is defined as N_F, same as the number of TBs in filter transformation kernel. The number of input transformation, GEMM, output transformation task group is same, which is defined as N_G.*

An input transformation task group is composed of tasks on transforming t tiles, which generates matrices for GEMM tasks. A GEMM task group is composed of the multiplication of $(m+r-1)^2$ matrices. The output transformation task group transforms the same number of tiles as the input transformation task group, but the number of tasks can be different because of their different channel numbers.

Definition 2 Task Mapping. *Given tasks from the four original kernels, namely $IN\_trans = \{I_{1,1} \dots I_{S_I,N_G}\}$ for input transformation, $Filter\_trans = \{F_1 \dots F_{N_F}\}$ for filter transformation,*

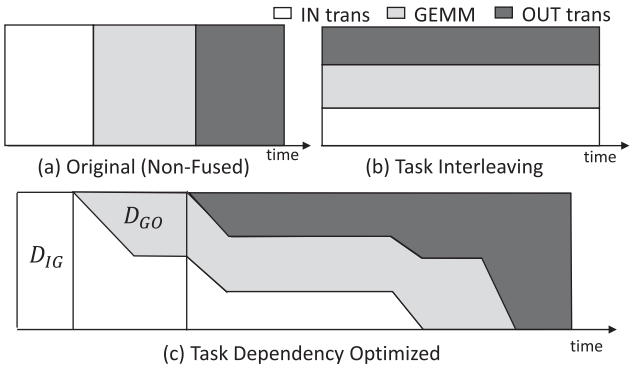


Fig. 3. Working task distribution in time.

$GEMM = \{G_{1,1} \dots G_{S_G,N_G}\}$ for GEMM, and $OUT\_trans = \{O_{1,1} \dots O_{S_O,N_G}\}$ for output transformation, and $N_F + N_G(S_I + S_G + S_O)$ TBs, task mapping is defined as the mapping f that maps TBs to tasks as follows,

$$f(TB) \rightarrow Task,$$

where

$$Task \in \{IN\_trans \cup Filter\_trans \cup GEMM \cup OUT\_trans\},$$

$$TB \in [0, N_F + N_G(S_I + S_G + S_O)).$$

Our goal is to find a task mapping strategy to minimize the execution time of the fused kernel. Since the number of TBs that can execute concurrently on GPU is fixed in compile time (we refer it as N_{CTB}), we try to minimize *the total execution time of all TBs* in the fused kernel. Since filter transformation is required by every GEMM tasks, we assign them to the beginning TBs (smallest blockIdx) and assume that filter transformation completes in advance. We only consider input/output transformation and GEMM tasks in the following discussion.

Fig. 3a shows the distribution of task types in fused kernel before optimization. Each type of tasks executes in order, which is the same as non-fused kernel. We optimize the task mapping by changing the TB index of each task in order to change their execution order.

5.2 Task Interleaving for Resource Balancing

One of the goal of kernel fusion is to improve resource balancing by the concurrent execution of different types of tasks. Transformation tasks are memory-intensive and GEMM tasks are often compute-intensive. When they run separately, one resource is used up while the other resource is not fully utilized. As discussed in Section 3, suppose the number of computation instruction is C_T for transformation tasks and C_G for GEMM tasks. The number of memory transaction are M_T and M_G respectively. The execution speed for computation and memory transaction is S_C and S_M for GPU. Without kernel fusion, the total time for two types of tasks can be estimated as

$$T_{Nonfused} = \frac{M_T}{S_M} + \frac{C_G}{S_C}. \tag{8}$$

After kernel fusion, the time of fused kernel is bounded by either computation or memory, so the total time can be estimated as

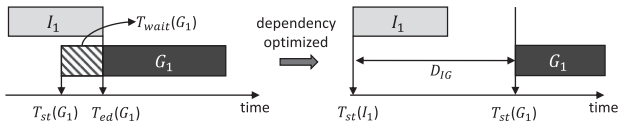


Fig. 4. Resolving dependency overhead.

$$T_{fused} = max\left(\frac{C_T + C_G}{S_C}, \frac{M_T + M_G}{S_M}\right). \tag{9}$$

Obviously, the total time of fused kernel is smaller than the non-fused one. However, in real situations, not every task can run concurrently with tasks of other types. We define a task is $fusible$, if there is a number of other type's tasks in nearby TBs, which can run concurrently with the particular task. Otherwise, the task is $infusible$. For better resource balancing, the number of infusible tasks should be kept to the minimum.

The proportion of task numbers in the original kernels is $S_I : S_G : S_O$. In the fused kernel, the number of tasks in an adjacent period of TBs should also follow the proportion. Otherwise, some tasks will be left behind, and become infusible tasks at the tail. As shown in Fig. 3b, we mix different type of tasks together so that they can execute concurrently.

5.3 Dependency Overhead Elimination

There is dependency relationship between tasks. As shown in Fig. 4, when a child task G_1 depends on a parent task I_1, if G_1 starts before I_1 finishes, it must wait until I_1 completes. Otherwise, the data required by child task is not ready, and it will produce incorrect results. We define the waiting time T_{wait}, which is caused by the current TB waiting for its unfinished parent tasks. T_{wait} of task G can be defined as

$$T_{wait}(G) = \max(0, \max^{S \in P_G}(T_{ed}(S)) - T_{st}(G)), \tag{10}$$

where P_G refers to all parent tasks of task G. Task G has to wait for all of its parent tasks to finish.

In NVIDIA GPUs, neighboring TBs are likely to be launched in a short time interval, which causes waiting if neighboring TBs have data dependency. To avoid this, we intentionally separate the dependent TBs when we map the tasks, keeping enough distance between them. We mentioned three types of task dependency. The dependency between GEMM tasks and $Filter\_Trans$ tasks are resolved by mapping the $Filter\_Trans$ tasks to the beginning TBs. For the other two types of dependency, we define their TB distances as follows. D_{IG} refers to the minimal interval of TB index between parent IN_trans task and child GEMM task, and D_{GO} refers to the minimal interval of TB index between parent GEMM task and child OUT_trans task. The two intervals should be long enough to avoid the waiting time of task dependency.

Fig. 3c shows the distribution of the running tasks during execution after optimizing for dependency. The first D_{IG} TBs are IN_trans tasks, because other tasks are not ready to execute at that time. For the next D_{GO} TBs, some GEMM tasks are ready to run, but OUT_trans tasks are still not

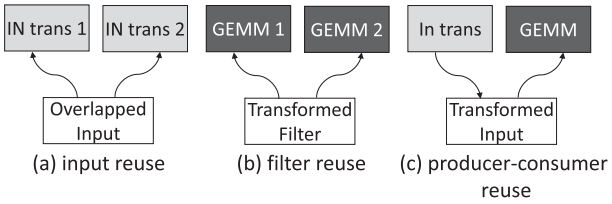


Fig. 5. Data reuse of three types.

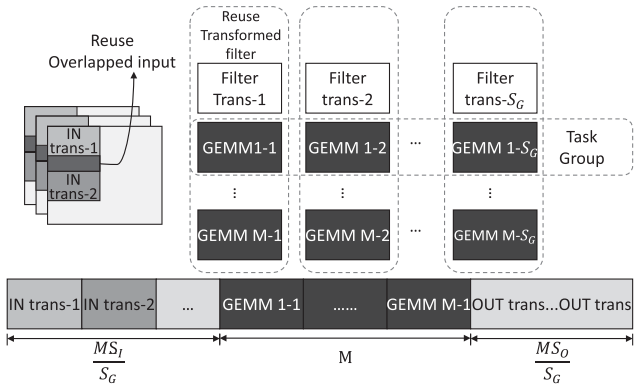


Fig. 6. Task mapping for input and filter reuse.

ready. Then each of three types of tasks can be available until input transformation tasks are used up. Resolving the dependency problem causes infusible tasks in the front D_{IG} TBs of IN_trans, and several TBs of OUT_trans TBs at the tail when other types of tasks are used up.

5.4 Data Reuse Optimization

Kernel fusion doesn't change the data reuse behavior inside TB, but it changes data reuse between different TBs by rearranging the order of tasks. Fig. 5 shows three types of data reuse. Type (a) is the reuse between two adjacent IN_trans tasks that share overlapping rows. Type (b) is the reuse between two GEMM tasks sharing the same transformed filter matrix. Type (c) is the producer-consumer data reuse of parent IN_trans task and child GEMM task. Parent GEMM tasks and child OUT_trans tasks have the same behavior of data reuse. For NVIDIA GPUs, every global memory transactions write L2 cache, which cannot be bypassed. Reusable data is stored in L2 cache in the first access. If the data didn't evict from cache in the second accessing, data can be fetched from the cache, which reduces the number of global memory transactions. To achieve the data reuse within L2 cache, the interval between two transactions on the same memory space must be small enough.

Fig. 6 gives an detailed model for data reuse of type (a) and (b). For type(a), the data reuse of input transformation, adjacent tasks share some overlapped data, and they can be automatically assigned to nearby TBs. For type (b), data reuse in GEMM tasks, M task groups form a grid of $M \times S_G$ GEMM tasks. Here S_G is the number of tasks in one GEMM task group. Every column of M GEMM tasks shares the same transformed filter matrix.

To maximize data reuse, we should group GEMM tasks in one column together to run in nearby TBs, because they share the same transformed filter matrix. To keep the proportion of task types within a task mapping pattern, every M GEMM TBs in the same column should launch together (having small TB distance) with MS_I/S_G IN_trans tasks and MS_O/S_G OUT_trans tasks. The sequence is illustrated at the bottom of Fig. 7. We repeat the sequence for S_G times to generate the task mapping for M task groups. In this way, for every M times of access on the transformed filter matrix, only the first time will suffer from cache miss. Other M-1 access will fetch data from the cache. The reuse rate is $(M-1)/M$, which increases when M grows larger.

Type (c) data reuse has more strict prerequisites. Data generated by parent tasks must stay in the L2 cache for enough long time because the child tasks start after a long interval. The channel sizes must be small enough for all transformed tiles to fit in the cache. Note that when type(c) data reuse satisfies, type (a) and (b) reuse are automatically viable because all data can be reused via cache. For typical Winograd convolution with float data type, type (c) data reuse is available when the channel number is smaller than 64.

5.5 Task Mapping Generation Algorithm

We discussed three optimization factors for kernel fusion so far: resource balancing, dependency waiting overhead elimination and data reuse optimization. These factors are tightly correlated, and cannot be optimized separately. We use three parameters D_{IG}, D_{GO}, M to demonstrate the optimization. More clearly, we can generate the task mapping sequence for every TB mapping with their tasks. Algorithm 1 presents the details. The tasking mapping algorithm contains four parts.

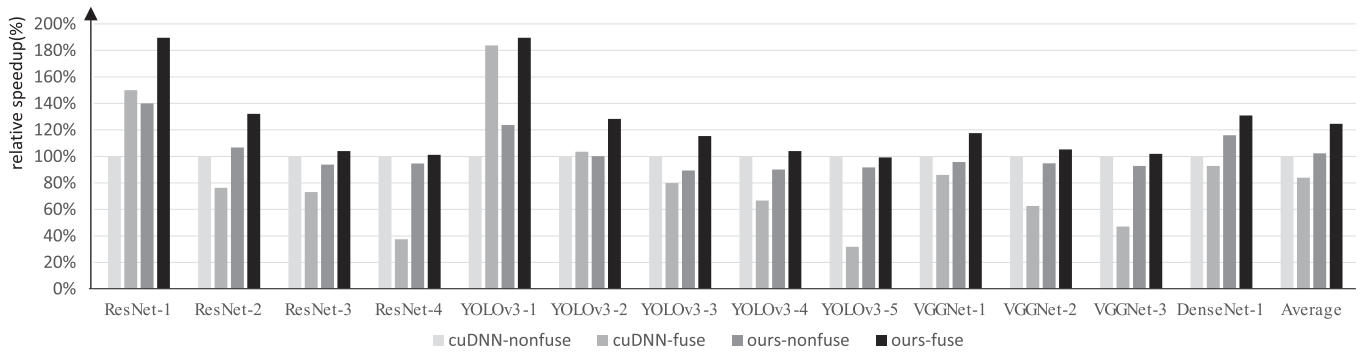


Fig. 7. Performance comparison of convolution layers with cuDNN and our Winograd implementations.

Algorithm 1. Task Mapping Algorithm

1: **Input** $N_F, S_I, S_G, S_O, D_{IG}, D_{GO}, M$
2: **Output** Task Mapping Sequence
3: **procedure** TaskMapping
4: First N_F TBs ← Filter_trans tasks.
5: Next D_{IG} TBs ← IN_trans tasks.
6: **for** $i \leftarrow 1$, to $D_{GO}/M(S_I + S_G)$ **do**
7: **for** $j \leftarrow 1$, to S_G **do**
8: MS_I/S_G TBs ← IN_trans tasks
9: M TBs ← GEMM tasks with jth filter matrix
10: **end for**
11: **end for**
12: **while** IN_trans task remains **do**
13: **for** $j \leftarrow 1$, to S_G **do**
14: MS_I/S_G TBs ← IN_trans tasks
15: M TBs ← GEMM tasks with jth filter matrix
16: MS_O/S_G TBs ← OUT_trans tasks
17: **end for**
18: **end while**
19: Last TBs ← remaining GEMM and OUT_trans tasks
20: **end procedure**

- *Filter Transformation TBs.* At the beginning, all filter transformation tasks are assigned to the first N_F TBs, since they are required by every GEMM task (Line 4). This part can also be separated into a new kernel which is invoked before the main kernel, since they are unable to work concurrently with GEMM tasks and no data reusing is available.
- *Initial TBs.* Because of distance restriction, only input transformation task can be placed in the next D_{IG} TBs (Line 5). Both input transformation and GEMM tasks can be placed in the following D_{GO} TBs. For M task groups, GEMM tasks reusing the same filter matrix are grouped together. IN_trans tasks are also separated to S_G groups and inserted to the task sequence, to keep the task proportion for resource balancing. (Line 6∼11).
- *Body TBs.* In the following TBs, each of the three types of tasks can be placed. Similar to above, M IN_trans, GEMM and OUT_trans task groups are rearranged to S_G groups. Each contains MS_I/S_G IT_trans tasks, M GEMM tasks and MS_O/S_G OUT_trans tasks. M GEMM tasks share the same transformed filter matrix. This step repeats until IN_trans tasks are used up. (Line 12 ∼18).
- *Tail TBs.* Finally, input transformation tasks will be used up, and the remaining tasks fill the last TBs of the fused kernel (Line 19).

6 Implementation of Task Scheduler

Although we manage to eliminate the task dependency overhead in the task mapping process, the actual execution time of TBs can be affected by some random factors, and the child task can sometimes start before the parent task finishes. To prevent from generating wrong results, we must guarantee that every child task starts after the completion of its parent tasks with the task scheduler.

Listing 2 shows a pseudo CUDA implementation of task scheduler to resolve the dependency of parent and child tasks. We introduce a volatile global array $counter$, which records the number of unfinished parent tasks in each task group. The counter is initialized with the total number of parent tasks of one task group. After each parent TB finishes, it atomically decrements the counter of its task group by one. Each TB only performs once and thread 0 does the job. For the child TBs, before the TB calls its device function, it checks the dependency counter to see whether it reaches zero, which means all parent tasks of its task group have finished. If not, the child TB keeps polling the counter until reaching zero. The $counter$ is implemented in $volatile$ to prevent caching.

Listing 2. Pseudo Code for Dependency Resolving

```
1  kernel(Task* task_map, volatile int* counter){
2    item=task_map[blockIdx];
3    switch(item.type){
4      case PARENT:
5        parent_func();
6        thread_synchronization();
7      if(threadIdx.x==0)
8        atomicDec(*counter[item.groupid]);
9      break;
10     case CHILD:
11       while(counter[item.groupid]!=0);
12       child_func();
13   }
14 }
```

7 Experiments

7.1 Experiment Overview

We perform experiments on NVIDIA Tesla V100 GPU. We test the typical 3×3 convolution layers from the state-of-the-art neural networks and compare the performance of our proposed kernel fusion design with cuDNN library. For Winograd convolution algorithm, cuDNN supports two implementations, *WINOGRAD* and *WINOGRAD_NONFUSED*. cuDNN's non-fused version is composed of four kernels corresponding to our input transformation, filter transformation, GEMM, and output transformation kernels. The software version in our experiment is CUDA 10.0 and we compare our performance of convolution with cuDNN v7.3.1. Our experiment mainly targets on Winograd convolutions, because it has a better performance compared with FFT on 3×3 convolution, and is already widely adopted in typical convolution layers.

To perform our experiments, we first implement the non-fused version of $F(4 \times 4, 3 \times 3)$ Winograd convolution with four kernels, which is the same as cuDNN's non-fused algorithm. We use NHWC data layout in our experiment for better memory coalescing. For transformation kernels, we use one thread to process transformation for one 6×6 tile, and one TB consists of 128 threads. For GEMM tasks, NVIDIA's GEMM library cuBLAS is closed source and not available to call inside kernels. We implemented GEMM by splitting the multiplication of $t \times C$ and $C \times K$ matrices to multiple tasks, according to channel sizes C and K. Fig. 7 shows that our non-fused implementation is slightly slower than cuDNN due to the inefficiency of GEMM kernel. To build the fused kernel, we rewrite the kernels to device functions, and bind them together with the task scheduler.

In the following, we first compare the performance of our fused kernel and cuDNN's kernels with single layers

TABLE 2
Convolution Layer Specifications

	K	C	H	W
ResNet-1_x	64	64	56	56
ResNet-2_x	128	128	28	28
ResNet-3_x	256	256	14	14
ResNet-4_x	512	512	7	7
YOLOv3-1	64	32	128	128
YOLOv3-2	128	64	64	64
YOLOv3-3	256	128	32	32
YOLOv3-4	512	256	16	16
YOLOv3-5	1024	512	8	8
VGGNet-1	128	128	112	112
VGGNet-2	256	256	56	56
VGGNet-3	512	512	28	28
DenseNet-1	48	192	56	56

extracted from neural networks. We discuss the effect of layer specification and fusion parameters by comparing the performance of different parameter values. Then, we integrate our fused convolution kernels into PyTorch deep learning framework [15] to evaluate the overall optimization for different networks. Finally, we also compare our results with some previous works. Our technique can be extended to other GPUs and multi-GPU platform straightforwardly.

7.2 Single Layer Performance

To evaluate the performance of kernel fusion on Winograd convolution algorithm, we test our fusion technique using several typical 3x3 layers from the state-of-the-art networks: YOLO-v3 [1], ResNet-50 [3], VGG-16 [16] and DenseNet-161 [17]. These convolution networks are widely used in computer vision tasks such as classification and object detection. They are also representative as benchmarks for the performance of convolution implementations.

Table 2 shows the parameters of different 3 x 3 convolution layers in our experiment. Stride and padding values are both 1 for every layer. We set the batch size of convolution to 64, and we only test the forward pass. We benchmark our fused, non-fused implementations with cuDNN's fused, non-fused Winograd implementation. Our implementation of convolution is based on NHWC-layout. Since cuDNN's performance on NHWC-layout is extremely slow (2∼3 times slower than NCHW-layout), we evaluate the performance of our implementation with cuDNN's implementation on NCHW-layout. The numerical result of our fused kernel is the same as cuDNN's NHWC version.

We set cuDNN's non-fused implementation as a baseline and evaluate the speedup of the other three implementations as shown in Fig. 7. For cuDNN convolution, the fused version is only faster than the non-fused version when channel size is smaller than 128 (ResNet-1, YOLOv3-1, YOLOv3-2). When channel size grows larger, the performance of fused cuDNN degrades heavily. On average, our non-fused kernels is 6 percent slower than cuDNN's faster implementation because of the slower GEMM kernels. For 12 of 13 convolution layers, our fused kernels outperform cuDNN's both fused and non-fused algorithm. Our fused kernels achieve an average speedup of 1.25X over cuDNN-nonfused, 1.7X over cuDNN-fused, and 1.13X over cuDNN's faster implementation.

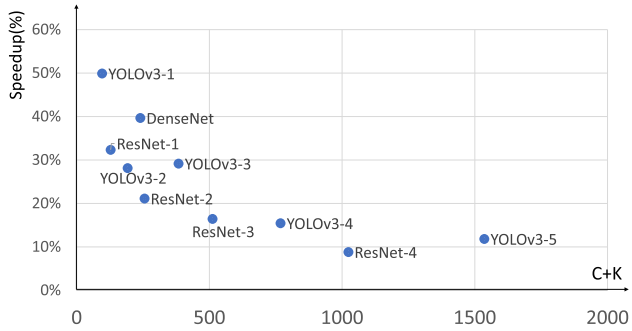


Fig. 8. Speedup on different channel size.

Our fused kernel is only 1 percent slower than cuDNN-fused version on YOLOv3-5 layer.

7.3 Analysis on Channel and Problem Size

Here, we analyze the influence of channel size and problem size on the performance.

Channel Size. Fig. 8 shows the performance speedup of our fused kernel versus our non-fused kernel when the channel size varies. X-axis is the sum of the input channel size C and output channel size K, and Y-axis is the speedup. We observe that the speedup of kernel fusion decreases as channel sizes grow larger. The reason comes from two aspects. First, for convolution with large channel size, the execution time of transformation tasks is far less then GEMM tasks. The effect of resource balancing on kernel fusion is not significant, and the overall performance is still bounded by GEMM computation which cannot be decreased. Second, for layers with small channel sizes, the producer-consumer data reuse optimization eliminates a huge amount of global memory transactions in the fused kernel, but it cannot be achieved in large channels because of the limited cache size.

Problem Size. Problem size for fast convolution algorithms can be represented as $Tiles\_per\_Image \times Batch\_size$. As the layers going deeper, the channel size increases but $Tiles\_per\_Image$ decreases, so problem size also decreases. Obviously, our fusion technique prefers larger problem size for more performance speedup. With problem size growing larger, the number of task increases, but the number of infusible tasks at front and tail remain unchanged. Thus, overall resource balancing is improved. In our experiment, VGGNet layers have same channel size as ResNet layers, but their number of tiles are different. We don't observe performance speedup on large problem size over small problem size compared with cuDNN, because cuBLAS is optimized for larger GEMMs. When compared with our non-fused version, larger problem size still has benefits.

7.4 Analysis on Task Mapping Parameters

Our task mapping algorithm relies on parameters to maintain task distance and group size. For each layer, we select the parameters which generate fused kernel with the best performance. Now we discuss the effect of task mapping parameters on different layers.

We evaluate the performance of fused kernel with different D_{IG} and M value on four layers from ResNet. Results

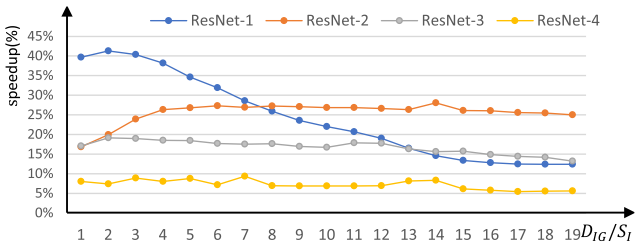


Fig. 9. Effect of task distance.

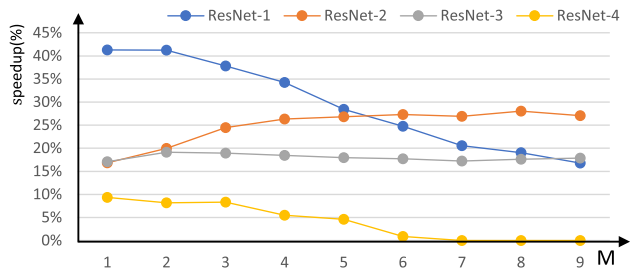


Fig. 10. Effect of group size.

are shown in Figs. 9 and 10. For convolution with small channel size (ResNet-1), small task distance and group size bring better performance. In this case, speedup mainly comes from producer-consumer data reuse, which requires a smaller distance between parent and child tasks.

For convolution with larger channel size, only filter matrix reuse and input tensor reuse are feasible. Larger task distances are maintained to reduce dependency overhead, and larger group size is used for filter matrix reuse. But when the parameters grow even larger, the number of infusible tasks increases and performance will slow down.

7.5 Convolution Network Performance

To evaluate the performance of our fused kernel on whole convolution networks, we integrate our kernel as a custom extension in PyTorch deep learning framework. The result is shown in Fig. 11. We replace the 3×3 convolution ($nn.Conv2d$ op with kernel size = 3 and stride = 1) operation with our fused kernel, and measure the overall time speedup of every 3×3 convolution layers in the convolution network. The shape of input image is (N, C, H, W) = (64, 3, 224, 224) for every networks. Batch size for DenseNet-161 is 48 due to limitation of device memory.

For 3×3 layers in DenseNet, our fused kernel can achieve 50 percent speedup over cuDNN, because DenseNet is composed of $(K, C) = (48, 192)$ convolution layers. Channel sizes are relatively small so fusion can achieve considerable speedup. The other three networks are mainly composed of layers with large channels (channel size ≥ 256), so our fused kernels only achieve about 10 percent speedup.

7.6 Comparison With Other Works

Apart from cuDNN Winograd-based convolution, we also compared our result with other convolution algorithm (cuDNN-GEMM) and other previous works on ResNet 2_x layer. The result is shown in Fig. 12. TVM [18] provides a code-generation approach for convolution optimization based on winograd convolution, but their performance is

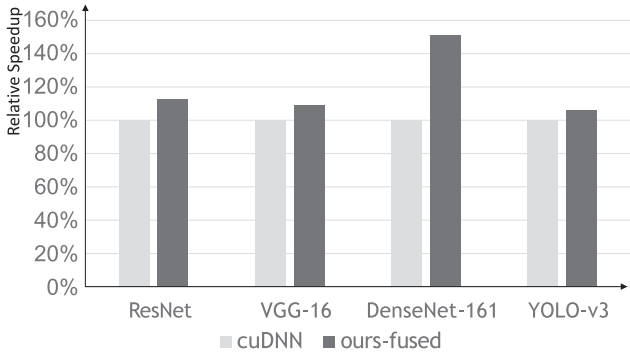


Fig. 11. Speedup for convolutional networks.

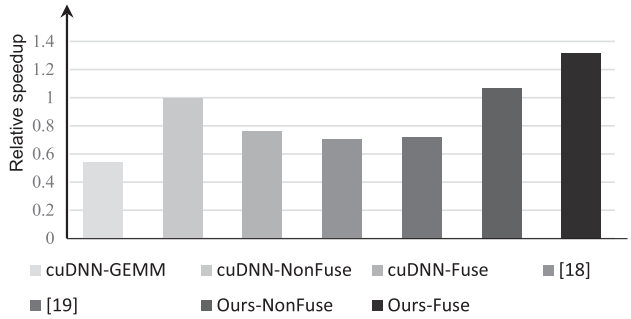


Fig. 12. Speedup comparison with other works for ResNet 2_x.

slower than cuDNN. [19] provides a tiling and batching engine based on GEMM convolution. Although their performance is faster than cuDNN-GEMM, it's slower than cuDNN-Winograd. Our approach is the only work that outperforms the fastest cudNN algorithm.

8 Related Work

Fast Processing of Neural Networks. Thanks to the massive parallelism capability, GPU is the most widely used accelerator for DNNs. Implementation of operations such as convolution, pooling, and activation layers are supported in various deep learning frameworks and libraries [9], [10], [20]. A significant progress of DNN processing is the discovery of fast algorithms, namely FFT convolution [21] and Winograd convolution [8], which reduces the arithmetic complexity for convolution. NVIDIA and Nervana implement the fast algorithms to their GPU deep learning library Neon and cuDNN. However, Neon's implementation of Winograd convolution is based on assembly code of earlier architectures, and we are not able to evaluate them. NVIDIA also presented a high-performance open-source linear algebra library CUTLASS [22]. Recent works also proposed effective implementations of Winograd convolution algorithm on Manycore CPU platform [23], and FPGA platform [24].

Apart from the optimization on convolution algorithm, other works focus on the optimization of neural network processing with the consideration of hardware architecture [25], [26]. Some works discuss factors of memory layout, and optimizes register/memory efficiency and tiling strategies of GEMM-based convolution [19], [27], [28], pooling and softmax layers [29], [30], [31]. In [32], the authors also use kernel fusion to eliminate data transfer with off-chip memory, but their fusion is cross-layers, which is different

from our technique that apply kernel fusion within a single layer.

Code generation is a different approach for high-performance computation of DNNs. By building an optimization system, code generator is able to auto-tune the optimized implementation for specific workload and hardware. Halide, TVM, Tensor Comprehension and other tools are able to generate CUDA source code of DNN kernels [18], [33], [34], [35].

GPU Task Scheduling. The fused convolution is considered as an irregular workload on GPUs, because tasks can have heavy dependency on each other, and cannot be fully parallel. Many scheduler for irregular GPU workloads are based on persistent thread design [11], while the same numbers of TBs stay active during the entire execution process. Kernel fusion and fission have be proposed to effectively reorganize the execution of GPU kernels [36]. Tzeng *et al.* [37] focus on the dependency between GPU tasks, and presented static and dynamic scheduling schemes to handle task dependencies on GPUs. The static scheduling is the same as our fusion model. Some other works intend to reduce the scheduling overhead. R Nasre *et al.* [38] presents atomic synchronization and lock-free synchronization for inter-block GPU data communication to implement barriers. B Wu *et al.* presented an approach to flexibly control GPU task scheduling [39]. For cache-level optimization, Chen *et al.* [40] studied on cache-friendly GPU scheduling techniques, and Xie *et al.* presents a cache bypassing technique for GPU warp scheduler and a compiler-level optimizer for register and cache optimization [41], [42]. Liang *et al.* [43] analyzed the resource utilization and partition for GPU multitasking. Zheng *et al.* [12] discussed different task scheduling strategies, including Megakernels and different pipelined scheduling models.

9 Conclusion

This paper proposes a kernel fusion technique targeting the efficient processing of fast convolution algorithms on GPU. We use a MegaKernel-based fusion technique which assign different tasks to different GPU thread blocks. We develop a static task mapping algorithm to assign tasks into thread blocks with the consideration of GPU performance. We use a task scheduler to maintain correct task dependency relationship on execution. Our kernel fusion technique enables better resource balancing and data reuse with minimized dependency overhead. Evaluation on modern CNNs shows that our technique can achieve an average of 1.25X and 1.7X speedup over cuDNN's non-fused and fused implementations on Winograd convolution algorithm.

Acknowledgments

This work was supported by the National Natural Science Foundation China under Grant 61672048 and Grant 61520106004, and PKU-SenseTime Joint Lab.

References

[1] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 779–788.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. 25th Int. Conf. Neural Inf. Process. Syst.*, 2012, pp. 1106–1114.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[4] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.

[5] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," *ACM SIGARCH Comput. Architecture News*, vol. 44, no. 3, pp. 367–379, 2016.

[6] T. Chen *et al.*, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proc. 19th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2014, pp. 269–284.

[7] M. Zhang, S. Rajbhandari, W. Wang, and Y. He, "DeepCPU: Serving RNN-based deep learning models 10x faster," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 951–965.

[8] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 4013–4021.

[9] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun, "Fast convolutional nets with fbfft: A GPU performance evaluation," in *Proc. 3rd Int. Conf. Learn. Representations*, 2015.

[10] S. Chetlur *et al.*, "cuDNN: Efficient primitives for deep learning," *CoRR*, vol. abs/1410.0759, 2014.

[11] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style GPU programming for GPGPU workloads," in *Proc. Innovative Parallel Comput.-Found. Appl. GPU Manycore Heterogeneous Syst.*, 2012, pp. 1–14.

[12] Z. Zheng, C. Oh, J. Zhai, X. Shen, Y. Yi, and W. Chen, "VersaPipe: A versatile programming framework for pipelined computing on GPU," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2017, pp. 587–599.

[13] Is blockidx correlated to the order of block execution? 2017. [Online]. Available: https://stackoverflow.com/questions/46660053/is-blockidx-correlated-to-the-order-of-block-execution

[14] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "GPU scheduling on the NVIDIA TX2: Hidden details revealed," in *Proc. IEEE Real-Time Syst. Symp.*, 2017, pp. 104–115.

[15] A. Paszke *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Proc. 33rd Int. Conf. Neural Inf. Process. Syst*, 2019, pp. 8024–8035.

[16] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. 3rd Int. Conf. Learn. Representations*, 2015.

[17] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 2261–2269,

[18] T. Chen *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 578–594.

[19] X. Li, Y. Liang, S. Yan, L. Jia, and Y. Li, "A coordinated tiling and batching framework for efficient GEMM on GPUs," in *Proc. 24th Symp. Princ. Pract. Parallel Program.*, 2019, pp. 229–241.

[20] INTEL, "Neon," 2015. [Online]. Available: https://github.com/NervanaSystems/neon

[21] M. Mathieu, M. Henaff, and Y. LeCun, "Fast training of convolutional networks through FFTs," in *Proc. 2nd Int. Conf. Learn. Representations*, 2014.

[22] NVIDIA, "CUTLASS: Fast linear algebra in CUDA C++," 2017. [Online]. Available: https://devblogs.nvidia.com/cutlass-linear-algebra-cuda/

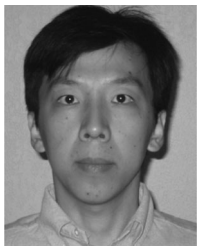
[23] Z. Jia, A. Zlateski, F. Durand, and K. Li, "Optimizing N-dimensional, winograd-based convolution for manycore CPUs," in *Proc. 23rd ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2018, pp. 109–123.

[24] Q. Xiao, Y. Liang, L. Lu, S. Yan, and Y.-W. Tai, "Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on FPGAs," in *Proc. 54th Annu. Des. Autom. Conf.*, 2017, Art. no. 62.

[25] A. A. Awan, C. Chu, H. Subramoni, X. Lu, and D. K. Panda, "OC-DNN: Exploiting advanced unified memory capabilities in CUDA 9 and volta GPUs for out-of-core DNN training," in *Proc. 25th IEEE Int. Conf. High Perform. Comput.*, 2018, pp. 143–152.

[26] A. A. Awan, H. Subramoni, and D. K. Panda, "An in-depth performance characterization of CPU- and GPU-based DNN training on modern architectures," in *Proc. Mach. Learn. HPC Environments*, 2017, pp. 8:1–8:8.
[27] A. Abdelfattah, A. Haidar, S. Tomov, and J. J. Dongarra, "Performance, design, and autotuning of batched GEMM for GPUs," in *Proc. 31st Int. Conf. High Perform. Comput.*, 2016, pp. 21–38.
[28] Y. Nagasaka, A. Nukada, R. Kojima, and S. Matsuoka, "Batched sparse matrix multiplication for accelerating graph convolutional networks," *19th Int. Symp. Cluster, Cloud Grid Comput.*, pp. 231–240, 2019.
[29] C. Li, Y. Yang, M. Feng, S. T. Chakradhar, and H. Zhou, "Optimizing memory efficiency for deep convolutional neural networks on GPUs," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2016, pp. 633–644.
[30] M. Cho and D. Brand, "MEC: Memory-efficient convolution for deep neural network," in *Proc. 34th Int. Conf. Mach. Learn.*, 2017, pp. 815–824.
[31] X. Li, G. Zhang, H. H. Huang, Z. Wang, and W. Zheng, "Performance analysis of GPU-based convolutional neural networks," in *Proc. 45th Int. Conf. Parallel Process.*, 2016, pp. 67–76.
[32] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2016, pp. 22:1–22:12.
[33] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. P. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2013, pp. 519–530.
[34] N. Vasilache *et al.*, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," *CoRR*, vol. abs/1802.04730, 2018.
[35] S. G. D. Gonzalo, S. Huang, J. Gómez-Luna, S. D. Hammond, O. Mutlu, and W. Hwu, "Automatic generation of warp-level primitives and atomic instructions for fast and portable parallel reduction on GPUs," in *Proc. IEEE/ACM Int. Symp. Code Generation Optim.*, 2019, pp. 73–84.
[36] H. Wu, G. Diamos, J. Wang, S. Cadambi, S. Yalamanchili, and S. Chakradhar, "Optimizing data warehousing applications for GPUs using kernel fusion/fission," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp. Workshops PhD Forum*, 2012, pp. 2433–2442.
[37] S. Tzeng, B. Lloyd, and J. D. Owens, "A GPU task-parallel model with dependency resolution," *Comput.*, vol. 45, no. 8, pp. 34–41, Aug. 2012.
[38] S. Xiao and W. Feng, "Inter-block GPU communication via fast barrier synchronization," in *Proc. 24th IEEE Int. Symp. Parallel Distrib. Process.*, 2010, pp. 1–12.
[39] B. Wu, G. Chen, D. Li, X. Shen, and J. S. Vetter, "Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations," in *Proc. 29th ACM Int. Conf. Supercomputing*, 2015, pp. 119–130.
[40] Y. Chen, A. B. Hayes, C. Zhang, T. Salmon, and E. Z. Zhang, "Locality-aware software throttling for sparse matrix operation on GPUs," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 413–426.
[41] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang, "Coordinated static and dynamic cache bypassing for GPUs," in *Proc. 21st IEEE Int. Symp. High Perform. Comput. Architecture*, 2015, pp. 76–88.
[42] X. Xie *et al.*, "Enabling coordinated register allocation and thread-level parallelism optimization for GPUs," in *Proc. 48th Int. Symp. Microarchitecture*, 2015, pp. 395–406.
[43] Y. Liang, H. P. Huynh, K. Rupnow, R. S. M. Goh, and D. Chen, "Efficient GPU spatial-temporal multitasking," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 3, pp. 748–760, Mar. 2015.

Liancheng Jia received the BS degree in computer science from Peking University, China. Since 2018, he is working toward the PhD degree in Center for Energy-Efficient Computing and Application (CECA), Peking University, Beijing, China. His current research interests include high-performance computation in GPU and embedded systems.



Yun Liang (Senior Member, IEEE) received the PhD degree in computer science from the National University of Singapore, Singapore, in 2010 and worked as a research scientist in UIUC before he joins Peking University, China. He is an associate professor (with tenure) in School of EECS, Peking University, China. His research interests include heterogeneous computing (GPUs, FPGAs, ASICs) for emerging applications such as AI and big data, computer architecture, compilation techniques, programming model and program analysis, and embedded system design. He has authored more than 80 scientific publications in premier international journals and conferences in related domains. His research has been recognized by Best Paper Award at FCCM 2011 and ICCAD 2017 and best paper nominations at PPoPP 2019, DAC 2017, ASPDAC 2016, DAC 2012, FPT 2011, and CODES+ISSS 2008. He serves as associate editor for the *ACM Transactions in Embedded Computing Systems (TECS)* and the *Embedded System Letters*, and serves in the program committees in the premier conferences in the related domain including (HPCA, DAC, ASPLOS, PACT, PPoPP, CGO, ICCAD, ICS, CC, DATE, CASES, ASPDAC, and ICCD).

Xiuhong Li received the BS and PhD degrees from Center for Energy-efficient Computing and Applications (CECA), Peking Univiersity, China. He is currently a senior researcher of deep learning platform department at Sensetime. His research interests include high-performance computation on GPUs and deep learning compiler.

Liqiang Lu received the BS degree from the Institute of Microelectronics, Peking University, Beijing, China, in 2017. He is currently working toward the PhD dgree in the School of EECS, Peking University, China. His research interests include algorithm-level and architecture-level optimizations of FPGA for machine learning applications.

Shengen Yan received the BS and PhD degrees from the Harbin Institute of Technology, Harbin, China and Institute of Software, Chinese Academy of Sciences, Beijing, China on 2009 and 2014, respectively. He was a visiting student in NC State University, Raleigh, North Carolina from June 2013 to February 2014. He was a postdoctoral researcher at Multimedia Lab in the Chinese University of Hong Kong, Hong Kong from 2015 to 2017. Currently, he is the executive research director of deep Learning platform department at SenseTime. His research interests interests include large scale deep learning and high performance computing. He has published about 20 papers in the area of parallel computing and deep learning.