High Performance Software Systolic Array Computing of Multi-channel Convolution on a GPU

Kazuya Matsumoto(✉), Yoichi Tomioka, and Stanislav Sedukhin

The University of Aizu, Aizu-Wakamatsu, Fukushima, Japan
{kazuya-m,ytomioka,c21stans}@u-aizu.ac.jp

Abstract. The multi-input/multi-output (MIMO) channel 2D convolution is the most compute-intensive operation in Convolutional Neural Networks (CNNs). This paper presents a high-performance implementation for the MIMO convolution by extending the well-known software systolic array model (SSAM) in which the partially computed results are shifted or shuffled across multiple threads in a CUDA warp to compute the single-input/single-output (SISO) channel convolution. We propose two methods for computing a full MIMO convolution on the GPU system. In the first method, the MIMO convolution is performed by iterations of the multi-input/single-output (MISO) convolution across multiple output channels while the second method iterates the single-input/multi-output (SIMO) convolution across multiple input channels. Both methods systolically shuffle partial results multiple times during the MIMO computing. It is shown that the first method mostly demonstrates a higher performance than the second one, since the first one reuses data effectively on the L1/L2 caches as well as on the register files. We also experimentally demonstrate that a single-precision performance of the directly implemented MIMO convolution is much better than that of the SSAM/SISO-based convolution and a GEMM-based MIMO convolution of the NVIDIA cuDNN library.

Keywords: Multi-channel convolution · Software systolic array · GPU

1 Introduction

Currently, Convolutional Neural Networks (CNNs) are an effective approach in machine learning to solve many kinds of real world problems. CNNs need a large amount of computations for both the training and inference procedures. CNNs consist of several layers including convolution, activation, fully-connected, and other types of layers. In particular, the convolution layers are the most compute-intensive part in the advanced CNN models. The required multi-channel 2D convolution can be implemented in different ways either by converting it to the General Matrix-matrix Multiplication (GEMM) [2,7], Winograd algorithm [3], Fast Fourier Transform (FFT) [6], or by direct computing [8].

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
O. Gervasi et al. (Eds.): ICCSA 2022, LNCS 13375, pp. 298–309, 2022.
https://doi.org/10.1007/978-3-031-10522-7_21

In [1], the authors have demonstrated that an SSAM single-input/single-output channel (SISO)-based direct computing can achieve a much higher performance than deep learning libraries such as cuDNN, ArrayFire and Halide on Graphics Processing Units (GPUs). However, to accelerate deep learning further we need an extension of the SSAM-based approach for the basic multiple-input/multiple-output (MIMO) channels convolution because the convolutional layers of CNNs typically deal with a tensor data with MIMO channels.

The SSAM-based input-stationary approach proposed in [1] realizes a systolic data movement for SISO convolution using CUDA shuffle primitives. The approach reuses the input feature map from caches and intermediate results on the register files more effectively and reduces the time and energy expensive DRAM read/write accesses, which are helpful to extract a higher performance and, at the same time, to reduce energy consumption. However, for the MIMO convolution, we need much more data for computing each output pixel. Therefore, it is essential to effectively use the L1/L2 caches as well as register files to extract the full performance of the SSAM-based approach.

This paper presents a high-performance implementation for the multi-channel convolution computing by extending the SSAM-based convolution kernel. We propose two methods for the multi-channel convolution computing on the GeForce RTX 3090 of the NVIDIA Ampere GPU architecture. The first method conducts iterations of a multi-input/single-output convolution. In the second method, iterations of a single-input/multi-output convolution are conducted by keeping the partial sums during the MIMO computing. This paper presents performance evaluation results by using a profiling tool and describes performance comparison results with the original SSAM/SISO-based convolution kernel and the NVIDIA cuDNN library.

2 Convolution Implementations with Software Systolic Array for Multi-input and Multi-output Channels

The terminology used in this paper is described in Table 1. The 2D convolution is expressed as

$$y[i,j] = (x * w)[i,j] = \sum_{s=a_l}^{a_u} \sum_{t=b_l}^{b_u} x[i-s, j-t] \cdot w[s,t],$$

where $*$ is the convolution operation, $\cdot$ is the multiplication operation, w is a filter of the size $M \times N$ ($M = a_u - a_l + 1$ and $N = b_u - b_l + 1$), x is a 2-dimensional input matrix (or image) of the size $H \times W$, and y is a 2-dimensional output matrix of the size $(H + 2pad - M + 1) \times (W + 2pad - N + 1)$ (pad is a padding value). The multi-channel convolution with C-input channels and K-output channels is expressed as

$$y_m[k,i,j] = (x_m * w_m)[k,i,j] = \sum_{c=1}^{C} \sum_{s=a_l}^{a_u} \sum_{t=b_l}^{b_u} x_m[c, i-s, j-t] \cdot w_m[k,c,s,t]. \quad (1)$$

w_m is a filter of the size $K \times C \times M \times N$. x_m is an input tensor of the size $C \times H \times W$. y_m is an output tensor of the size $K \times (H + 2pad - M + 1) \times (W + 2pad - N + 1)$.

Table 1. Terminology

Term	Description
H	Input image height
W	Input image width
M	Filter height
N	Filter width
C	The number of input channels
K	The number of output channels
x_m	Input tensor
y_m	Output tensor
w_m	Filter tensor
pad	Padding value
BS	CUDA thread block size
P	Number of rows processed by each thread

In this study, the given tensor data are stored in KCHW, CHW, and KHW orders for filter, input and output tensors, respectively. For example, in the KCHW order, the W data elements of the same row are continuously stored, the $H \times W$ matrix data for each of C input channels are continuously allocated, and the $C \times H \times W$ tensor data for each of K output channels are continuously stored.

The methods proposed in this paper basically follow the 2D convolution algorithm by the Software Systolic Array Model (SSAM) described in [1]. The software systolic array simulates a mechanism of hardware systolic arrays, and the SSAM is suitable for memory-bound computations with regular memory access patterns. The features of the SSAM-based 2D convolution algorithm are as follows:

– All data of filters are stored into shared memory in the beginning of the CUDA kernel and are reused during the convolution computing.
– Input image data are cached into registers. A sliding window scheme is used for the careful management of the limited number of registers.

– Each thread in a CUDA warp caches $D(= N + P - 1)$ elements of the input image also in the beginning, where P is the number of sliding window steps. Hence, each thread computes the P rows of elements in the image.
– The input data are multiplied with the filter data and the product is accumulated. The computation is conducted by the fused multiply-add (`MAD`) operation, and the partial sums are transferred to the neighbor threads by the *shuffle* primitive.

CUDA kernels are called with a block dimension (blockDim) and a grid dimension (gridDim) of the 3D arrays. The blockDim denotes the number of threads in a block. The gridDim denotes the number of the blocks in a grid. In this study, the dimensions of the CUDA kernels are given as

$$\text{blockDim} = (BS, 1, 1) \text{ and}$$
$$\text{gridDim} = (\lceil \frac{W + 2pad}{WarpCount \cdot (WarpSize - N + 1)} \rceil, \lceil \frac{H + 2pad}{P} \rceil, 1),$$

where BS is the CUDA thread block size, $WarpSize$ is the CUDA warp size, and $WarpCount = BS/WarpSize$. Threads in a block of the CUDA kernels are in charge of computing a $K \times P \times BS$ sub-tensor of the $K \times (H + 2pad - M + 1) \times (W + 2pad - N + 1)$ output tensor.

In the present study, we have implemented two convolution methods for multi-input and multi-output channels. Listing 1.1 presents iterations across output channels in the outermost loop, while Listing 1.2 shows iterations across input channels in the outermost loop. In the following, Listings 1.1 and 1.2 are called OUT_IN method and IN_OUT method, respectively. In the implementation of both the methods, the filters are accessed in a decremental order and the input images are accessed in an incremental order, whereas each of these is accessed in the opposite order of the Eq. (1).

In the OUT_IN method, the data for the accumulation is initialized to the zero at the beginning of each outermost-loop (Lines 8–9 in Listing 1.1) and the computed results are stored to the destination array at the end of each outermost-loop (Lines 32–33). On the other hand, in the IN_OUT method, the array for the accumulation is initialized before the outermost loop (Lines 7–9 in Listing 1.2) and the results are written out to the destination array after the outermost-loop (Lines 34–37).

For both the methods, the number of required floating-point operations per CUDA thread is $FLOP_{thread} = 2 \cdot K \cdot C \cdot M \cdot N \cdot P$. The output amount to the GPU off-chip memory for both the methods is $K \cdot P \cdot sizeof(dataType)$. The input amount of data from the GPU's off-chip memory for the OUT_IN method is $K \cdot C(P + M - 1 + \lceil M \cdot N/BS \rceil) \cdot sizeof(dataType)$. The IN_OUT method reuses input data in each outermost iteration after reading the data (Lines 11–12 in Listing 1.2), and the input amount from the GPU's off-chip memory for the IN_OUT method is $C(P + M - 1 + \lceil M \cdot N/BS \rceil) \cdot sizeof(dataType)$. This means

that the IN_OUT method requires K times less memory accesses for the input tensor than the OUT_IN method. However, the IN_OUT method has to keep K times larger temporary data for the summations of `sum_final` in a 2D array. Note that the temporary data might be forced out to the off-chip memory if the available number of registers is not enough for keeping the data.

Listing 1.1. CUDA kernel of multi-channel SSAM-based convolution (the loop across the output channels is the outermost).

```
1  template<typename T, int BS, int P, int M, int N>
2  __global__ void convolution_OUT_IN(const T *xm, T *ym, const T *wm, const int
       H, const int W, const int C, const int K) {
3    const int D = P + N - 1;
4    T data[D], sum[P];
5    __shared__ T smem[N][M];
6    T *psmem = &smem[0][0];
7    for (int k = 0; k < K; k++) {
8      for (int p = 0; p < P; p++)
9        sum[p] = 0;
10     for (int c = 0; c < C; c++) {
11       __syncthreads();
12       for (int l = threadIdx.x; l < M*N; l += BS)
13         psmem[l] = wm[M*N*(C*k+c)+l];
14       __syncthreads();
15       for (int d = 0; d < D; d++)
16         data[d] = xm[SRC_IDX+H*W*c+W*d];
17       #pragma unroll
18       for (int p = 0; p < P; p++) {
19         T sum_partial = 0;
20         #pragma unroll
21         for (int j = 0; j < N; j++) {
22           if (j >= 1)
23             sum_partial = __shfl_up_sync(0xffffffff, sum_partial, 1);
24           #pragma unroll
25           for (int i = 0; i < M; i++) {
26             sum_partial = MAD(data[p+i], smem[M-1-i][N-1-j], sum_partial);
27           }
28         }
29         sum[p] += sum_partial;
30       }
31     }
32     for (int p = 0; p < P; p++)
33       ym[DST_IDX+(W+2*pad-N+1)*((H+2*pad-M+1)*k+p)] = sum[p];
34   }
35 }
```

Listing 1.2. CUDA kernel of multi-channel SSAM-based convolution (the loop across the input channels is the outermost).

```
1  template<typename T, int BS, int P, int M, int N>
2  __global__ void convolution_IN_OUT(const T *xm, T *ym, const T *wm, const int
        H, const int W, const int C, const int K) {
3    const int D = P + N - 1;
4    T data[D], sum[K][P];
5    __shared__ T smem[N][M];
6    T *psmem = &smem[0][0];
7    for (int k = 0; k < K; k++)
8      for (int p = 0; p < P; p++)
9        sum[k][p] = 0;
10   for (int c = 0; c < C; c++) {
11     for (int d = 0; d < D; d++)
12       data[d] = xm[SRC_IDX+H*W*c+W*d];
13     for (int k = 0; k < K; k++) {
14       __syncthreads();
15       for (int l = threadIdx.x; l < M*N; l += BS)
16         psmem[l] = wm[M*N*(C*k+c)+l];
17       __syncthreads();
18       #pragma unroll
19       for (int p = 0; p < P; p++) {
20         T sum_partial = 0;
21         #pragma unroll
22         for (int j = 0; j < N; j++) {
23           if (j >= 1)
24             sum_partial = __shfl_up_sync(0xffffffff, sum_partial, 1);
25           #pragma unroll
26           for (int i = 0; i < M; i++) {
27             sum_partial = MAD(data[p+i], smem[M-1-i][N-1-j], sum_partial);
28           }
29         }
30         sum[k][p] += sum_partial;
31       }
32     }
33   }
34   for (int k = 0; k < K; k++)
35     for (int p = 0; p < P; p++)
36       ym[DST_IDX+(W+2*pad-N+1)*((H+2*pad-M+1)*k+p)] = sum[k][p];
37 }
```

3 Performance Evaluation

The specification of the NVIDIA RTX 3090 GPU [5] is shown in Table 2. The used CUDA toolkit version is 11.4 and the GPU driver version is 470.57.02. The operating system of the environment is Ubuntu 20.04.2 LTS with Linux 5.13.0-35-generic kernel. The program codes are compiled by `nvcc` command with `-gencode arch=compute_86,code=sm_86 -fmad true` options and the param-

eter setting[1] of $P = 4$ and $BS = 128$. The performance measurements are conducted with the padding value $pad = 1$. Matrix and tensor data in the program are allocated as single-precision floating-point arrays and single-precision operations are used for computing the multi-channel convolution. CUDA cores of the GPU are utilized and its Tensor Cores are not used in the evaluation.

Table 2. Specification of the NVIDIA RTX 3090 GPU. The L1 cache and shared memory use the same 128 KB hardware resources per SM, and the preferred cache configuration can be selected.

Number of CUDA/shader cores	10,496
Number of Streaming Multiprocessors (SMs)	82
Core base clock speed	1,395 MHz
Theoretical peak single-precision performance	29,284 Gflop/s
Memory type	GDDR6X
Memory size	24 GB
Memory base clock speed	1,219 MHz
Memory bus width	384-bit
Memory system bandwidth	936 GB/s
L2 cache size	6 MB
L1 cache size per SM	Up to 96 KB
Max 32-bit registers per SM	65,536
Max 32-bit registers per thread	255
Warp size	32

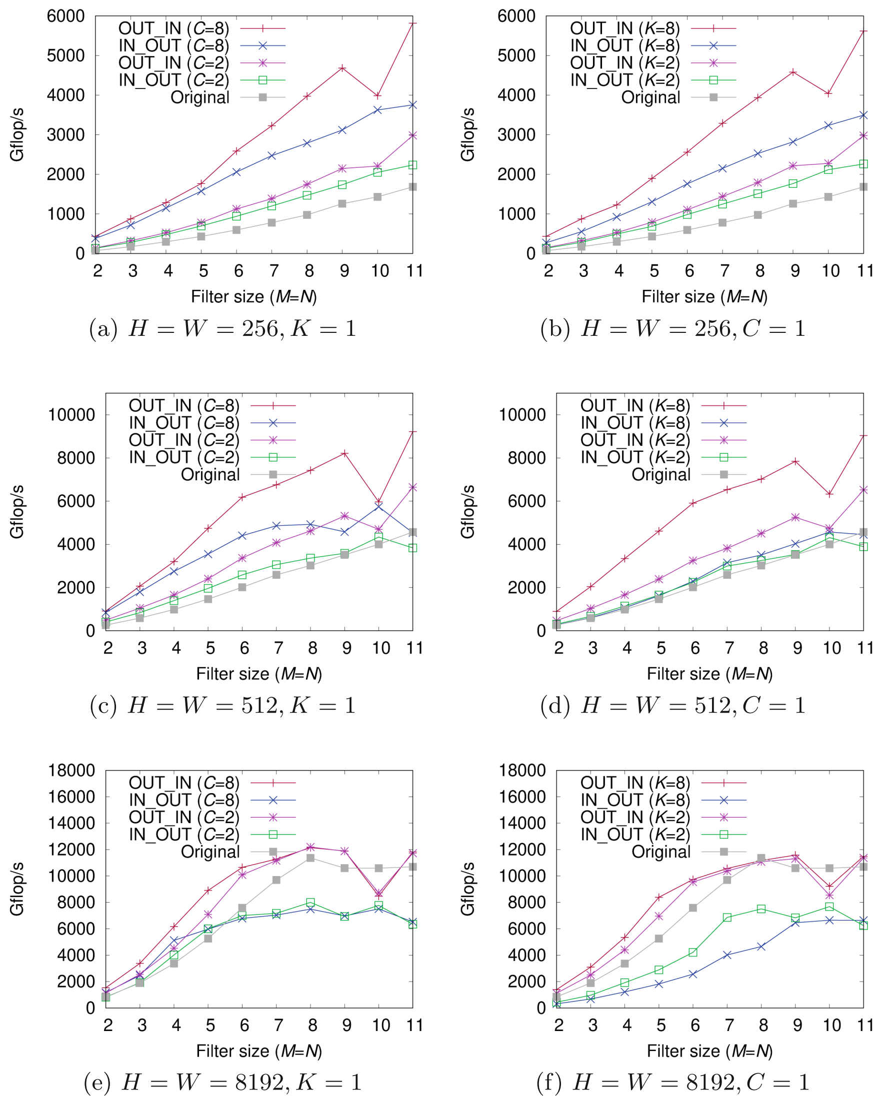
Figure 1 shows the performance of the convolution kernel implementation for the $H \times W$ image with C-input and K-output channels. The Gflop/s performance is calculated by the following formula:

$$\text{Performance [flop/s]} = \frac{2 \cdot H \cdot W \cdot M \cdot N \cdot C \cdot K \text{ [flop]}}{\text{time [second]}}.$$

In the evaluation, each of the convolution CUDA kernels is called 20 times and its average time is used as the computation time. The "Original" in Fig. 1 is the performance of the 2D convolution kernel[2] by Chen et al. [1]. Note that the performance of the "Original" method is almost the same as the performance of the implemented SISO case ($C = K = 1$ case).

[1] The parameter setting in the convolution kernels affects its performance. Evaluations of the effects and optimizations of the parameter setting are considered in our future work.

[2] The "Original" kernel implementation was slightly modified for evaluating the performance on the same condition because their kernel supposes that the input matrix size is equal to the output matrix size.



(a) $H = W = 256, K = 1$ (b) $H = W = 256, C = 1$

(c) $H = W = 512, K = 1$ (d) $H = W = 512, C = 1$

(e) $H = W = 8192, K = 1$ (f) $H = W = 8192, C = 1$

Fig. 1. Performance of computing convolution with C-input and K-output channels for the $H \times W$ image.

As shown in Fig. 1, the performance of the IN_OUT method is higher than the OUT_IN and Original methods in the most cases. For example, in case of the $H = W = 256, K = 1$ (Fig. 1a), the performance of the OUT_IN method with $C = 8$ for $M \times N$ filter is 435 Gflop/s and it is 1.14 and 5.95 times higher than

that of the IN_OUT method with $C = 8$ (383 Gflop/s) and the Original method (73 Gflop/s), respectively. The performance advantage of the OUT_IN method over Original method becomes weaker when the image sizes and filter size are larger; the Original method shows higher performance than OUT_IN method in the case when $H = W = 8192, M = N = 10$ (Figs. 1e, 1f).

Table 3 shows profiling results of multi-channel convolution implementations by using `ncu` command from the NVIDIA Nsight Compute profiler[3] for 512×512 images and 3×3 filters. The OUT_IN method shows higher cache hit rates and higher instruction issue slot utilization than the other methods. In case $K = 8$, the profiling results of the IN_OUT method indicate that its memory utilization is not efficient: the DRAM (off-chip memory) read amount is much larger and the cache hit rates are lower than the other cases. The low memory utilization is probably caused by register spills [4], which lead to repeating read-write accesses between the registers and off-chip memory. The IN_OUT method requires to keep `sum` array data during computing all K output channels; hence, a lack of available registers are highly possible in the case of large number of output channels.

Table 3. Profiling results of multi-channel convolution computing by Nsight Compute for the case of $H = W = 512, C = 1, M = N = 3$.

	OUT_IN		IN_OUT		Original
	$K = 2$	$K = 8$	$K = 2$	$K = 8$	
Performance [Gflop/s] (% to peak performance)	1,030 (3.52%)	2,042 (6.97%)	665 (2.27%)	601 (2.05%)	587 (2.00%)
DRAM read amount [Mbytes]	1.05	1.06	1.06	12.68	1.05
L1 cache hit rate	52.50%	69.73%	59.13%	39.07%	27.66%
L2 cache hit rate	75.42%	90.91%	89.20%	72.25%	65.05%
Warp cycles per issued instruction	11.10	10.15	21.92	61.28	15.56
Issued instructions (Issue slot utilization)	952,283 (56.56%)	3,494,891 (63.23%)	920,085 (27.56%)	2,560,904 (10.91%)	464,372 (39.01%)
Issued instuctions per active cycle	2.26	2.53	1.10	0.44	1.56

Figure 2 shows a performance for the different numbers of input and output channels, and different sizes of images, and 3×3 filters. In Fig. 2, the performance of a GEMM-based computing of multi-channel convolution computing in the NVIDIA cuDNN v8.3.1 is also depicted. The cuDNN is a GPU-accelerated library for deep neural networks. The cuDNN supports several algorithms for the convolution foward (`cudnnConvolutionForward`) function. Figure 2 shows the cuDNN performance of the `IMPLICIT_PRECOMP_GEMM` algorithm which demonstrates higher performance than other supported algorithms in the most cases for single-precision data. A performance of the cuDNN library modestely increases when the image and the number of channels are increased.

[3] https://developer.nvidia.com/nsight-compute.

The OUT_IN implementation shows much higher performance than the cuDNN library. As shown in Fig. 2, increasing the number of channels does not always improve the performance of the OUT_IN implementation. In the case of relatively small image sizes like $H = W = 256$ or $H = W = 512$ (Fig. 2a, b), the performance tends to improve with respect to increasing the number of channels. However, the performance deteriorates when $H = W = 1024, K = 8$ or $K = 32$ (Fig. 2d). Figure 3 shows L1 and L2 cache hit rates of the OUT_IN implementation (the rates are measured also by the Nsight Compute profiler). The L1 hit rates for 128×128 images do not drop down even when the number of channels increases. On the flip side, the L1/L2 hit rates for 512×512 or 1024×1024 images are first decreased and leveled off along with increasing the number of channels. These results of cache hit rates are compiled with the performance tendencies as shown in Fig. 2.

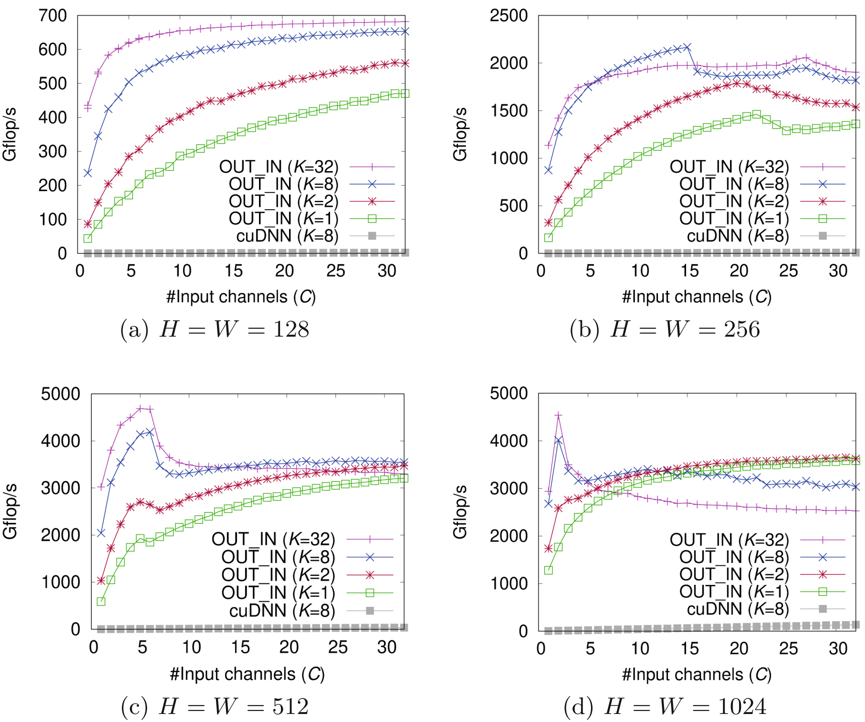


Fig. 2. Performance of convolution computing by the OUT_IN method and cuDNN with C-input and K-output channels for the $H \times W$ image and 3×3 filters.

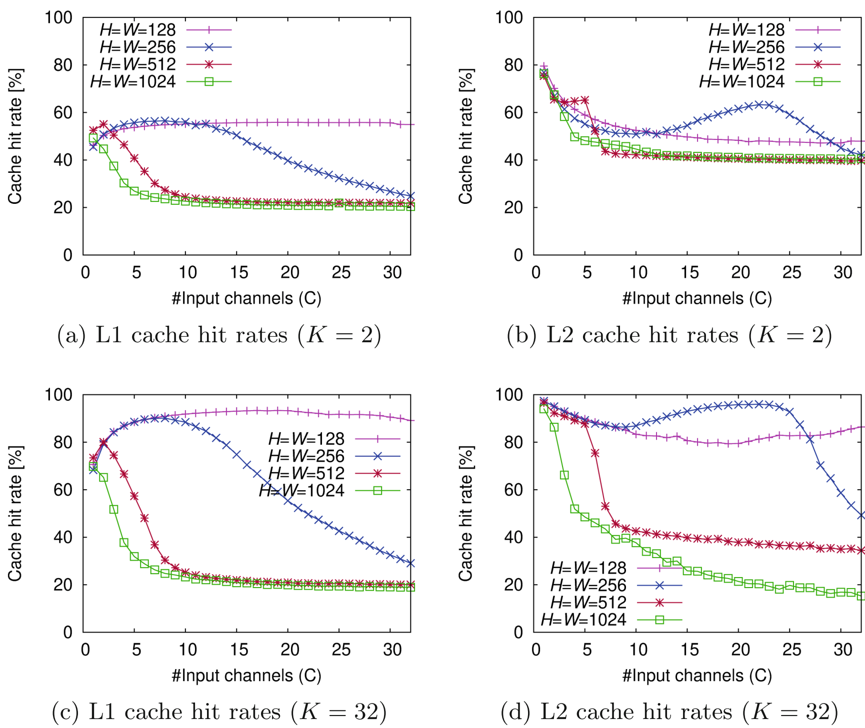


Fig. 3. L1 and L2 cache hit rates of the OUT_IN convolution computing with C-input and K-output channels for the $H \times W$ image and 3×3 filters.

4 Conclusion

We have developed a high-performance kernel for multi-channel convolution computing by extending an existing convolution kernel with a software systolic array model [1]. The implemented multi-channel kernel demonstrates a higher performance than the original single-channel kernel and the GEMM-based computing of cuDNN library. Profiling results of the implemented kernel show that efficient usage of registers and caches is more important for the multi-channel computing than the single-channel computing.

Our future work includes an application of the implemented multi-channel convolution kernel to different CNN models such as VGG, GoogleNet, and ResNet. Another future work is to conduct more detailed performance comparison with other deep learning libraries using Tensor and CUDA Cores of the current and future GPUs.

References

1. Chen, P., Wahib, M., Takizawa, S., Takano, R., Matsuoka, S.: A versatile software systolic execution model for GPU memory-bound kernels. In: SC 2019: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–81. ACM (2019). https://doi.org/10.1145/3295500.3356162
2. Jorda, M., Valero-Lara, P., Pena, A.J.: Performance evaluation of cuDNN convolution algorithms on NVIDIA volta GPUs. IEEE Access **7**(1), 70461–70473 (2019). https://doi.org/10.1109/ACCESS.2019.2918851
3. Lavin, A., Gray, S.: Fast algorithms for convolutional neural networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 4013–4021 (2016)
4. Micikevicius, P.: Local Memory and Register Spilling. NVIDIA Corporation (2011)
5. NVIDIA Corporation: 3090 & 3090 Ti Graphics Cards — NVIDIA GeForce. https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3090-3090ti/
6. Podlozhnyuk, V.: FFT-based 2C convolution. NVIDIA White Paper (2007)
7. Vasudevan, A., Anderson, A., Gregg, D.: Parallel multi channel convolution using general matrix multiplication. In: Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors, pp. 19–24 (2017). https://doi.org/10.1109/ASAP.2017.7995254
8. Zhao, Y., Wang, D., Wang, L.: Convolution accelerator designs using fast algorithms. Algorithms **12**(5), 112 (2019). https://doi.org/10.3390/a12050112