Optimizing Batched Winograd Convolution on GPUs

Da Yan
HKUST
dyanab@cse.ust.hk

Wei Wang
HKUST
weiwa@cse.ust.hk

Xiaowen Chu
Hong Kong Baptist University
chxw@comp.hkbu.edu.hk

Abstract

In this paper, we present an optimized implementation for single-precision Winograd convolution on NVIDIA Volta and Turing GPUs. Compared with the state-of-the-art Winograd convolution in cuDNN 7.6.1, our implementation achieves up to 2.13× speedup on Volta V100 and up to 2.65× speedup on Turing RTX2070. On both Volta and Turing GPUs, our implementation achieves up to 93% of device peak.

Apart from analyzing and benchmarking different high-level optimization options, we also build a SASS assembler *TuringAs* for Volta and Turing that enables tuning the performance at the native assembly level. The new optimization opportunities uncovered by *TuringAs* not only improve the Winograd convolution but can also benefit CUDA compilers and native assembly programming. We have released *TuringAs* as an open-source software. To the best of our knowledge, this is the first public-available assembler for Volta and Turing GPUs.

CCS Concepts • **Theory of computation** → **Massively parallel algorithms**; • **Computing methodologies** → *Neural networks*; • **Software and its engineering** → *Assembly languages*;

Keywords Convolution, GPU, Performance

1 Introduction

Convolutional Neural Network (CNN) has demonstrated state-of-the-art performance in many computer vision and machine learning applications [4, 8, 22, 24]. However, training CNN models on large datasets is computationally expensive, often requiring hundreds of GPU-hours [3]. The key to improving the training performance is to accelerate the *convolutional operations* used in the convolutional layers of CNN models, which are computation-intensive by nature and usually dominate the training time [23].

PPoPP '20, February 22–26, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-6818-6/20/02…$15.00
https://doi.org/10.1145/3332466.3374520

Winograd [11] was proposed recently as an efficient algorithm to speed up convolutional operations in CNNs. It reduces the number of arithmetic operations required in convolution using Shmuel Winograd's minimal filtering algorithm [26]. Theoretical analysis shows that Winograd convolution can reduce the arithmetic complexity by 2.25× for popular 3×3 filters in the state-of-the-art CNN models [11]. Owing to its significant performance benefits, Winograd convolution has quickly gained its popularity and has been supported by modern deep learning libraries such as Nvidia cuDNN and Intel(R) MKL-DNN.

However, it remains a challenge to efficiently implement Winograd convolution on GPUs: the state-of-the-art implementation fails to deliver the full speedup as promised in theory. We benchmarked the performance of Winograd convolution in cuDNN 7.6.1 for all 3×3 convolutional layers in ResNet [4] on an Nvidia Tesla V100 GPU. Compared with GEMM-based convolution, Winograd convolution only achieves 0.81×-1.67× speedup with an average of 1.4× (Section 2.2), which is far below the expected speedup with 2.25× reduction of multiplications shown in theory.

To bridge the gap between the theoretical benefits and those achieved in practice, we need to address the following implementation challenges:

1. As a multi-step algorithm, Winograd convolution requires data transposing between two steps, in which global memory accesses should be coalesced and shared memory accesses should be free of bank conflict. Both requirements pose more constraints to the data layout design.
2. Compared with the heavily studied matrix multiplication, the computation intensity of Winograd convolution is lower, leaving less room for latency hiding.
3. GPU hardware has limited regular and predicate registers. We need to tailor the implementation to meet the constraints while achieving high performance.

In this paper, we tackle the aforementioned challenges with the following approaches:

1. We redesign the workload partition and data layout to make the global memory access fully coalesced and shared memory access bank conflict-free.
2. We enlarge the cache blocking size to increase the computation intensity. We also hide global memory latency and shared memory latency with software pipelining.
3. We ensure that the registers required by the main loop are below the hardware constraint. Predicate registers

are packed to regular register to eliminate the recomputation of zero-padding masks.

To implement these optimization techniques, we must address two problems. First, configuring a large cache block size enforces more threads to run in a synchronized manner, making the performance more sensitive to the balance of the progress on different warps. Second, efficient predicate register to regular register packing (the *P2R* (Predicate to Register) instruction) is not exposed at CUDA C/C++ or PTX level. Without such capability, more regular registers are required to hold predicate information, which leads to register spilling.

Note that the *P2R* instruction and the control logic to balance the progress between different warps are only accessible at the SASS (Shader ASSembly) level. Yet, there is no publicly available SASS assembler for NVIDIA Volta and Turing GPUs. We therefore build a SASS assembler[1] for Volta and Turing, with which we can achieve a balanced progress between warps and P2R instructions, so as to fully saturate the hardware.

Combining the high-level and SASS-level optimizations, we implement an efficient Winograd convolution. We evaluate our implementation on NVIDIA Turing RTX2070 and Volta V100 GPUs on all 3×3 convolutional layers in ResNet [4]. The results show that compared with the state-of-the-art implementation of Winograd convolution in cuDNN 7.6.1, our implementation delivers up to 2.65× (1.96× on average) speedup on RTX2070, and up to 2.13× (1.5× on average) speedup on V100. On both devices, our implementation achieves up to 93% of theoretical peak, narrowing the gap between theory and practice.

We summarize our main contributions as follows:

- We build a SASS assembler for NVIDIA Volta and Turing GPUs. To the best of our knowledge, this is the first publicly available SASS assembler for Volta and Turing architectures.
- We implement a single-precision Winograd convolution for 3×3 kernels. The optimized Winograd convolution achieves up to 93% of device peak and up to 2.65× speedup over the state-of-the-art cuDNN 7.6.1.
- We study the effect of different SASS-level optimization techniques, including the warp load balancing with yield flag and the load/store instruction scheduling strategies. Our experiment shows that tuning the yield flag alone contributes to around 10% higher throughput. To our knowledge, this is the first study on the effect of the yield flag.

2 Background and Motivation

In this section, we briefly introduce the Winograd convolution algorithm. We show through measurement studies that the state-of-the-art implementations fail to deliver the performance speedup as promised in theory. We also summarize the major technical challenges posed by an efficient implementation of Winograd algorithm. We refer to [18] for a CUDA programming guide and [17] for a detailed description of the Turing architecture.

2.1 Winograd Convolution

In CNN models, the 3×3 convolutional layers serve as important building blocks. For example, in VGG19 model [24], 16 out of 19 layers are 3×3 convolutional layers; in ResNet34 model [4], 32 out of 34 layers are 3×3 convolutional layers.

The Winograd convolution employs the Winograd minimal filtering algorithm [26] and can reduce the number of multiplications for 3×3 layers by at least 2.25× [11]. We briefly illustrate how this can be done yet refer to [11] for a detailed description of the algorithm.

To compute the convolution $O = I * F$, where I is 4×4 input, F is 3×3 filter, and O is 2×2 output (denoted $F(2 \times 2, 3 \times 3)$), direct convolution needs $2 \times 2 \times 3 \times 3 = 36$ multiplications while Winograd convolution only needs 16 (element-wise) multiplications[2] through the following equivalent computation:

$$O = A^T[(GFG^T) \odot (B^T IB)]A. \tag{1}$$

where $\odot$ denotes element-wise multiplication, and A^T, G, B^T are respectively

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}, \tag{2}$$

$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}, B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}. \tag{3}$$

Here, $\hat{F} = GFG^T$ is the **filter transformation (FTF)**, which needs 28 float instructions; $\hat{I} = B^T IB$ is the **input transformation (ITF)**, which needs 32 float additions; $\hat{O} = \hat{F} \odot \hat{I}$ is the **element-wise multiplication (EWMM)**; $O = A^T \hat{O} A$ is the **output transformation (OTF)**, which needs 24 float additions.

Note that the transformation matrices for the $F(3\times3, 2\times2)$, $F(4 \times 4, 3 \times 3)$ and the other cases are also given in [11, 26]. In this paper, we limit the discussion to $F(2 \times 2, 3 \times 3)$ only, a common case in practice.

2.2 Efficiency of Current Implementation

We evaluate the performance of Winograd convolution against GEMM-based convolution in cuDNN 7.6.1 on all 3×3 convolutional layers in ResNet (parameters listed in Table 1) with different batch sizes on a V100 GPU. ResNet is a widely used CNN model that has been included in the standard machine learning benchmarks like *MLPerf* [21].

[1] https://github.com/daadaada/turingas

[2] We only consider element-wise multiplication as the operations needed by transformation can be amortized by a large number of channels.

Layer	Output($H \times W$)	Filter ($C, R \times S, K$)
Conv2	56×56	$[64, 3 \times 3, 64]$
Conv3	28×28	$[128, 3 \times 3, 128]$
Conv4	14×14	$[256, 3 \times 3, 256]$
Conv5	7×7	$[512, 3 \times 3, 512]$

Table 1. All 3×3 convolutional layers in ResNet. In the rest of this work, we use *ConvxNn* to represent convolution layer *x* with batch size *n*. For example, *Conv2N32* represent *Conv2* layer with batch size *32*.

We use speedup over GEMM-based convolution as a proxy for the gap between the current implementation and the upper bound. The speedup is expected to be around 2.25×. However, our experimental results in Table 2 show that the average speedup over GEMM-based convolution is only 1.4×, suggesting a significant room for improvement.

	Layers			
N	Conv2	Conv3	Conv4	Conv5
32	1.57×	1.53×	1.62×	1.10×
64	1.54×	1.50×	1.57×	0.91×
96	1.59×	1.53×	1.58×	0.81×
128	1.55×	1.48×	1.67×	0.86×

Table 2. Speedup of cuDNN's Winograd convolution over cuDNN's GEMM-based convolution on V100.

2.3 Challenges in Optimizing Winograd Convolution

It is harder to optimize Winograd convolution than the GEMM-based convolution due to the following challenges.

First, the multiple steps make the algorithm hard to optimize in nature. We need to design the layout to maximize throughput when transposing data. We summarize our layout in Section 4. Moreover, batched GEMM is a subproblem of Winograd convolution. All the techniques we have developed in Section 4.3 can be applied to batched GEMM.

Second, the computation intensity of $F(2 \times 2, 3 \times 3)$ Winograd convolution is 2.25× lower than the GEMM-based convolution (Figure 2), which poses a tighter constraint on latency hiding. We enlarge the cache block size to increase computation intensity. As a result, more registers are used to do software pipelining compared with GEMM. The high pressure on registers pushes us to save registers with *P2R*[3], which is only accessible at SASS level.

[3]For example, we can pack *P0* to *P3* (4 predicate registers) to one 32-bit register (*R0*) with **P2R R0, 0xf;**, and unpack the 0 to 3 bits of *R0* to *P0* to *P3* with **R2P R0, 0xf;**.

2.4 Necessity of SASS Programming

With our SASS assembler, *TuringAs*, we can not only access *P2R*, but also place load/store instructions at better locations (Section 6.2). Also, we found that the suboptimality of yield flag[4] in the NVCC and cuDNN hurts performance. We show that by changing the yield flag, we can achieve 10% higher throughput than NVCC-generated code and cuDNN's code in Section 6.1. To the best of our knowledge, this is the first time that the effect of yield flag is investigated.

TuringAs enables more applications beyond performance optimization. First, developers can use it to benchmark performance without worrying about the compiler reordering or optimizing away some code. Second, it will enable a deeper understanding of the GPU hardware. Finally, comparing the human-optimized SASS code and compiler-generated SASS code gives insights to improving algorithms in the compiler.

3 Design Overview

In this section, we introduce the basic workflow and how we partition and map the workload to tens of SMs on a GPU card. These are the fundamentals of the implementation.

We also introduce the philosophy based on which we choose the important cache block size, the software pipelining technique to hide memory access latency, and how we do zero-padding implicitly.

Notations used in this work are listed in Table 3.

Symbol	Meaning
$I_{c,h,w,n}$	Input data element
$F_{c,r,s,k}$	Filter element
$\tilde{h}$	Tile index in height
$\tilde{w}$	Tile index in width
$\hat{I}_{c,\tilde{h},\tilde{w},n}$	Transformed input tile
$\hat{F}_{c,k}$	Transformed filter tile
$\hat{O}_{k,\tilde{h},\tilde{w},n}$	Pre-transform output tile
$O_{k,\tilde{h},\tilde{w},n}$	Output tile
b_k	Filters assigned to each thread block
b_n	Input tiles assigned to each thread block
b_c	Channels loaded in each iteration

Table 3. Summary of notations. $\tilde{h}$ is computed as $\lceil h/2 \rceil$ and $\tilde{w}$ is computed as $\lceil w/2 \rceil$ (h and w are indexed from 1).

[4]The 1-bit yield flag is embedded in each instruction to balance the workload on each warp scheduler [5]. When this flag is set, the scheduler prefers to issue the next instruction from the current warp. When the bit is cleared, the scheduler prefers to switch to another warp. This costs one extra cycle to switch to another warp.

3.1 Workflow Overview

The 2D batched 3×3 convolution can be written as:

$$O_{k,h,w,n} = \sum_{r=1}^{R} \sum_{s=1}^{S} \sum_{c=1}^{C} I_{c,h+r,w+s,n} \times F_{c,r,s,k} \tag{4}$$

The equivalent 2D batched $F(2 \times 2, 3 \times 3)$ Winograd convolution can be written in the following steps:

Filter transform (FTF) for each 3×3 filter tile:

$$\hat{F}_{c,k} = G F_{c,k} G^T \tag{5}$$

Input transform (ITF) for each 4×4 input tile:

$$\hat{I}_{c,\tilde{h},\tilde{w},n} = B^T I_{c,\tilde{h},\tilde{w},n} B \tag{6}$$

Element-wise multiply (EWMM) and accumulate along channels c (also called **batched matrix multiplication** step):

$$\hat{O}_{k,\tilde{h},\tilde{w},n} = \sum_{c=1}^{C} \hat{I}_{c,\tilde{h},\tilde{w},n} \odot \hat{F}_{c,k} \tag{7}$$

Output transform (OTF) for each output tile:

$$O_{k,\tilde{h},\tilde{w},n} = A^T \hat{O}_{k,\tilde{h},\tilde{w},n} A \tag{8}$$

We use a separate kernel to transform the filter. The input transform (ITF) and element-wise multiplication (EWMM) steps form the **main loop**. After the main loop, we will transform output, with shared memory as buffer to transpose the data.

3.2 Workload Mapping

In the EWMM step (Equation (7)), $\lceil H/2 \rceil \lceil W/2 \rceil N \times K \times C$ of 4×4 EWMMs and accumulation along C will be computed.

Two-level cache blocking. Since the fast memory (shared memory, registers) on GPUs are relatively small, we adopt cache blocking strategy [10] to maximize data reuse.

Following the practice in previous works [25, 27], we adopt two-level blocking strategy. In each iteration, a thread block will load $b_k \times b_c$ of filter tiles and $b_n \times b_c$ of input tiles. And as Figure 1 shows, each thread block will compute $b_k \times b_n$ of 2×2 output tiles. After the transformation, each thread will load 2 of (transformed input and filter) 8 float elements fragment to do matrix multiplication. The performance is sensitive to cache block size. We illustrate how we choose cache block size in the next subsection.

3.3 Choosing Cache Block Size

In cuDNN [1] and Neon [16], they choose cache block size as follows: $b_k = 32$, $b_n = 32$, i.e., each thread block computes 32×32 output tiles. Having observed that the number of filters (K) for all convolutional layers on many recent CNN models, including VGG and ResNet, is a multiple of 64, we adopt a more aggressive cache block size: $b_k = 64$, $b_n = 32$, $b_c = 8$. Since input data needs to be loaded and transformed K/b_k times, doubling the b_k can reduce the times of loading input data and performing input transform by half.

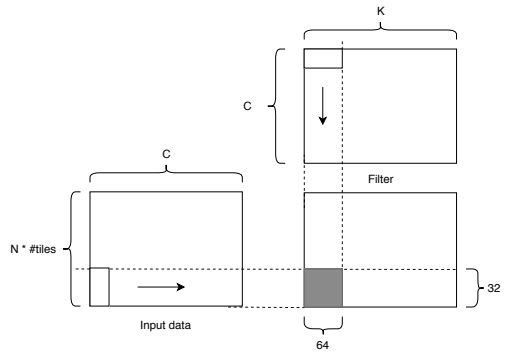

Figure 1. Workload mapping overview. We adopt the cache blocking strategy. Each thread block will compute $b_k \times b_n$ of output tiles. The grey block represents the output area that one thread block is responsible for.

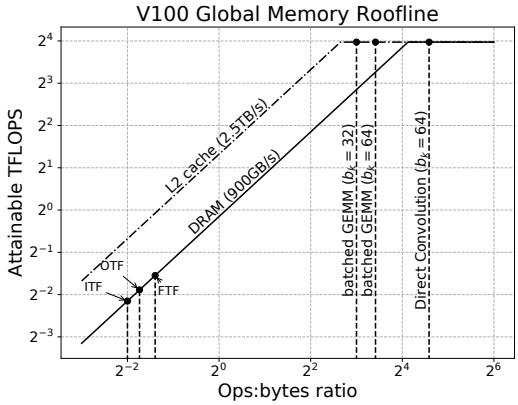

Figure 2. Roofline model of different steps of the Winograd convolution on V100 (peak FP32 FLOPS: 15.7T). Input, filter and output transform steps are memory-bound. Increasing the cache block size can increase arithmetic intensity.

As Figure 2 shows, the input transform (ITF), filter transform (FTF) and output transform (OTF) steps are memory-bound. Even the batched GEMM step also requires a certain level of L2 hit rate to keep the math pipe busy. Increasing the cache block size b_k from 32 to 64 can increase the arithmetic intensity from 8 ops/byte to 10.67 ops/byte (+33%), making the implementation more robust to L2 cache miss.

3.4 Software Pipelining

LDG (load data from global) instruction has a latency up to more than 1000 cycles (L2 cache miss + TLB miss) [5, 13, 14]. Hiding global load latency is the most important consideration in many applications. We hide the long global memory access latency by software pipelining.

32 registers are used to hold prefetched 2 filter tiles and 16 registers are needed to hold prefetched one input tile in our implementation.

The latency of shared memory loading (LDS) is around 20 cycles and can grow to hundreds of cycles when the load/store units are busy [5]. We also hide the latency of LDS with software pipelining. $4 \times 8 = 32$ registers are used to hold the data to do matrix multiplication for the next iteration.

3.5 Implicit Zero-Padding

We implicitly do zero-padding by masking LDG instructions with predicate mask[5]. Each of predicate registers stores one bool value. Since each thread will always load input tile at the same location (same h, w), we can precompute the zero-padding mask.

We need 16 bool values to mask one 4×4 input tile. However, the hardware only provides 7 predicate registers for each thread [5]. The NVCC compiler will choose to store one bool value in one regular register. This strategy leads to register spilling since the total register requirement exceeds 255. We leverage *P2R* instruction to pack 16 predicates to one regular register before the main loop and unpack the register inside the loop with *R2P* to avoid register spilling.

4 Implementation Detail

In this section, we describe the implementation of each step in detail. We also introduce our optimization techniques in this part. All techniques in this section can be applied at CUDA C++ level except for register allocation. SASS level optimizations are discussed in Section 5 and 6.

In each thread block, 256 threads cooperate to compute $b_k \times b_n = 2048$ of 2×2 output tiles. In each iteration, each thread block will load $b_k \times b_c = 512$ of filter tiles and $b_n \times b_c = 256$ of input tiles, and perform element-wise multiplication and accumulation on them.

We show how our implementation works in Algorithm 1. We omit details including software pipelining, barrier synchronization and index calculation for brevity. Line 6 to 16 is the main loop.

Algorithm 1: Simplified workflow of our Winograd convolution. Fragments reside in registers. We configure $b_c = 8$, $b_n = 32$, $b_k = 64$.

```
1  __shared__ input_smem[16][b_c][b_n];
2  __shared__ filter_smem[16][b_c][b_k];
3  input_frag[2][8];
4  filter_frag[2][8];
5  accumulator[2][64];
6  for iter←0 to C by b_c do
7      filter_smem ← b_k × b_c of transformed filter tiles;
8      input_smem ← b_n × b_c of transformed input tiles;
9      for i←0 to b_c do
10         filter_frag ← 2 × 8 elements from filter_smem;
11         input_frag ← 2 × 8 elements from input_smem;
12         foreach element in accumulator do
13             accumulator[][] ← accumulator[][] +
                input_frag[][]× filter_frag[][];
14         end
15     end
16 end
17 Transpose and transform accumulated result;
18 Store result to global memory;
```

4.1 Filter Transform

We implement the filter transformation in a separate kernel (called *FX* variant in [6, 11]). Since the filter is usually much smaller than the input, this step only contributes to a small fraction of the total running time.

Each thread block will load $b_k \times b_c = 64 \times 8 = 512$ filter tiles in each iteration. And each thread will load $512/256 = 2$ tiles. Threads within a warp will load filter of continuous k. Since the transformed filters are stored in *CR'S'K* layout, the global memory access is fully coalesced. 32KB (512×4B×4×4) shared memory is used to store the transformed filter.

[5]E.g., `@P1 LDG R0, [R2];` will only load data to `R0` when `P1` is true.

4.2 Input Transform

In each iteration, $b_n \times b_c = 32 \times 8 = 256$ input tiles are loaded and transformed (line 8 in Algorihtm 1). Each thread will load $256/256 = 1$ input tile. Threads within a warp will load input of continuous batches. The *CHWN* layout makes the loading fully coalesced. 16KB shared memory is used to store the transformed input data.

Each thread uses 32 FADDs to transform a tile. The 32 FADDs add $32/1024 = 3.1\%$ more pressure to the float pipe (As we will show in the later section, 1024 FFMAs are used in each thread in the EWMM step).

4.3 Batched Matrix Multiply

The EWMM step (line 9 to 15 in Algorithm 1) is where most of the computation happens. In this step, each thread block computes 16-batched $64 \times 32 \times 8$ GEMM. Each thread computes two $8 \times 8 \times 8$ GEMM with 1024 FFMAs.

Since tiles of different channels are scattered in different threads, we need to transpose the data first. The data transposing buffer is arranged as $(16, 8, 64)$ for filter data and $(16, 8, 32)$ for input data (Table 4) to make both store-to and load-from the shared memory bank conflict-free.

To make the discussion easier, we simply the notations as: μ is the local (private to each thread block, range from 1 to 64) filter tile index, and ν is the local input tile index (range

from 1 to 32). We can then write the EWMM step as:

$$\hat{O}_{\mu,\nu} = \sum_{c=1}^{C} \hat{I}_{c,\nu} \odot \hat{F}_{c,\mu}. \tag{9}$$

We can rewrite the accumulation for each *element* in the tile:

$$\hat{O}_{\mu,\nu}^{(x,y)} = \sum_{c=1}^{C} \hat{I}_{c,\nu}^{(x,y)} \times \hat{F}_{c,\mu}^{(x,y)}, \tag{10}$$

where $\hat{O}_{\mu,\nu}^{(x,y)}$ represents the element at location (x, y) of tile $\hat{O}_{\mu,\nu}$. And the accumulation of *different elements* in each tile is *independent* of each other.

Element-wise multiplication to batched matrix multiplication. Since the accumulation on each element (Equation (10)) is independent of each other, we can perform equivalent 16-batched $\hat{O}_{\mu,\nu} = \sum_{c=1}^{C} \hat{I}_{c,\nu} \times \hat{F}_{c,\mu}$ matrix multiplication.

Doing batched matrix multiplication can increase computation intensity. The computation intensity to compute a 4×4 element-wise multiplication is $(16 \times 2)/(32 \times 4) = 0.25(ops/bytes)$, and shared memory is not fast enough to feed the data. However, if we do matrix multiplication and let each thread compute two $8 \times 8 \times b_c$ matrix multiplication, the computation intensity is now $2(ops/bytes)$.

Thread arrangement. The $64 \times 32 \times 8$ GEMM (workload of a warp) is split to 32 of $8 \times 8 \times 8$ GEMM (workload of a thread) and dispatched to 32 lanes as Figure 3 shows. The arrangement decides how to compute the shared memory access offset (line 10, 11 in Algorithm 1) based on lane ID.

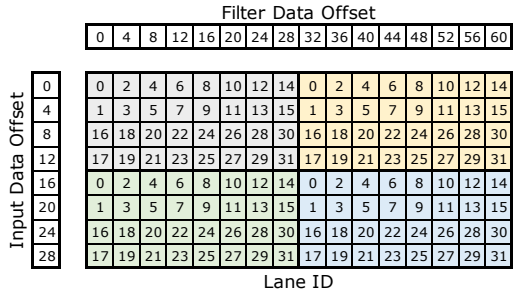

Figure 3. Lane ID arrangement. Input data and filter data offset stands for offset in element (4 bytes). For example, *lane*0 will load filter at location 0,1,2,3 (128bits) with one LDS.128. And *lane*1 will load input data at location 4,5,6,7 (128bits) with one LDS.128.

The arrangement in Figure 3 is the only pattern we find so far to eliminate shared memory bank conflict for LDS.128. The previous belief that "*a shared memory request for a warp does not generate a bank conflict between two threads that access any address within the same 32-bit word*" [18], is not complete. According to this belief, other patterns should also be bank conflict-free since the data is expected to broadcast to all threads. However, the profiling results show other patterns do lead to bank conflict.

Register allocation. In each iteration, each thread will compute two $8 \times 8 \times 8$ GEMM (line 9 to 14 in Algorithm 1). 2×64 registers are used as accumulators, 2×8 registers are used to hold input, and 2×8 are for filter data. The shared memory latency is hidden by software pipelining, and $2 \times (8 + 8)$ registers are needed to hold data in the next loop (Figure 4).

Also, the allocation needs to fulfill the following requirements to maximize performance: (i) Destination of LDS.128 must be a 128-bit vector register (4 continuous registers, starting from a multiple of 4, *e.g.*, *R0, R1, R2, R3*); (ii) FFMA sequence to be register bank conflict[6] free. Our allocation can fulfill these requirements, and is depicted in Figure 4.

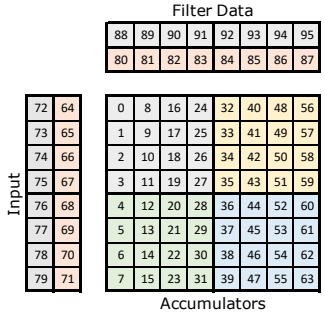

Figure 4. Register allocation of the EWMM step. This corresponds to the declaration in line 3 to 5 in Algorithm 1. Number in the cell is register index. Odd registers reside in one bank and even registers reside in the other bank.

Thanks to the wider 64-bit register bank (Section 5.2.2), the register bank conflict can be eliminated easier compared with previous architectures [9, 27]. We propose the following way to avoid register bank conflict:

1. For even columns (indexed from 0) of the accumulators, start with the odd row (indexed from 0), reuse the filter register, then compute the even row. (*e.g.*, FFMA R1, R65, R80.reuse, R1; FFMA R0, R64, R80, R0;)
2. For odd columns of the accumulators, start with the even row, reuse the filter register, then compute the odd row. (*e.g.*, FFMA R8, R64, R81.reuse, R8; FFMA R9, R65, R81, R9;)

4.4 Output Transform

After the accumulation, we have the pre-transform output data $\hat{O}$ in registers. Since elements of a tile are scattered over different warps, we need to transpose the data to do the final output transform. There are 128KB of $\hat{O}$ in registers, while shared memory on Turing GPUs can be configured up to

[6] If all three source registers are odd or are even, register bank conflict occurs and the FFMA will occupy the float pipe for one more cycle.

64KB [17]. So we do the output transform in 4 *rounds*. In each round, 1/4 of $\hat{O}$(32KB) will be transposed and transformed.

We use padding to avoid shared memory store bank conflict. The layout of the buffer is depicted in Figure 5.

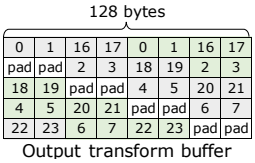

Figure 5. Output transform buffer. Number in the cell is the laneID. For example, $lane0 \sim lane7$ will store output element of 32 continuous batch to shared memory on different banks.

4.5 Summary

We summarize the data layout in Table 4 and register usage in Table 5.

Data Layout. We use 32KB shared memory to store filter tiles, 16KB shared memory to store input tiles. In the output transform step, we reuse the shared memory allocated for filter and input tiles. 40KB shared memory is used as a buffer to transpose the output data.

Variable	Layout	Value	Location
Input	(C,H,W,N)	(C,H,W,N)	GMEM
Filter	(C,R,S,K)	(C,3,3,K)	GMEM
Transformed filter	(C,R',S',K)	(C,4,4,K)	GMEM
Local input buffer	(16, b_c, b_n)	(16,8,32)	SMEM
Local filter buffer	(16, b_c, b_k)	(16,8,64)	SMEM
Local output buffer	(16, 2, 8, b'_n)	(16,2,8,40)	SMEM
Output	(K,H,W,N)	(K,H,W,N)	GMEM

Table 4. Data layout in global memory (GMEM) and shared memory (SMEM), where 16 represents 16 elements in a 4×4 tile, and b'_n represents b_n with 8 padding elements.

Register Usage. We keep the registers for the main loop under 255 to avoid register spilling. The register usage is listed in Table 5.

Usage	#Registers
Accumulators	128
Data from SMEM to do outer product	32
Prefetch data from SMEM	32
Prefetch filter from GMEM	32
Prefetch input from GMEM	16
Filter data pointer	2
Input data pointer	2
SMEM filter/input read offset	2
SMEM filter/input write offset	2
Zero-padding mask	1
Current iteration	1
Input transform workspace	3
Total	253

Table 5. Number of registers for the main loop.

5 Native Assembly Code Programming on Volta and Turing

The needs for P2R/R2P instructions and the temptation of manually scheduling instructions drive us to develop the SASS assembler, TuringAs.

We document instruction encoding, hardware details and other key components in this section.

5.1 ISA Encoding on Volta and Turing

A typical SASS instruction is specified as

```
@P1  LDG R0, [R2];                (11)
```

where P1 is the predicate mask, i.e., only when P1 is true will the instruction be executed. Unlike the pre-Volta architectures employing 64-bit instructions, both Volta and Turing use 128 bits to encode an instruction with an embedded control logic. Figure 6 shows the typical instruction format consisting of four components: (1) Opcode, (2) Operands, (3) Flags, and (4) Control code. We next explain them in detail.

5.1.1 Opcode

Contrary to the previous belief [5] that Volta and Turing use various bit lengths to encode opcode, we believe that the opcode is 12-bit. Examples include FFMA(0x223), FADD(0x221), LDG(0x381), and LDS(0x984).

5.1.2 Operands

An operand can be a regular register, a predicate register, constant memory, or an immediate value.

1. **Regular register.** The 32-bit regular register is indexed by 8 bits. Each thread can access 32-bit registers ranging from R0 to R254. Zero register (RZ) is indexed by 0xff.
2. **Predicate register.** Each thread can access 7 predicate registers, indexed by 4 bits. 0xf is the true predicate register (PT). Instructions like ISETP and R2P can set the value of predicate registers. Carry-in information is stored in predicate registers. The indices of predicate registers are encoded at different places in a regular register, usually at [25:17].
3. **Immediate.** Volta and Turing use 32-bit immediate, which can be used to represent a float or an integer,

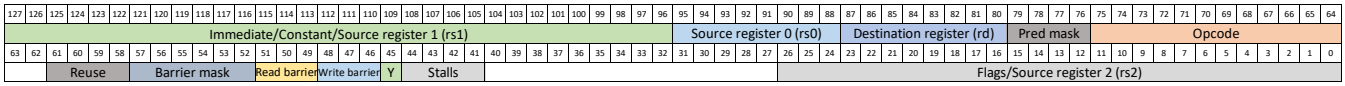

Figure 6. Instruction encoding on Nvidia Volta and Turing. White parts are left unused and are filled with 0.

whereas the pre-Volta architectures use 24-bit immediate.

4. **Constant memory.** Many instructions accept constant memory (e.g., `c[0x0][0x160]`) as one of the operands. Parameters passed to CUDA kernels are stored in constant memory. Other information like `gridDim` is also stored in constant memory.

5.1.3 Flags

Instructions usually specify flags (also known as funct in some literature) to modify its behavior. For example, `LDG` can change its width with `.16`, `.32`, `.64`, and `.128` flag, and `SHF` (funnel shift) can choose to shift left or right with `.L` or `.R` flag. The flag information is usually encoded at [26:0].

5.1.4 Control Code

An interesting feature of Nvidia GPUs is that it is the programmer's/compiler's responsibility to prevent data hazards. For fixed-latency instructions like `FFMA` and `IADD3`, the compiler just needs to stall this instruction for certain cycles if the next instruction reads its output. For variable-latency instructions like `LDG` and `STG`, the compiler will associate the instruction with a (read) barrier, and the instructions which rely on its output, will wait on that barrier.

The aforementioned mechanism is supported by the control code. Control code stores information to prevent data hazards, control reuse flag, and balance progress between warps. A detailed introduction of the control code can be found in Section 2.1 in [5]. We give a detailed description of the yield flag (at [45]), since we found this flag will affect the overall performance.

The yield flag. Multiple warps may reside on one warp scheduler concurrently. To balance the progress of different warps on the same warp scheduler, a one-bit yield flag is used. When the yield flag is set to 1, the warp scheduler prefers to issuing the next instruction from the current warp. Otherwise, the warp scheduler prefers to issuing the next instruction from other warps, but this will take one more clock cycle and disable the register reuse cache. Currently, the *NVCC* compiler seems to simply set the yield flag to 0 every 7 instructions. We have shown that this strategy may hurt performance for certain applications.

5.2 GPU Hardware

5.2.1 Resource Limitations on GPU Device

On Volta and Turing, each thread can use up to 255 32-bit regular registers[7], indexed by 8 bits.

There are 7 predicate registers (P0-P6) for each thread, indexed by 4 bits. Each predicate register stores a bool value. Carry-in information also occupies a predicate register.

Each thread has 6 wait barriers to prevent data hazard for instructions with variable latency like LDG.

5.2.2 Register Banks

Pre-Volta architectures have four 32-bit register banks. If two source registers fall in the same bank, register bank conflict will occur. The instruction will occupy the pipe for one more cycle.

The four 32-bit register banks have been replaced by two 64-bit register banks in Volta and Turing [5], with odd indexed registers reside in one bank and even indexed registers in the other bank. The wide 64-bit register bank makes the register bank conflict less likely to happen.

5.3 TuringAs Implementation

We have implemented TuringAs in 1,400 lines of Python code. TuringAs is a lightweight assembler using built-in Python libraries. Our current implementation supports an essential subset of instructions for linear algebra routines. Our design is extensible and is easy to add support for additional instructions.

TuringAs supports features like inline Python code, which we use to print the long sequence unrolled SASS loop, and register name mapping, which allows us to use a meaningful register name (e.g., `index`) rather than a register index (e.g., `R1`). TuringAs accepts the SASS source file as input and generates `.cubin` files. The `.cubin` file can be loaded with CUDA runtime APIs.

6 Assembly-Level Optimizations

In this section, we discuss some optimizations that can only be applied at SASS level and evaluate their effects. The reported throughput is the average of 10 repeated experiments on an RTX2070. CUDA C code is compiled with *NVCC* 10.1.

[7] In our experiment, the number of registers must be smaller than 253; otherwise, the hardware will not recognize the instruction.

6.1 Load Balancing with Yield Flag

At least since Maxwell architecture[15], a 1-bit yield flag is used to balance the load between different warps on the same warp scheduler [5].

By observing the *NVCC*-generated SASS code and cuDNN's SASS code, we speculate that NVCC and cuDNN use the following heuristic to scatter yield flag:

- **NVCC**[8]**:** scatter yield flag every 8 float instructions.
- **cuDNN:** scatter yield flag every 7 float instructions.

We adopt a new **Natural** yield strategy, which is not to scatter yield flag at all. Tests show that the **Natural** strategy achieves 1.09× speedup for the main loop over NVCC's strategy and 1.11× speedup over cuDNN's strategy. We show the throughput of the main loop under different yield strategies in Figure 7.

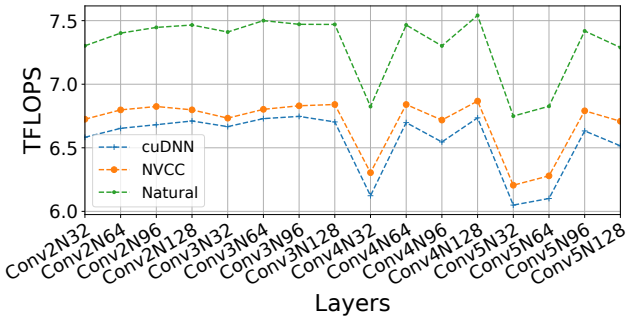

Figure 7. Throughput of the main loop on different layers with different yield strategies.

The yield flag can hurt performance in two ways. First, the yield flag takes one more cycle to switch to another warp [5]. Second, the yield flag will disable the reuse flag of the current instruction and may lead to register bank conflict.

6.2 Scheduling Load/Store Instructions

Apart from FFMAs, load/store instructions are another important part of the implementation. We interleave load/store instructions with FFMAs to not overwhelm load/store unit.

Global memory access. The cuDNN's Winograd implementation interleaves LDG with 2 FFMAs (4 cycles). Rather, we interleave LDG with 8 FFMAs. This can contribute to 1.24× speedup. The throughput of different LDG scheduling strategies is shown in Figure 8.

Shared memory access. By checking the NVCC-generated assembly code, we speculate that the *NVCC* compiler and cuDNN use a heuristic to interleave STS with 2 FFMAs (4 cycles). Rather, we increase the distance between consecutive STS instruction from 2 FFMAs to 6 FFMAs. And this contributes to 2% of higher throughput of the main loop. Throughputs of different STS scheduling strategies are shown in Figure 9.

[8]Their heuristic is more complex than this, but this is enough to illustrate the effect of the yield flag.

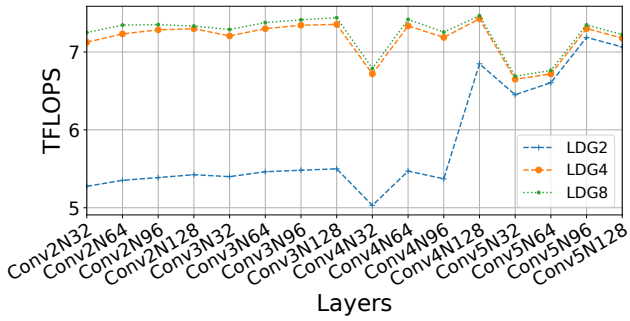

Figure 8. Throughput of the main loop on different layers with different LDG scheduling strategies. *LDGn* represents to interleave LDGs with *n* FFMAs.

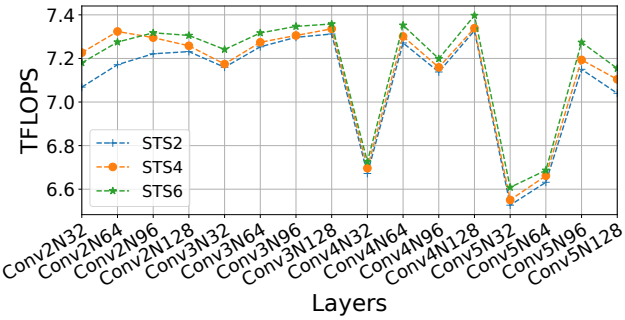

Figure 9. Throughput of the main loop on different layers with different STS scheduling strategies. *STSn* represents to interleave STSs with *n* FFMAs.

7 Evaluation

In this section, we evaluate our optimized Winograd convolution on Volta V100 and Turing RTX2070 GPUs on all 3×3 convolution layers in ResNet. Parameters of different layers are listed in Table 1. Kernel running time is collected using *CUDA event* [19] and the reported running time is the average of 20 times of measurement. We compared the performance of our implementation against Winograd convolution[9] and other algorithms of cuDNN 7.6.1, which was released in June 2019, with *NCHW* data layout.

7.1 Compare with cuDNN's Winograd Convolution

The speedups of our implementation over cuDNN's Winograd convolution are shown in Table 6. On RTX2070, we see up to 2.65× and on average 1.95× speedup. On V100 we see up to 2.13× and on average 1.5× speedup.

On both devices, the speedups on *Conv5* are significantly better than other layers. This is because the *Conv5* layer

[9]CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD in cuDNN

Device	N	Layers			
		Conv2	Conv3	Conv4	Conv5
RTX2070	32	1.67×	1.85×	1.73×	2.59×
	64	1.65×	1.83×	1.79×	2.47×
	96	1.68×	1.83×	1.74×	2.65×
	128	1.67×	1.82×	1.77×	2.57×
V100	32	1.32×	1.42×	1.31×	1.95×
	64	1.24×	1.40×	1.41×	1.77×
	96	1.24×	1.38×	1.34×	2.13×
	128	1.23×	1.38×	1.38×	1.97×

Table 6. Speedup over cuDNN's Winograd convolution.

has the greatest number of filters ($K = 512$), making the overfetch of input data a more serious problem. Our implementation has a larger b_k and is less vulnerable to the large filter size.

The speedups on RTX2070 are higher than the speedups on V100. The main reason is that the occupancy on V100 is twice as the occupancy on RTX2070. The shared memory of V100 can be configured to 96KB, while the shared memory on RTX2070 (and other Turing GPUs) is limited to 64KB [17]. cuDNN's Winograd convolution needs 48KB shared memory per block (Table 7). Each SM can hold 2 thread blocks on V100 but only 1 on RTX2070. More concurrent thread blocks give the warp scheduler chance to switch to other warps to hide latency, and thus increase performance.

Parameters	Ours	cuDNN's
(b_k, b_n, b_c)	(64, 32, 8)	(32, 32, 8)
Threads per block	256	256
SMEM per block	48KB	48KB
Registers per thread	253	126
Registers per block	64768	32256

Table 7. Parameters of our implementation and cuDNN 7.6.1's Winograd convolution.

7.2 Percentage of Peak

We use the Speed Of Light (SOL, SM[%]) value to represent the percentage of peak achieved by this implementation. The SOL value is the "*achieved percentage of utilization with respect to the theoretical maximum*", reported by the *Nsight Compute* [20] profiler.

We give two SOL values. One is the SOL of the whole program, except for filter transformation (labeled with *Total*). The other is the SOL of the main loop (labeled with *Main loop*). Since we cannot mix the compute-bound main loop and the memory-bound output transform, the SOL of the whole program is smaller than the SOL of the main loop.

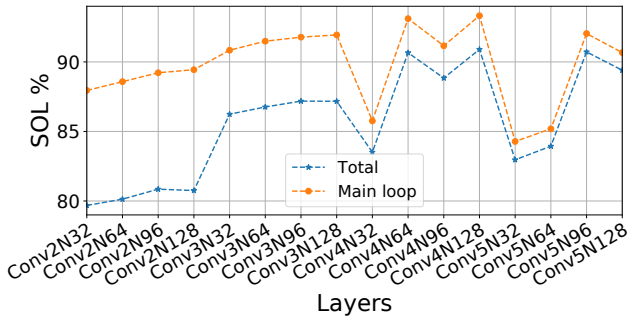

Figure 10. Speed of Light (SOL) on RTX2070. The SOL value represents the achieved percentage of utilization to the theoretical peak.

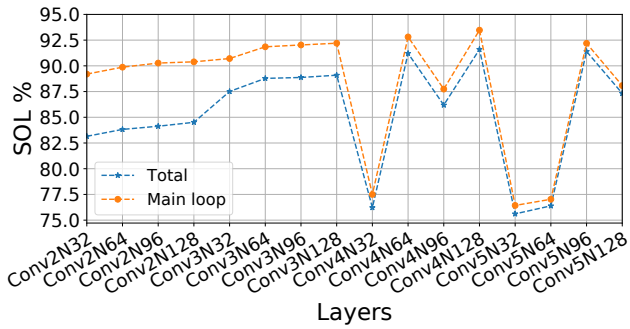

Figure 11. Speed of Light (SOL) on V100.

On both devices, the SOL of the main loop can be above 87.5% and up to 93% for large batch size. And the SOL of the whole program can be above 90%.

For layers like *Conv4N32* and *Conv5N32*, there is a drop in the SOL value. This is because there are not enough thread blocks to keep the GPU busy. If we increase the batch size, the SOL will increase dramatically.

7.3 Compare with Other Algorithms

We compare our Winograd convolution with all other convolution algorithms[10] in cuDNN. The speedups on RTX2070 and V100 are shown in Figure 12 and Figure 13, respectively. The workspace required by different algorithms are listed in Figure 14. Our implementation only needs a small workspace to hold $16KC$ transformed filter data (0.25MB for *Conv2*, 1MB for *Conv3*, 4MB for *Conv4*, 16MB for *Conv5*). We have the following observations:

1. Compared with GEMM-based convolution (the IMPLICIT PRECOMP version), our implementation achieves

[10]IMPLICIT_PRECOMP_GEMM computes convolution by doing matrix multiplication (GEMM) implicitly. WINOGRAD_NONFUSED computes convolution with the Winograd algorithm. Compared with the fused version, which stores the intermediate result in shared memory, the non-fused version stores the intermediate result in global memory.

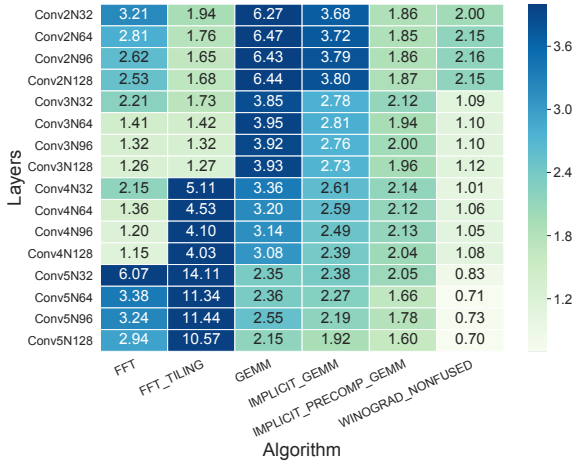

Figure 12. Speedup over all other algorithms on RTX2070.

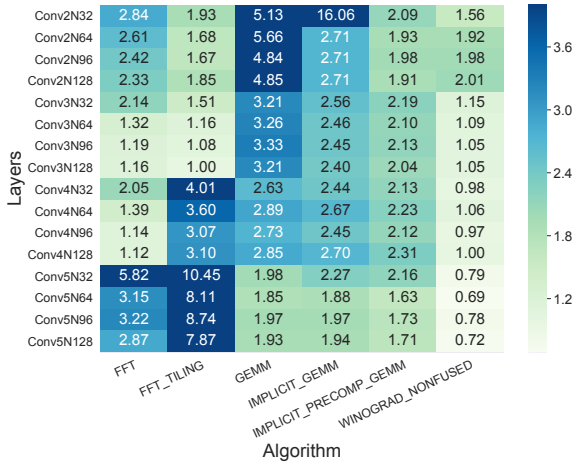

Figure 13. Speedup over all other algorithms on V100.

1.6× to 2.31×, and on average 1.99× speedup, which is close to the 2.25× multiplication reduction.

2. For the *Conv5* layer, the speedup over GEMM-based convolution is smaller. This is because the size of the input is 7×7, and the $F(2\times2, 3\times3)$ Winograd computes one more pixel, which will be discarded later.
3. For the *Conv2* layer, our Winograd convolution is at least 1.56× faster than all other algorithms in cuDNN on all layers on both devices and consumes little (0.25MB) global memory as workspace.
4. For the *Conv3* layer, our implementation is 5% to 15% faster than the non-fused Winograd convolution in cuDNN. FFT-based convolution also gives good performance on this layer, but not as fast as ours.
5. For the *Conv4* layer, the performance of our implementation is comparable with the non-fused version

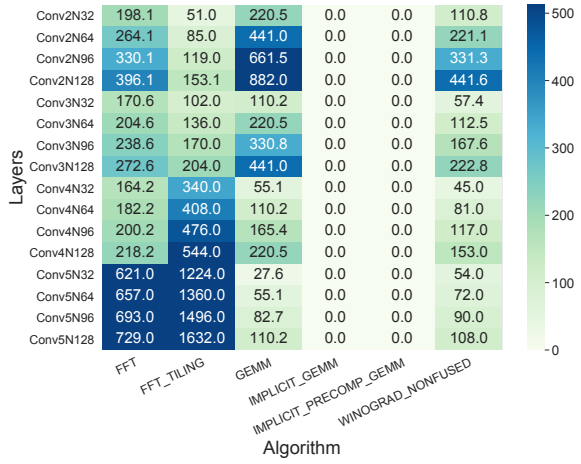

Figure 14. Workspace (MB) required by different algorithms in cuDNN.

(with smaller workspace) and faster than all other algorithms. Moreover, compared with the non-fused version, our implementation requires fewer requests to GPU's DRAM, which reduces overall power consumption.

6. For the *Conv5* layer, our performance is considerably faster than all other algorithms but slower than the non-fused version. This is because the non-fused version uses $F(4\times4, 3\times3)$ Winograd, which reduces the number of multiplication by a factor of 4 [11]. The input and output of this layer are relatively small. The benefit of more reduction in multiplication outweighs the time to store (and load) transformed data to (from) global memory at this layer.

8 Discussion

8.1 Fused or Non-fused Winograd Convolution

For 3×3 convolutional layers, $F(2\times2, 3\times3)$ and $F(4\times4, 3\times3)$ Winograd are popular options. Other variants like $F(6\times6, 3\times3)$ may bring numerical issue and require considerably large workspace for intermediate result. Usually, the fused version adopts $F(2\times2, 3\times3)$ variant (in this work and cuDNN's fused Winograd) and non-fused implementations apply the $F(4\times4, 3\times3)$ variant. We analyze which one will be faster under different conditions.

For the fused $F(2\times2, 3\times3)$ version, we assume the data loading time can be hidden by computation and ignore data transformation time for brevity, thus the total time for fused $F(2\times2, 3\times3)$ is

$$\frac{2NCHWKRS}{2.25\text{FLOPS}},$$

where $R = 3$ and $S = 3$ are the filter height and width.

For the non-fused $F(4\times4, 3\times3)$ version, the data transformation steps are memory-bound, and the size of transformed

input is $(6 \times 6)/(4 \times 4) = 2.25$ times of the original input, thus the total running time can be computed as

$$\frac{2NCHWKRS}{\text{4FLOPS}} + \frac{NCHW \times (1 + 2.25) \times 2 \times 4 \text{ Bytes}}{\text{DRAM Bandwidth}}.$$

By substituting the FLOPS and bandwidth data of V100 and RTX2070, we find the break-even point for V100 is $K = 129$ (when $K < 129$, fused $F(2 \times 2, 3 \times 3)$ is faster, and when $K > 129$, non-fused $F(4 \times 4, 3 \times 3)$ is faster), and the break-even point for RTX2070 is $K = 127$. These analytical results are in accordance with our evaluation results in Figure 12 and 13.

We expect greater speedup in the future if the fused $F(4 \times 4, 3 \times 3)$ is well optimized.

8.2 Integrate with Compiler

To achieve comparable performance at a higher level rather than SASS can increase productivity. We make the following suggestions to help the compiler generate better code.

Expose P2R and R2P instructions at PTX level. The P2R and R2P instructions can pack and unpack multiple predicate registers, thus save registers. They can also help to save instructions. Besides, we notice that this pair of instructions exist in all architectures since Fermi.

New algorithm to scatter the yield flag. In Section 6.1, we have shown that changing the strategy of scattering yield flag alone can increase the performance by 10%. To look into its mechanism and how to set yield flag under different conditions would be valuable.

Increase space between load/store instructions. Current space between continuous load/store instructions is not enough. The program may be stalled by busy load/store units. Besides, the width of memory accesses is known at compile time. Such information can help the compiler to interleave load/store instructions of different width with different space.

8.3 Generality of This Work

Our implementation will achieve maximum performance when N is a multiple of 32, K is a multiple of 64 and C is a multiple of 8, which are common cases for many widely used CNNs [4, 24].

The implementation can be ported to the *fp16* version by increasing b_n to 64. To further increase the throughput with newly introduced *tensor core*, the data layout needs a redesign. Nevertheless, many techniques introduced in this work, like large cache block size and load balancing between warps, can be adopted. These techniques can also be applied to other dense linear algebra routines.

8.4 For Other Data Layout

The implementation in this work can be ported to NCHW layout with little effort. For example, each thread block can load and transform a 16×8 input tile (32 of 2×2 tiles) to make the global load fully coalesced. The offsets of global and shared memory accesses need to be recomputed, while all other optimizations can be adopted.

9 Related Work

There are other works focusing on optimizing Winograd convolution. Zhen *et al.* optimized Winograd convolution on manycore CPUs [6]. Scott implemented Winograd convolution in SASS for Maxwell and Pascal GPUs [11, 15].

The other strategy used to implement Winograd convolution is to store intermediate results in global memory (non-fused version). It is easier to implement because it can utilize optimized batched matrix multiplication routines. However, it needs significant amount of global memory as workspace and data loading can be the new bottleneck.

Other than Winograd convolution, researchers have made different efforts to reduce the CNN training time, including: **1.** To optimize direct convolution on CPUs [2] and GPUs [7]. **2.** To express convolution as matrix multiplication [1] to utilize the highly optimized matrix multiplication routines. **3.** To use the FFT approach to compute the convolution [12]. Compared with Winograd convolution, FFT-based convolution performs better at large filter size [11], while small filters like 3×3 are more popular in today's CNNs.

10 Conclusion

In this work, we have presented a solution to optimize the performance of single-precision $F(2 \times 2, 3 \times 3)$ Winograd convolution on NVIDIA Volta and Turing GPUs.

Apart from the high-level optimizations, we also build a SASS assembler for NVIDIA Volta and Turing GPUs to tune the performance at SASS level and propose new insights to increase the performance. We make the assembler publicly available to inspire more works in this area.

Acknowledgments

We thank Andrew Lavin for the advice in SASS programming, and the anonymous reviewers for their feedbacks that help improve the qualitiy of this work. This research is supported by HK Research Grants Council under Grant No. 26213818.

References

[1] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *CoRR* abs/1410.0759 (2014), 1–9.

[2] Evangelos Georganas, Sasikanth Avancha, Kunal Banerjee, Dhiraj D. Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. 2018. Anatomy of high-performance deep learning convolutions on SIMD architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018*. IEEE/ACM, Dallas, TX, USA, 66:1–66:12.

[3] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming

He. 2017. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *CoRR* abs/1706.02677 (2017), 1–12.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016*. IEEE Computer Society, Las Vegas, NV, USA, 770–778.

[5] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele Paolo Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *CoRR* abs/1804.06826 (2018), 1–66.

[6] Zhen Jia, Aleksandar Zlateski, Frédo Durand, and Kai Li. 2018. Optimizing N-dimensional, winograd-based convolution for manycore CPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018*. ACM, Vienna, Austria, 109–123.

[7] Alex Krizhevsky. 2015. cuda-convnet2. Retrieved Jan 12, 2019 from https://github.com/akrizhevsky/cuda-convnet2

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems, NIPS 2012*. NIPS, Lake Tahoe, NV, USA, 1106–1114.

[9] Junjie Lai and André Seznec. 2013. Performance upper bound analysis and optimization of SGEMM on Fermi and Kepler GPUs. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013*. IEEE Computer Society, Shenzhen, China, 4:1–4:10.

[10] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. 1991. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems , ASPLOS 1991*. ACM, Santa Clara, CA, USA, 63–74.

[11] Andrew Lavin and Scott Gray. 2016. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016*. IEEE Computer Society, Las Vegas, NV, USA, 4013–4021.

[12] Michaël Mathieu, Mikael Henaff, and Yann LeCun. 2013. Fast Training of Convolutional Networks through FFTs. *CoRR* abs/1312.5851 (2013), 1–9.

[13] Xinxin Mei and Xiaowen Chu. 2017. Dissecting GPU Memory Hierarchy Through Microbenchmarking. *IEEE TPDS* 28 (2017), 72–86.

[14] Xinxin Mei, Kaiyong Zhao, Chengjian Liu, and Xiaowen Chu. 2014. Benchmarking the memory hierarchy of modern GPUs. In *IFIP International Conference on Network and Parallel Computing*. Springer, Ilan, Taiwan, 144–156.

[15] NervanaSystems. 2016. Maxas. Retrieved Jan 12, 2019 from https://github.com/NervanaSystems/maxas

[16] NervanaSystems. 2016. Neon. Retrieved Jan 12, 2019 from https://github.com/NervanaSystems/neon/tree/master/neon/backends/kernels/sass

[17] NVIDIA. 2018. NVIDIA TURING GPU ARCHITECTURE. Retrieved Jan 12, 2019 from https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[18] NVIDIA. 2019. CUDA C Programming Guide. Retrieved Jul 2, 2019 from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[19] NVIDIA. 2019. How to Implement Performance Metrics in CUDA C/C++. Retrieved Jul 2, 2019 from https://devblogs.nvidia.com/how-implement-performance-metrics-cuda-cc/

[20] NVIDIA. 2019. Nsight Compute. Retrieved Jul 2, 2019 from https://docs.nvidia.com/nsight-compute/NsightCompute/index.html

[21] MLPerf Org. 2019. MLPerf. Retrieved Jul 2, 2019 from https://mlperf.org/

[22] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in Neural Information Processing Systems, NIPS 2015*. NIPS, Montreal, Quebec, Canada, 91–99.

[23] Shaohuai Shi, Qiang Wang, Pengfei Xu, and Xiaowen Chu. 2016. Benchmarking state-of-the-art deep learning software tools. In *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*. IEEE, Macau, China, 99–104.

[24] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *CoRR* abs/1409.1556 (2014), 1–14.

[25] Vasily Volkov and James W. Demmel. 2008. Benchmarking GPUs to Tune Dense Linear Algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC)*. IEEE Press, Piscataway, NJ, USA, 31:1–31:11.

[26] Shmuel Winograd. 1980. *Arithmetic complexity of computations*. Vol. 33. Siam, Salt Lake City, UT, USA.

[27] Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. 2017. Understanding the GPU Microarchitecture to Achieve Bare-Metal Performance Tuning. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2017*. ACM, Austin, TX, USA, 31–43.