Pushing the Limits for 2D Convolution Computation On CUDA-enabled GPUs

Peng Chen[1,3,a)] Mohamed Wahib[2,b)] Shinichiro Takizawa[3,c)] Satoshi Matsuoka[1,3,d)]

Abstract: The 2D convolution operator is the computational bottleneck in a variety of image processing and machine learning applications. We propose an algorithm to compute convolution by employing register files to cache image data (known as register cache), rather than using the user-managed scratch-pad memory. We take advantage of CUDA's warp shuffle functions to accelerate the intra-warp communication of partial results. Unlike the GEMM-based, FFT-based or Winograd method, our algorithm executes the convolution computation without using any GPU memory as a workspace, and is general to all filter shapes. Our algorithm performs better than state-of-the-art 2D convolution implementations. Using a single TitanXp GPU, it is in average 4.7x faster than NPP (Nvidia Performance Primitives), and 1.8x faster than the highly-optimized ArrayFire library.

1. Introduction

Convolution-based operators, namely filtering, are of fundamental importance to a vast range of digital signal and image processing applications. For example, the Gaussian-filter and Laplace-filter are widely used to enhance image visibility. More specifically, Difference of Gaussian (DOG) is commonly applied for Scale-Invariant Feature Transform (SIFT) [1] to detect robust local-features in images. In the domain of artificial intelligence, convolution is also a core technology of Deep Neural Networks (DNN) [2]. Extracting sparse features by multiple convolutions, Convolutional Neural Networks (CNN) [3] achieves outstanding performance in many challenging tasks, such as text indexing, voice recognition, and image classification. Due to the huge amount of input data and high compute intensity, convolutions easily become the bottleneck of the computation in many applications. Hence, it is significant to improve the performance for convolution to meet low latency demand.

Many effective ways have been introduced to speed-up the 2D convolution computations. FFT-based approaches [4] [5] reduce the computation complexity from $O(N^4)$ to $O(N^2 \log^2 N)$ regardless of the variance in filter sizes; it is particularly suitable for convolutions with larger filters. As for some filters with special sizes, Winograd's algorithm [6] can also reduce the computation to some extent. Another popular approach is that of unrolling the input data and filter coefficients to two matrices [7], which are stored by the workspace buffer, then applying the highly optimized General Matrix-Matrix Multiplication (GEMM) [8] function to accelerate the convolution computation. All of those methods have been incorporated as parts of the *cuDNN*'s [9] convolution functions.

With respect to computational efficiency, Graphics Processing Units (GPUs) provide a huge computing capability over conventional CPUs, and have been successfully used in High Performance Computing (HPC) [10] applications. In addition, GPUs are also widely used to accelerate the Deep Learning (DL) workloads [11] [12]. Hence, GPUs, especially CUDA-enabled GPUs, are well-suited many-core accelerators for computing convolutions with high throughput.

According to the memory hierarchy of CUDA architecture [13], caching data by explicitly managed scratch-pad memory is effective in hiding the latency of accessing global memory. Naturally this technology is widely used to optimize the application's performance (including convolution) by many state-of-the-art libraries, such as ArrayFire [14].

In this work, we propose an algorithm that directly computes convolution without relying on scratchpad or cache memory. More specifically, we manually cache data by register files directly, accumulate the partial correlation results, and communicate the partial results via shifting register files with CUDA's *shuffle_up* instruction. Results show that our algorithm achieves the highest efficiency for computing 2D convolution, compared to existing state-of-the-art libraries.

The rest of this paper is organized as follows. In Section 2, we review GPU technology with CUDA. In Section 3, we analyze linear convolution algorithms. Section 4 introduces the proposed parallelized algorithm in detail. Section 5 evaluates the proposed algorithm. Section 6 discusses related works, and finally we conclude in Section 7.

[1] Tokyo Institute of Technology
[2] National Institute of Advanced Industrial Science and Technology
[3] AIST-Tokyo Tech Real World Big-Data Computation Open Innovation Laboratory, National Institute of Advanced Industrial Science and Technology
[a)] chen.p.aa@m.titech.ac.jp
[b)] mohamed.attia@aist.go.jp
[c)] shinichiro.takizawa@aist.go.jp
[d)] matsu@is.titech.ac.jp

2. Parallel Computing with CUDA

2.1 CUDA Architecture

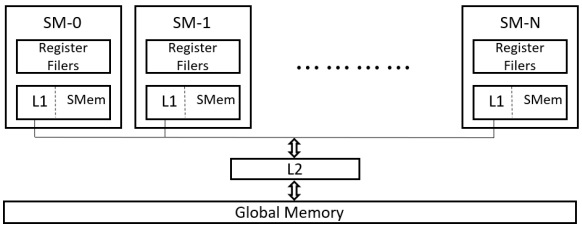


Fig. 1: Streaming Multiprocessors (SM) Architecture On Nvidia GPUs

CUDA (Compute Unified Device Architecture) [15] is built around a scalable array of multi-threaded Streaming Processors (SMs), as shown in Fig. 1. Massive thread-level parallelism is abstracted into a hierarchy of threads running in a SIMT (single instruction multi-thread) fashion [16]. The threads in CUDA are grouped into warps, blocks and grids [17]. Thousands of threads are managed, created, scheduled and executed within SMs efficiently. In comparison to latency-optimized CPUs, GPUs are throughput-optimized processors.

In all generations of CUDA architecture [15], multi-threads execute in groups of threads, namely warp. The total thread count in a warp, called WarpSize, has not changed throughout GPU generations (shown in Equ. 1). Each thread in a warp is assigned an ID, called LaneId, which ranges from 0 to 31 (WarpSize-1).

$$WarpSize = 32 \tag{1}$$

2.2 CUDA Memory Hierarchy

GPUs supports different memory types: global, local, texture, constant, shared and register [17]. Because convolution algorithms tend to be bandwidth bound, it is essential to design and implement algorithms that efficiently utilize this complex memory hierarchy in order to achieve the best performance.

Global memory is a GPU's largest off-chip memory with the highest R/W (Read/Write) latency and throughput. Global memory is typically used to store input and output data. Furthermore, only in a coalesced access fashion (i.e. unit-strided contiguous memory access), can the global memory achieve throughput that approaches the peak memory bandwidth.

Constant memory and texture memory are cache-optimized read-only on-chip memories. Local memory is a private storage for an executing thread, and is invisible outside the scope of the thread.

Shared memory is a fast on-chip scratchpad memory. It shares space with L1 cache as shown in Fig. 1, and its scope is restricted to CUDA threads blocks. Like banks in Dynamic Random Access Memory (DRAM) modules, shared memory is similarly divided into banks (32 banks in modern GPUs). Each bank may process only one R/W request at a time. Bank conflicts results in lower throughput for CUDA kernels [13].

Register files are the fastest on-chip memory available. Registers are private to the executing thread, and could be exchanged within a warp using the *shuffle* intrinsic.

2.3 Register files Vs. Shared memory

Register cache, is an approach in which a single warp builds a virtual cache layer on top of register files as a mechanism for low-latency data R/W [18]. More specifically, the large amount of register memory available in modern GPUs can be used to store some data arrays like the shared memory. To our knowledge, the register cache is difficult to use, mainly due to its scope considerations in addition to the SIMT-fashion execution of the threads in a warp [17].

A SM has up to 96KB available shared memory in modern GPUs, however the aggregate register memory is up to 65535×4B=256KB, which is up to 2.7 times larger than shared memory. Hence, caching data by registers becomes a practical and effective approach to improve many applications' throughput. Furthermore, in recent years, the gap between the total amount of shared memory vs. registers files is becoming larger and larger (Table 1),

Avoiding bank conflicts is an important issue for high performance computation when using shared memory. Accordingly, applications are required to avoid access patterns that can cause bank conflicts. In addition, reducing the use of shared memory releases more L1 cache capacity and contributes to the throughput of CUDA applications [19].

Table 1: Shared Memory and Register Files

GPU	Shared Memory / SM	Max 32-bit Registers / SM	SMs
GTX Titan X	48 KB	65536	24
Tesla M40	96 KB	65536	24
GTX Titan Xp	48 KB	65536	30
Tesla P100	64 KB	65536	56
Tesla V100	Configurable up to 96 KB	65536	80

2.4 Efficient Intra-Warp Communication

Shuffle is an intra-warp communication mechanism for a CUDA-enabled GPU. It allows the exchange of data between threads directly within a single warp. It opens the door, so to speak, for fast data communication between intra-warp threads.

Threads in a kernel can exchange data through global memory. Threads in the same thread block may exchange data through shared memory (plus a synchronization barrier). Threads in the same warp are also able to communicate with each other by shuffle instructions, which is not only fast but also a direct communication method requiring no global memory or shared memory.

Like shared memory, bank conflicts that occur when accessing register files is a challenge to algorithm design. To make matters more complicated, there are few Nvidia official documents about this issue. To our understanding, *shuffle_up* and *shuffle_down*, which perform differently from the *shuffle* instruction, tend to solve the conflict problem easily. This is because threads are orchestrated to request data from the independent bank (or thread) in a single warp.

2.5 Register Spilling

Registers are a limited resource for GPUs. If the required number of registers per thread is too large, the compiler (nvcc) may

spill parts of them to L1 cache (or shared memory). In the case that the L1 cache is insufficient, the compiler may end up spilling them to global memory. Avoiding register spilling to global memory is necessary, or else performance dramatically drops.

3. Computing 2D Convolution

3.1 2D Convolution

Mathematically, convolution combines two linear functions to a third one [20], which measures the correlation of overlap between two functions. Equ. 2 shows the canonical form of a two-dimensional convolution. In the equation, f(x, y) denotes a 2D matrix (or image) with size (W, H), while w(x, y) is a filter with size (M, N), where M=b-a+1, N = d-c+1. The filter is often centrosymmetric, where b=-a, b ≥ 0, d=-c, d ≥ 0. As a special case, a=0, b=0, d=-c, d ≥ 0 means a 1D convolution in Y direction; a ≥ 0, b=-a, d=0, c=0 is a 1D convolution in X direction.

$$f(x,y) * w(x,y) = \sum_{t=c}^{d} \sum_{s=a}^{b} f(x-s, y-t) w(s,t) \qquad (2)$$

Where $*$ is the convolution operator.

3.2 Convolution Theorem and Fourier Transform

The convolution theorem [21] provides an efficient way for computing convolution, particularly with larger filter sizes. As the theorem stated within the fourier transform [22], the convolution operator becomes the point-wise dot product. In other words, the spatial domain convolution equals point-wise inner production in the frequency domain (Equ. 3, 4); ultimately the benefit is that the complexity of the point-wise multiplication is considerably less than original convolution at the price of computing FFT (Fast Fourier Transform) [23] [24] and Inverse FFT. In addition, cuFFT [17], a state-of-the-art library for computing Fourier transform on CUDA-enabled GPUs, provides great scalability to compute FFT workload.

$$F(f(x,y) * w(x,y)) = F(f(x,y))F(w(x,y)) \qquad (3)$$

$$f(x,y) * w(x,y) = F^{-1}(F(f(x,y))F(w(x,y))) \qquad (4)$$

Where F means Fourier transform, F^{-1} is inverse Fourier transform, and $*$ is the convolution operator.

4. Proposed 2D Convolution By CUDA

Our proposed method is implemented in CUDA. As shown in Algorithm 1, the steps are as follows:

(1) All of the filter weights are stored into shared memory (lines 7~12).

(2) A subset of the image data residing in global memory is cached into registers (lines 13~14). Note that we use a register cache array for each thread as well (shown as T data[C] array in line 6). The indices of the register cache array could be determined as constant quantities by the compiler at compile time. Therefore, the nvcc compiler could utilize register files rather than global memory for holding the values of the data array. Since registers are a limited resource, the register cache is managed with careful consideration (as will be discussed in more detail later).

(3) As shown in Fig. 2, according to the sliding window position and filter height, we fetch both sub-vector v and w from register cache and filter coefficients, respectively. Next, we compute the sum of their inner products (lines 24~26) and shift the partial sum to the neighbor thread via *shuffle_up* function (line 22).

(4) Repeat (3) M times for all of the sub-vectors (w_1, w_2.....w_M), then store the final partial sum (i.e. convolution result) to the register cache again (line 28)

(5) We move the sliding window step by step as shown in Fig. 1. Next, we repeat the convolution computation, namely (3) and (4), P times (line 17)

(6) Finally, the convolution results which are managed by register cache are stored back to global memory (lines 30~31)

Algorithm 1 Our CUDA kernel. P and C are defined in Equ. 10 and Equ. 11, respectively

```
1  template<typename T, int  BLOCK_SIZE, int P, int
       FILTER_WIDTH, int FILTER_HEIGHT>
2  __global__ void kernel_convolution2D  (
3    const T* src, T* dst, int width, int widthStride, int
       height, const T* weight) {
4    const int C = P + FILTER_HEIGHT - 1;
5      //register files
6    T data[C];
7    __shared__ T smem[FILTER_HEIGHT][FILTER_WIDTH];
8    T* psmem = &smem[0][0];
9      //1, Load filter weights to shared memory
10   if (threadIdx.x < FILTER_HEIGHT*FILTER_WIDTH)
11     psmem[threadIdx.x] = weight[threadIdx.x];
12   __syncthreads();
13   //2, Load data from global memory to registers
14     data[...] = src[...];
15   //3, Compute the colleration result
16   #pragma unroll
17   for (int i = 0; i < P; i++) {
18     T sum = 0;
19     #pragma unroll
20     for (int m = 0; m < FILTER_WIDTH; m++) {
21       if (m > 0)
22         sum = __my_shfl_up(sum, 1);
23       #pragma unroll
24       for (int n = 0; n < FILTER_HEIGHT; n++) {
25         sum = MAD(data[i + n], smem[n][m], sum);
26       }
27     }
28     data[i] = sum;
29   }
30     //4, Store Result to Global Memory
31     dst[...] = data[...];
32 }
```

The following sections elaborate on the steps of the algorithm.

4.1 Caching the Filter Using Shared Memory

The number of filter weights is often tens of bytes (e.g. 3×3, 5×5). Since the weights are shared by all of the threads in a CUDA block, it is reasonable to access the weights via shared memory. It is also possible to use other kinds of memory, such as constant memory, texture memory and global memory. However, considering performance and scalability, we adopt shared memory for our implementation.

In is worth mentioning that in our algorithm all of the threads in a thread block access the same address of the shared memory, which results in a broadcast read pattern to shared memory. In other words, there is no bank conflict problem (Sec. 2.2) in our

implementation when using shared memory.

Additionally, when using a float32 type 3×3 filter (M=3, N=3), for example, we only use 3×3×4=36 bytes of shared memory space, which is a relatively small fraction of the shared memory space (48K∼96K). The rest of shared memory could be used as L1 cache, which further improves the utilization of global memory, to some extent.

4.2 Caching Data Using Register Files

As shown in Algorithm 1, in order to make full use of GPU's cache line, all of the threads in a warp read data from global memory contiguously (one element per thread). The operation is repeated in order to cache multiple lines of data from global memory into register files line by line. As illustrated in Fig. 2, each thread in a single warp caches C elements (Equ. 5). Each thread will then generate an output of P elements using a sliding window. The sliding window is designed such that a portion of the data in the register cache can be reused when computing the neighboring output points. More specifically, computing the convolution of point p in a thread can reuse the data in the register cache loaded when computing the convolution of point $p-1$. Using this scheme, at any given point, a WarpSize×C register matrix is loaded in the register cache.

$$C = N + P - 1 \tag{5}$$

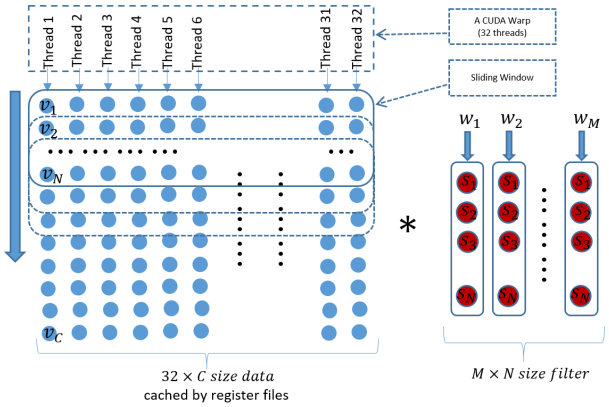


Fig. 2: The left side of the figure illustrates how to build the register cache for a warp. In a single warp, each thread reserves C registers for storing data. The register cache size in each warp is equal to 32×C (32 is the warp size). The right side of the figure shows how to cache the filter matrix. We store the filter coefficients in shared memory, then compute the convolution by moving the sliding window step by step (C-N+1=P times). At each step we compute the inner products of $[v_i, v_{i+1}, \ldots v_{i+N-1}]$ with $w_1, w_2, \ldots w_M$ as detailed in Fig 3. Next we shift the partial inner product to neighbor threads as detailed in Fig. 4

4.3 Parallel Inner Product

It is essential to access the filter weights in the same order as the data is stored in the cache register: namely unit-strided access in the vertical direction as shown in Fig. 3. As mentioned in section 4.1, the shared memory is accessed without bank conflict. The partial sum requires N multiplications and N-1 addition operations. The multiplication and addition operations are typically optimized to fused-multiply-add (FMA) instructions [17]. For the M×N filter, inner products are done M times (Fig. 2) to compute an element of convolution results (Algorithm 1 line 20∼27).

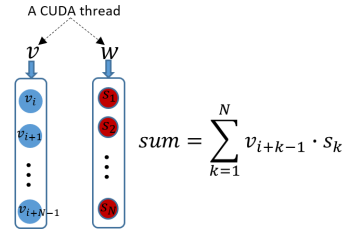


Fig. 3: Parallel Inner Product. Simultaneously all threads in a warp compute the inner product between a register vector v ($[v_i, v_{i+1}, ..., v_{i+N-1}]$) and a column of filter $w_1, w_2, ...$ or w_M. The vector v is held by each thread and w is managed by shared memory. The inner product (*sum*) is computed by a CUDA thread.

4.4 Shuffle Registers Holding the Partial Sum

As shown in Fig. 2, $M \times N$ filter is decomposed into M vectors, namely w_1, w_2, ..., w_M. Each partial sum is computed between register vector v ($[v_i, v_{i+1}, ..., v_{i+N-1}]$) and filter vector w. Next, all of inner products, namely partial sums, are shifted to the right side neighbor thread within a single warp using the CUDA primitive *shuffle_up* function (function arguments : *delta=1, width=WarpSize*) [17]. As shown in Fig. 4, all of registers are shifted only once at each step (Algorithm 1 line 22). Next, the shifted partial results are added to the accumulated results by each thread (Algorithm 1 line 25). This process is repeated M-1 times (Algorithm 1 line 20∼27). Finally a row of convolution results could be attained from a group of threads, namely whose laneIds (Sec. 2.1) range from M-1 to WarpSize-1. By moving sliding window once, each group of threads computes (WarpSize-1)-(M-1)+1=33-M convolution results.

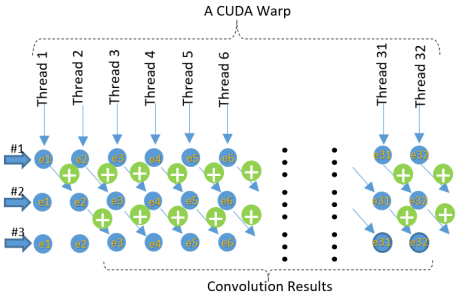


Fig. 4: Shifting partial results: e_i is a partial result computed by thread i. Threads($1 \sim 32$) represent all threads in a warp. The indicator #*i* points to the *i*th time of *shuffle_up* of partial sum registers in a warp. $\oplus$ is an add operator.

The sliding window moves step by step, as shown in Fig. 2, between v_1 and v_C elements in the register cache. At each step, the above computation scheme is repeated C-N+1 times in order to obtain the convolution results, with size (33-M)×(C-N+1)=(33-M)×P.

Finally, the convolution results (from v_1 to v_{C-N+1}) are stored back to global memory in a coalesced pattern (Algorithm 1 line 30~31).

4.5 Number of CUDA Blocks Required

It is crucial to answer the question when given an image matrix with size (W, H) (W means the image width, H is height): what is the optimized cache count C, and CUDA block size (namely B)? Given that we use a one dimensional block in our implementation, B is equal to *blockDim.x*, both *blockDim.y* and *blockDim.z* are 1. In each CUDA block, its warp count is defined as:

$$WarpCount = B/WarpSize \tag{6}$$

The required CUDA grid dimensions (namely GridDimX and GridDimY) could be defined as Equ.7 and Equ.8, respectively.

$$GridDimX = \frac{W}{WarpCount * (WarpSize - M + 1)} \tag{7}$$

$$GridDimY = H/P = H/(C - N + 1) \tag{8}$$

In this study, we only focus on discussing the scenario that a GPU's compute capability is larger than 3.5. Accordingly, the maximum count of threads in a CUDA block becomes 1024 [17]. In order to achieve the highest SMs occupancy, we require as many resident active threads as possible in a block. Finally, GridSize is defined as:

$$\begin{aligned} GridSize &= GridDimX * GridDimY \\ &= \frac{W * H * WarpSize}{B * (WarpSize - M + 1) * (C - N + 1)} \\ &= \frac{32 * W * H}{B * (33 - M) * (C - N + 1)} \end{aligned} \tag{9}$$

4.6 Block Size B & Cache Count C

In our convolution CUDA kernel, the number of active warps that can processed in parallel on SMs largely depends on the amount of register files available on the SMs. In other words, B and C tend to be determined mainly by the amount of register files.

On one hand, we want to make the GridSize (Equ. 9) as large as possible to improve the kernel's occupancy, hence, the B and C require to be as small as possible. On the other hand, as shown in Fig. 2, the larger cache count (C) means caching more original data from global memory and increasing register file reuse while computing. This improved locality improves the algorithm's throughput. To conclude, it is crucial to find the balance between B, C and register files for a given GPU architecture.

The choice of B affects the computing performance only slightly, so we emphasize our discussion on how to decide the value of C. We empirically choose B = 512, 368, and 256, respectively in experiments at lower SMs occupancy without lowering implementation's throughput. Additionally, our empirical evaluation suggests that when the P is equal to 8 (Equ. 10), we could achieve the peak of computing performance as illustrated in Fig. 5. Then C is defined as in Equ. 11.

$$P = 8 \tag{10}$$

$$C = P + N - 1 = N + 7 \tag{11}$$

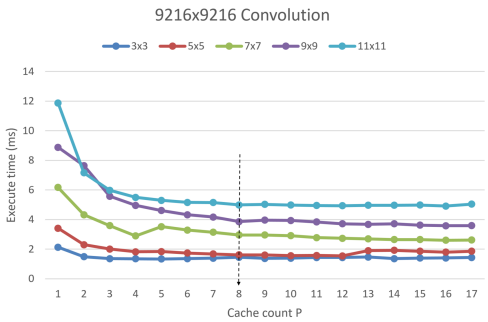


Fig. 5: The correlation between cache count P and execution time

4.7 Thread Divergence and Synchronization

In a single warp, threads execute in an SIMT fashion, hence the performance of CUDA programs is often significantly degraded by branch divergence. We carefully designed our algorithm to avoid any divergence when computing the partial sums and shifting register files, which are the heaviest and most complex parts of computing the 2D convolution. Further, unrolling the computing loop not only eliminates the negative effect of functional divergence, but also helps in improving IPL (Instruction Level Parallelism) [25] .

It is important to note the inevitability of using the synchronization primitive when caching filter weights to shared memory. In our kernel function, only one synchronization is applied. Empirical analysis revealed that the synchronization penalty is insignificant to the overall throughput.

5. Evaluation

5.1 Software & Hardware Setup

We evaluate our system with Nvidia GPUs, namely TitanX and TitanXp. For comparison, we choose software libraries, such as Nvidia's NPP [17], Nvidia's cuDNN [9], Nvidia's cuFFT [17], and the ArrayFire [14] library. We use CUDA 9 Toolkit in all the experiments. The OS-independent cudaEvent functions are used to measure the computing time, and Nvidia's nvprof profiler is used for the performance analysis [17].

5.2 Filter Size Limitations

In our algorithm, the supported filter size is limited by the total number of registers in a GPU. We examined a series of profiled runs. In our experiments P=8 is fixed, where W≤10 & H≤10, B=512; where W≤20 & H≤20, B=368; where W⩽32 & H⩽32, B=368. As Fig. 6 shows, when the filter sizes are larger than 20×20, our application's performance suddenly degrades. This is due to reaching the register limits of the GPU. In conclusion, empirically the acceptable limitation filter size is about 20×20 for both TitanX and TitanXp. It is important to note that this limit would increase with GPUs that have more aggregate register files (e.g. Volta V100 GPU).

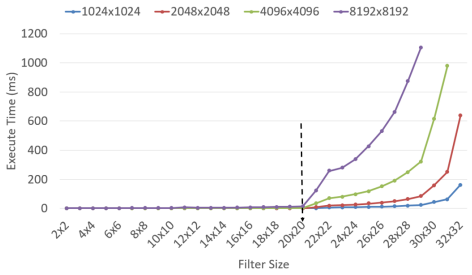


Fig. 6: Filter size limitation evaluation on TitanXp. Four kinds of image sizes are evaluated. When the filter size become larger than 20×20, the performance degrades dramatically

5.3 Results

A substantial part of this section is focused on understanding the achieved performance of our algorithm. We experimented with 2D convolutions for various input image sizes (e.g. 1024×1024, 2048×2048, 4096×4096, 8192×8192) and filter sizes (e.g. 2×2∼16×16) using Nvidia TitanXp (Fig. 7a, 8a, 9a and 10a) and Nvidia TitanX (Fig. 11a, 12a, 13a and 14a) GPUs. Nvidia's NPP library performs well with smaller images and filters, particularly for the filter sizes of 3×3 and 5×5. The execution time of FFT-based convolutions has a constant value for any filter size applied to a given size of the input image. We also evaluated the GEMM-based convolution of Nvidia's cuDNN for filter sizes with odd numbers (cuDNN limitation). We also compare to the highly optimized ArrayFire library. We emphasize the comparison with ArrayFire in (Fig. 7b, 8b, 9b, 10b, 11b, 12b, 13b, 14b). ArrayFire uses shared memory to cache data, resulting in outstanding performance. Note however that its filter size is limited from 2×2 to 15×15.

For larger sizes of input image data and filters, our algorithm demonstrably outperfroms Nvidia's libraries and ArrayFire.

Finally, it is worth mentioning that although all our discussed experiments are for square-shaped filters (i.e. M=N), in fact, our system is capable of computing 2D convolution for any filter shape (M≠N) as well.

5.4 Verification of Accuracy

To verify the accuracy of our algorithm, for every experiment, we compared the differences of the convolution results between the standard library CPU version and our GPU version. Our GPU implementation computation consistently achieved the same accuracy as the CPU, for both float32 and int32 data types.

6. Related Work

Due to the importance of 2D convolution, there has been a plethora of research on improving the computational efficiency in the past few decades.

Algorithm strength reduction based methods, such as the Cook-Toom algorithm [26], and Winograd algorithm [27], are traditionally and practically fast convolution methods by employing fewer multiplication operations. Recently, the modified Winograd method has been implemented by *cuDNN* to successfully speed up DNNs. However, it is not a general convolution method due to many requirements, such as small filter sizes and workspace buffer [6]. FFT-based [23] method is another way to reduce compute convolutions. The FFT-based convolution computational cost is constant for any given filter size. Hence the performance gain from using FFT increases as the filter size increases.

Another line of research is to accelerate convolution computation by specialized hardware such as LSIC (Very Large Scale Integration Circuit) [28] and FPGA (Field-Programmable Gate Array) [29]. However, the development cost for specialized hardware is too high for wide adoption. Another hardware approach are DSPs (Digital Signal Processors) [30]. DSPs are cheaper and easier for software programming. Powered by its efficient SIMD instructions, many embedded systems achieve impressive real-time performance by optimizing the convolution computation [31].

With the rapid growth of many-core and parallel computing technologies over the last few years, GPUs are becoming more and more prevalent in adoption as the go-to accelerator for speeding up a variety of complex computations (including convolution computation). Highly optimized GEMM kernels enable high computational efficiency on GPUs for 2D convolution. At the cost of using a large amount of temporary GPU memory to unroll the image data to large matrix, GEMM-based convolution achieves high computing efficiency.

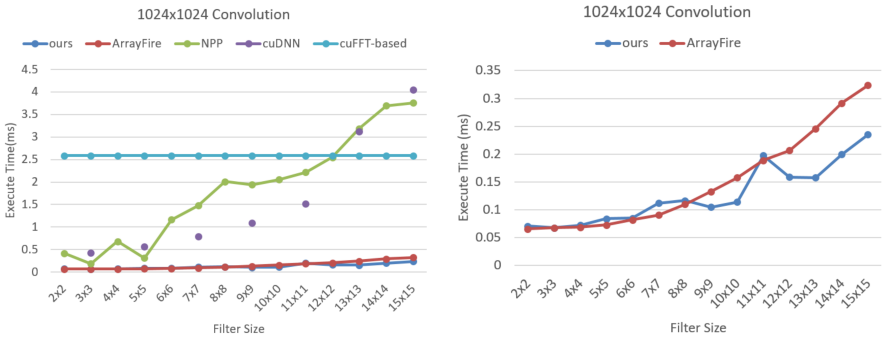
To the best of our knowledge, our proposed algorithm is the first to directly compute 2D convolution that uses register cache and shuffle instruction to accelerate 2D convolution computation without using workspace memory on CUDA-enabled GPUs.

7. Conclusion

The 2D convolution operation is widely used in many applications. Its computational efficiency is critical to the overall performance of those applications. We accelerate 2D convolution computation on CUDA-enabled GPUs using a combination of register cache technology and a novel parallel computing scheme. Unlike the prevalent methods that cache image data by shared memory, our system not only uses a virtual register cache layer to improve the data access efficiency, but also computes convolution without using workspace buffer.

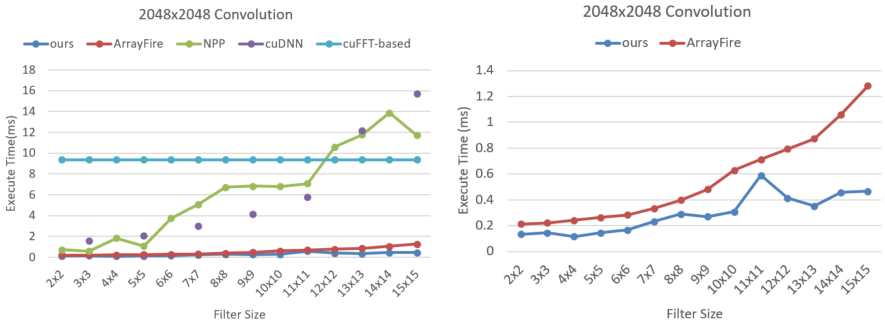
We evaluate the performance of our implementation on a single Nvidia GPU using a variety of image and filter sizes. Performance evaluation demonstrates that on average our implementation is up to 4.7× faster than NPP, and 1.8× faster than the highly-optimized ArrayFire library. To the best of our knowledge, our implementation achieves state-of-the-art performance for computing 2D convolution.

In addition to convolution computation, our proposed register cache technique and parallel inner product computation method can be applied to speed up other problems, such as stencil computation [32], which resemble convolution. It is also a promising approach to improve the computing performance for training and inference in CNNs, which rely on a large amount of 2D convolution to compute feature maps with some small filters, such as 3×3, 5×5, and 7×7 [33] [34] [35]. Those kinds of filters are well-suited to the computational pattern of our algorithm.



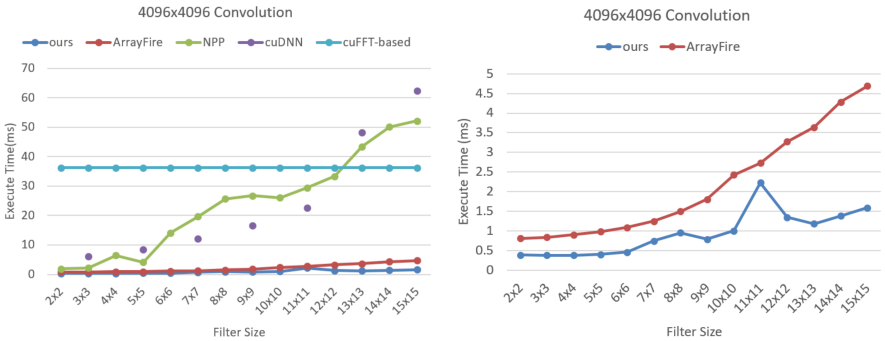
(a) Performance evaluation on ours, ArrayFire, NPP, cuDNN, and cuFFT-based method (b) Performance evaluation on ours and ArrayFire

Fig. 7: 1024×1024 TitanXp



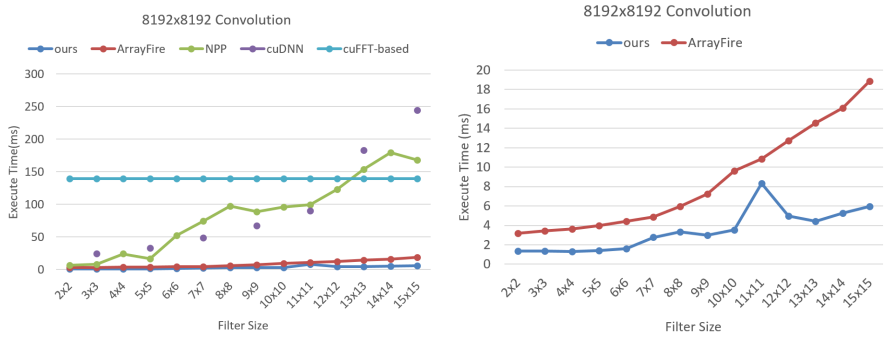
(a) Performance evaluation on ours, ArrayFire, NPP, cuDNN, and cuFFT-based method (b) Performance evaluation on ours and ArrayFire

Fig. 8: 2048×2048 TitanXp



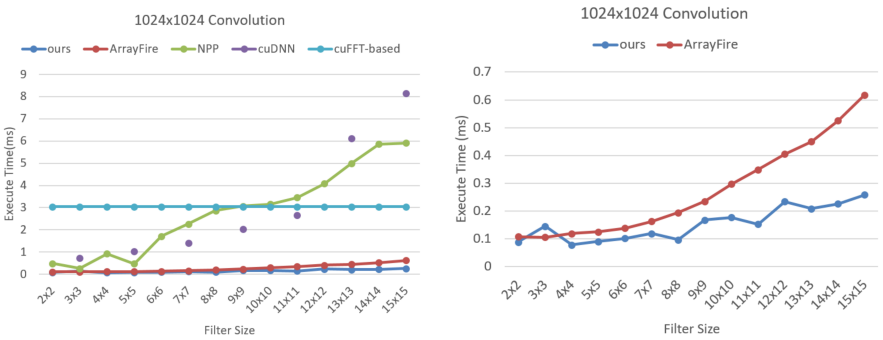
(a) Performance evaluation on ours, ArrayFire, NPP, cuDNN, and cuFFT-based method (b) Performance evaluation on ours and ArrayFire

Fig. 9: 4096×4096 TitanXp



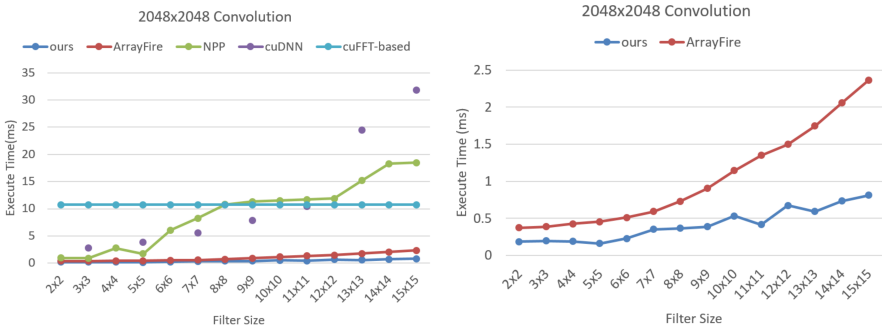
(a) Performance evaluation on ours, ArrayFire, NPP, cuDNN, and cuFFT-based method (b) Performance evaluation on ours and ArrayFire

Fig. 10: 8192×8192 TitanXp



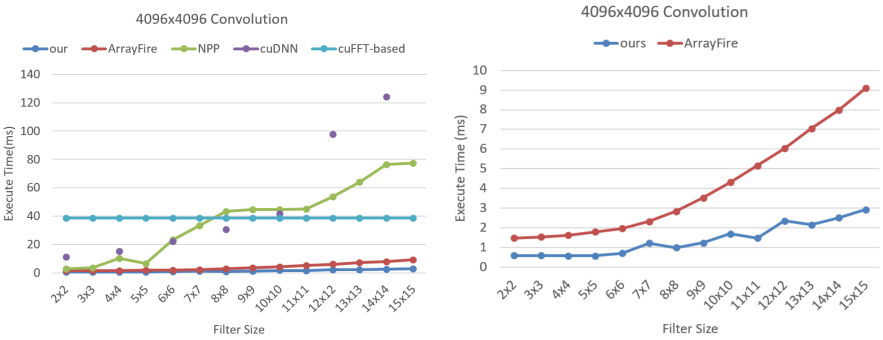
(a) Performance evaluation on ours, ArrayFire, NPP, cuDNN, and cuFFT-based method (b) Performance evaluation on ours and ArrayFire

Fig. 11: 1024×1024 TitanX



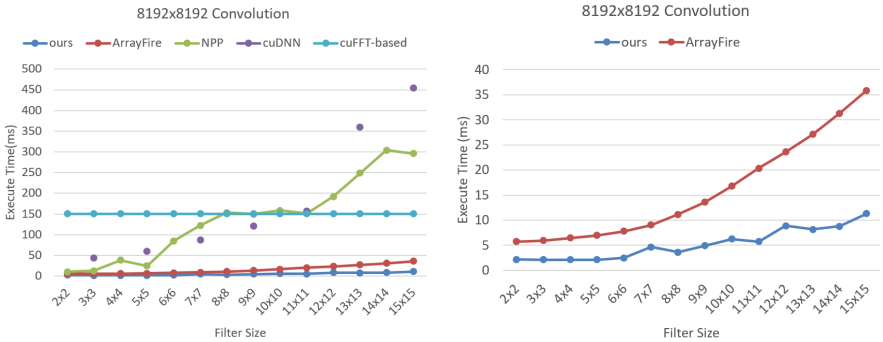
(a) Performance evaluation on ours, ArrayFire, NPP, cuDNN, and cuFFT-based method (b) Performance evaluation on ours and ArrayFire

Fig. 12: 2048×2048 TitanX



(a) Performance evaluation on ours, ArrayFire, NPP, cuDNN, and cuFFT-based method (b) Performance evaluation on ours and ArrayFire

Fig. 13: 4096×4096 TitanX



(a) Performance evaluation on ours, ArrayFire, NPP, cuDNN, and cuFFT-based method (b) Performance evaluation on ours and ArrayFire

Fig. 14: 8192×8192 TitanX

References

[1] Lowe, D. G.: Distinctive image features from scale-invariant keypoints, *International journal of computer vision*, Vol. 60, No. 2, pp. 91–110 (2004).

[2] LeCun, Y., Bengio, Y. and Hinton, G.: Deep learning, *Nature*, Vol. 521, No. 7553, pp. 436–444 (2015).

[3] Krizhevsky, A., Sutskever, I. and Hinton, G. E.: Imagenet classification with deep convolutional neural networks, *Advances in neural information processing systems*, pp. 1097–1105 (2012).

[4] Vasilache, N., Johnson, J., Mathieu, M., Chintala, S., Piantino, S. and LeCun, Y.: Fast convolutional nets with fbfft: A gpu performance evaluation, 2014, *URL http://arxiv. org/abs/1412.7580* (2014).

[5] Kolba, D. and Parks, T.: A prime factor FFT algorithm using high-speed convolution, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. 25, No. 4, pp. 281–294 (1977).

[6] Lavin, A. and Gray, S.: Fast algorithms for convolutional neural networks, *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4013–4021 (2016).

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

[8] Kågström, B., Ling, P. and Van Loan, C.: GEMM-based level 3 BLAS: High-performance model implementations and performance evaluation benchmark, *ACM Transactions on Mathematical Software (TOMS)*, Vol. 24, No. 3, pp. 268–302 (1998).

[9] Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B. and Shelhamer, E.: cudnn: Efficient primitives for deep learning, *arXiv preprint arXiv:1410.0759* (2014).

[10] Shimokawabe, T., Takaki, T., Endo, T., Yamanaka, A., Maruyama, N., Aoki, T., Nukada, A. and Matsuoka, S.: Peta-scale phase-field simulation for dendritic solidification on the TSUBAME 2.0 supercomputer, *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, IEEE, pp. 1–11 (2011).

[11] Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D. and Bengio, Y.: Theano: A CPU and GPU math compiler in Python, *Proc. 9th Python in Science Conf*, pp. 1–7 (2010).

[12] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S. and Darrell, T.: Caffe: Convolutional architecture for fast feature embedding, *Proceedings of the 22nd ACM international conference on Multimedia*, ACM, pp. 675–678 (2014).

[13] Cook, S.: *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition (2013).

[14] Yalamanchili, P., Arshad, U., Mohammed, Z., Garigipati, P., Entschev, P., Kloppenborg, B., Malcolm, J. and Melonakos, J.: ArrayFire - A high performance software library for parallel computing with an easy-to-use API (2015).

[15] NVIDIA: NVIDIA TESLA V100 GPU ARCHITECTURE, *NVIDIA white paper*.

[16] Kirk, D. et al.: NVIDIA CUDA software and GPU parallel computing architecture, *ISMM*, Vol. 7, pp. 103–104 (2007).

[17] NVIDIA: CUDA TOOLKIT DOCUMENTATION, (online), available from ⟨http://docs.nvidia.com/cuda/index.html⟩ (accessed 2018-1-25).

[18] Ben-Sasson, E., Hamilis, M., Silberstein, M. and Tromer, E.: Fast multiplication in binary fields on GPUS via register cache, *Proceedings of the 2016 International Conference on Supercomputing*, ACM, p. 35 (2016).

[19] Nickolls, J., Buck, I., Garland, M. and Skadron, K.: Scalable parallel programming with CUDA, *Queue*, Vol. 6, No. 2, pp. 40–53 (2008).

[20] Gonzalez, R. C., Woods, R. E. et al.: Digital image processing (1992).

[21] Bracewell, R. N. and Bracewell, R. N.: *The Fourier transform and its applications*, Vol. 31999, McGraw-Hill New York (1986).

[22] Harris, F. J.: On the use of windows for harmonic analysis with the discrete Fourier transform, *Proceedings of the IEEE*, Vol. 66, No. 1, pp. 51–83 (1978).

[23] Nussbaumer, H.: Fast polynomial transform algorithms for digital convolution, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. 28, No. 2, pp. 205–215 (1980).

[24] Van Loan, C.: *Computational frameworks for the fast Fourier transform*, SIAM (1992).

[25] Hwu, W.-M. W., Mahlke, S. A., Chen, W. Y., Chang, P. P., Warter, N. J., Bringmann, R. A., Ouellette, R. G., Hank, R. E., Kiyohara, T., Haab, G. E. et al.: The superblock: an effective technique for VLIW and superscalar compilation, *Instruction-Level Parallelism*, pp. 229–248 (1993).

[26] Blahut, R. E.: *Fast algorithms for signal processing*, Cambridge University Press (2010).

[27] Winograd, S.: *Arithmetic complexity of computations*, Vol. 33, Siam (1980).

[28] Elnaggar, A., Alouweiri, H. and Ito, M. R.: A new tensor product formulation for Toom's convolution algorithm, *IEEE Transactions on signal Processing*, Vol. 47, No. 4, pp. 1202–1204 (1999).

[29] Benedetti, A., Prati, A. and Scarabottolo, N.: Image convolution on FPGAs: the implementation of a multi-FPGA FIFO structure, *Euromicro Conference, 1998. Proceedings. 24th*, Vol. 1, IEEE, pp. 123–130 (1998).

[30] Gardner, W. G.: Efficient convolution without input/output delay, *Audio Engineering Society Convention 97*, Audio Engineering Society (1994).

[31] Armelloni, E., Giottoli, C. and Farina, A.: Implementation of real-time partitioned convolution on a DSP board, *Applications of Signal Processing to Audio and Acoustics, 2003 IEEE Workshop on.*, IEEE, pp. 71–74 (2003).

[32] Christen, M., Schenk, O. and Burkhart, H.: Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures, *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, IEEE, pp. 676–687 (2011).

[33] Simonyan, K. and Zisserman, A.: Very deep convolutional networks for large-scale image recognition, *arXiv preprint arXiv:1409.1556* (2014).

[34] Iandola, F., Moskewicz, M., Karayev, S., Girshick, R., Darrell, T. and Keutzer, K.: Densenet: Implementing efficient convnet descriptor pyramids, *arXiv preprint arXiv:1404.1869* (2014).

[35] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. and Rabinovich, A.: Going deeper with convolutions, *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9 (2015).