Accelerating Finite Impulse Response Filtering Using Tensor Cores

*Takumi Kondo, †Yoshihiro Maeda, and *Norishige Fukushima
*Nagoya Institute of Technology, Japan
†Tokyo University of Science, Japan

***Abstract*—This paper studies how to accelerate a single channel 2D image convolution using NVIDIA's Tensor Core. Tensor Core is a dedicated arithmetic unit for speeding up matrix products and was proposed to speed up the convolution in machine learning. From the Volta architecture of NVIDIA's GPU, Tensor Core is utilized for various applications. The Tensor Core is not limited to machine learning but various applications. This paper focus acceleration of image convolution of a single-channel filter of finite impulse response (FIR) filtering. Usually, Tensor Core is used for multi-channel convolution; thus, direct usage of Tensor Core cannot maximize the performance of the usual FIR filtering. Therefore, we propose three programming patterns for parallel processing of FIR filtering: kernel-loop unrolling, kernel-image-loop unrolling, and diagonal kernel-image-loop unrolling. Our experiments compare the proposed loop unrolling methods for linear time-invariant (LTI) and variant filters (LTV): Gaussian filtering and bilateral filtering. The results show that Tensor Core can accelerate the convolution for LTI filters but cannot accelerate linear LTV filters. Also, we compare the two GPU architectures of Turing and Ampere, and we can accelerate the LTI filter using TCs in both architectures.**

I. Introduction

Image filtering is an essential tool for image processing, and neighborhood filtering, i.e., finite impulse response (FIR) filtering, is fundamental. These filters are used for various applications: edge detection [1], scale-space analysis [2], denoising [3], [4], [5], detail enhancement [6], high-dynamic range imaging [7], dehazing [8], [9], style transfer [10], depth filtering [11] image quality assessment [12], and image compression [13], [14], [15].

The computational order of the naïve FIR convolution is $O(r^2)$, where r is the filtering radius; thus, its computational time becomes long when r is large. Therefore, acceleration algorithms for specific filters are proposed. For example, Gaussian filtering is accelerated by separable convolution and fast Fourier transform (FFT), traditionally. The recursive approximations [16], [17] and sliding discrete cosine transform methods [18], [19] further accelerate the Gaussian filtering. In the bilateral filtering [4] cases, there are many accelerating approaches [20], [21], [22], [23], [24], [25], [26]. Look-up-table approaches can accelerate any filters [27], [28]. However, these algorithms are insufficient scalability for massively parallel computing such as graphic processing units (GPUs). Under the insufficient parallelability algorithm, naïve implementation with GPUs is faster than the optimized algorithms.

This work was supported by JSPS KAKENHI (21H03465, 21K17768).

GPUs have several to several hundred times the number of cores of CPUs and can achieve high computational speed by parallelizing computations. Moreover, in May 2017, NVIDIA releases Volta architecture equipped with units of Tensor Cores (TC) [29], hardware dedicated to matrix-multiply-and-accumulate operation on a 4×4 matrix in one GPU clock cycle. TCs perform matrix multiplication with input data in half floating-point precision and the accumulation in 32-bit single-precision in mixed-precision mode.

The TCs is developed for deep learning, consisting of the multi-channeled convolution, represented by matrix-multiply operations. However, various image processing techniques used the matrix-multiply operations, such as FFT [30]. Also, TCs is used for various tasks [31]. Moreover, TCs is effective for reduction and scan parallel computing pattern [32], which are used in various image processing. In addition to NVIDIA GPUs, accelerating matrix-multiply is also possible with ACIS, such as Google's Tensor Processing Unit (TCP) [33] and Intel Movidius Myriad 2 [34]. CPU extensions for the matrix-multiply unit will include the Intel Advanced Matrix Extension (AMX), which will be available from the Sapphire Rapids architecture, and the ARM Scalable Matrix Extension (SME), which will be available from Armv9-A. Moreover, the computing unit is extended for Halide [35], [36], [37], which is a domain specific language for image processing, to utilize TCs [38] for easy to use TCs. Therefore, the usage of the matrix-multiply unit will become more common in the future.

In this paper, we study how to accelerate a single-channel 2D image convolution using TCs in GPUs. TCs prefers multi-channel convolutions; thus, the single-channel convolution cannot maximize the performance of TCs by usual usage. This paper proposes three loop unrolling patterns for parallel programming using TC: kernel-loop unrolling, kernel-image-loop unrolling, and diagonal kernel-image-loop unrolling to maximize the performance. Also, we compare the speed of two types of filters to verify the variation of filters: linear time-invariant (LTI) filter and linear time-variant (LTI) filter.

II. FIR Image Convolution

We review two types of FIR filters of LTI and LTV filters.

A. Linear Time-invariant Filter

LTI filters are linear and time-invariant filters, i.e., the kernel weight is constant for each pixel. LTI filters include moving-averaging filtering, Gaussian filtering, Sobel filtering, Prewitt

filtering, Laplacian filtering, Gaussian of Laplacian filtering, Gabor filtering. The definition of the LTI is defined as follows:

$$\bar{\boldsymbol{I}}(\boldsymbol{p}) = \frac{\sum_{\boldsymbol{q}\in N(\boldsymbol{p})} W(\boldsymbol{p},\boldsymbol{q})\boldsymbol{I}(\boldsymbol{q})}{\sum_{\boldsymbol{q}\in N(\boldsymbol{p})} W(\boldsymbol{p},\boldsymbol{q})} \tag{1}$$

where $\boldsymbol{I}(\boldsymbol{q})$ is an image intensity at a pixel $\boldsymbol{q}$, and $\bar{\boldsymbol{I}}(\boldsymbol{p})$ is an output intensity at $\boldsymbol{p}$. The weight function $W(\boldsymbol{p},\boldsymbol{q})$ indicates a kernel weight between a focusing pixel $\boldsymbol{p}$ and a neighborhood pixel $\boldsymbol{q}$. The LTI weight function is fixed for each pixel; thus, when both the pixel of the focusing and the neighborhood pixel are shifted by one position, they are constant; $W(\boldsymbol{p},\boldsymbol{q}) = W(\boldsymbol{p}+1,\boldsymbol{q}+1)$. Therefore, the LTI filter is a time-invariant filter. The LTI convolution weight does not depend on the luminance value of the focusing pixel, but it only depends on the relative position of reference pixels. For example, we show the weight of the Gaussian filtering:

$$W(\boldsymbol{p},\boldsymbol{q}) = \exp\left(\frac{-\|\boldsymbol{p}-\boldsymbol{q}\|_2^2}{2\sigma_s^2}\right), \tag{2}$$

where, σ is a standard deviation parameter of the Gaussian distribution, and $\|\cdot\|$ indicates the L2 norm function.

B. Linear Time-variant Filter

An LTV filter has linearity and time degeneration, including a parameter adaptive LTI filter, a wavelet transform, and a bilateral filter. Here, we define bilateral filtering:

$$\begin{aligned}
\bar{\boldsymbol{I}}(\boldsymbol{p}) &= \frac{\sum_{\boldsymbol{q}\in N(\boldsymbol{p})} \omega_s(\boldsymbol{p},\boldsymbol{q})\omega_r(\boldsymbol{I}(\boldsymbol{p}),\boldsymbol{I}(\boldsymbol{q}))\boldsymbol{I}(\boldsymbol{q})}{\sum_{\boldsymbol{q}\in N(\boldsymbol{p})} \omega_s(\boldsymbol{p},\boldsymbol{q})\omega_r(\boldsymbol{I}(\boldsymbol{p}),\boldsymbol{I}(\boldsymbol{q}))} \\
\omega_s(\boldsymbol{p},\boldsymbol{q}) &= \exp\left(\frac{-\|\boldsymbol{p}-\boldsymbol{q}\|_2^2}{2\sigma_s^2}\right) \\
\omega_r(\boldsymbol{I}(\boldsymbol{p}),\boldsymbol{I}(\boldsymbol{q})) &= \exp\left(\frac{-\|\boldsymbol{I}(\boldsymbol{p})-\boldsymbol{I}(\boldsymbol{p})\|_2^2}{2\sigma_r^2}\right)
\end{aligned} \tag{3}$$

where $\omega_s(\boldsymbol{p},\boldsymbol{q})$ is the same weight function of Gaussian filtering; thus, the weight has time-invariant property. By contrast, $\omega_r(\boldsymbol{I}(\boldsymbol{p}),\boldsymbol{I}(\boldsymbol{q}))$ refers to the luminance value of the pixel of interest, its weight changes depending on the pixel of interest. Therefore, the filter is a time-varying function.

III. Convolution on GPU

A. Graphic Processing Unit

NVIDIA's GPUs consist of Streaming Multiprocessor (SM) units, and each SM contains four sub-cores. Each sub-core contains INT32, FP32, and FP64 processors and TC. The consumer GPU's micro-architectures are Pascal (GTX 10, 2016), Turing (RTX 20, 2018), and Ampere (RTX 30, 2020) in NVIDIA GeForce. The professional usage GPUs are Pascal (Tesla P100, 2016), Volta (Tesla V100/Titan V, 2017), and Ampere (A100, 2020). In the current newest GPU of A100 GPU, they consist of the following [39].

- SMs: 128 (16 SM/Graphics Processing Cluster (GPC))
- INT32 CUDA Core: 64 per SM
- FP32 CUDA Core: 64 per SM
- FP64 CUDA Core: 32 per SM
- TC: 4 per SM

TABLE I
Matrix layout that can be computed with WMMA API

instruction	Input Type	Accumulation Type	Possible size
DMMA	double (FP64)	double	$8\times8\times4$
HMMA (TF32)	TF32	float (FP32)	$16\times16\times8$
HMMA (bfloat16)	bfloat16		
HMMA (FP16)	half (FP16)	half	$16\times16\times16$
		float	$32\times8\times16$
IMMA (8 bit)	char	integer (INT32)	$8\times32\times16$
	unsigned char		
IMMA (4 bit)	u4 (4 bit unsigned)		$8\times8\times32$
	s4 (4 bit signed)		
BMMA (1 bit)	b1(1 bit)		$8\times8\times128$

CUDA Cores are usual arithmetic computing unit, and TC is an arithmetic computing unit that performs matrix-multiply operations at high speed, and the Volta micro-architecture firstly mounts it. Volta and Turing architectures have 8 TCs per SM. Each TC can perform $4\times4\times4$ matrix-multiply operations in one clock cycle, equivalent to performing 64 FP16/FP32 mixed-precision fusion multiply-accumulate (FMA) operations in one clock cycle. The Ampere architecture has only 4 TCs per SM, while it has the 3rd generation TCs, which can perform $8\times4\times4$ (GA102) or $8\times8\times4$ (GA100) matrix multiply per a clock. Therefore, the Ampere architecture has twice the processing power per one TC compared to the Volta and Turing.

The lowest level interface to program the usage of TCs is CUDA warp matrix multiply accumulate (WMMA) API [29]. TC is executed in a synchronized manner at the warp level (32 threads). The computable matrix size is represented by $M\times N\times K$, which represents the following matrix operations.

$$\begin{aligned}
&A_{M\times K} \times B_{K\times N} + C_{M\times N} = D_{M\times N} \\
&d_{ij} = c_{ij} + \sum_k a_{ik}b_{kj}
\end{aligned} \tag{4}$$

The Ampere architecture's TC supports various data types, and also supports the computable matrix size depending on the data type shown in Tab. I. We note that while TCs implements $4\times4\times4$ matrix multiplications in hardware, WMMA API allows us only to compute larger matrix multiplications than the minimum size. The large size is the warp-level primitive.

B. Loop Unrolling for SIMD

Single instruction multiple data (SIMD) computing is a parallel processing type categorized by Flynn [40]. SIMD computing simultaneously computes multiple data with a single instruction. In GPU, single instruction multiple threads (SIMT) is a similar computing type. Also, the Tensor Core instruction can compute SIMD type parallelization for the matrix multiply operation. This section reviews SIMD computing for standard computing units such as CUDA Core.

Using SIMD computing, loop unrolling for handling multiple data is essential. In the FIR filtering, there are three loops: image-loop, kernel-loop, and color-loop [41]. Generally, the used image data structure is Array of Structure (AoS), which interleaves multi-channel information in the most inner loop. For example, the RGB image format with AoS data sequence is RGBRGB...RGB. The SIMD unit loads data consecutively for the number of simultaneous calculations;

Algorithm 1 Image loop

Require: image, kernel
  For ($i = 0$, $i <$image.height, $i = i + 1$)
    For ($j = 0$, $j <$image.width, $j = j + 4$)
      acc0=acc1=acc2=acc3=0
      For ($k = 0$, $k <$kernel.size, $k = k + 1$)
        acc0 = acc0+$W(k)I(j + k.x + 0, i + k.y)$
        acc1 = acc1+$W(k)I(j + k.x + 1, i + k.y)$
        acc2 = acc2+$W(k)I(j + k.x + 2, i + k.y)$
        acc3 = acc3+$W(k)I(j + k.x + 3, i + k.y)$

Algorithm 2 Kernel Loop (4 pixels)

Require: image, kernel
  For ($i = 0$, $i <$image.height, $i = i + 1$)
    For ($j = 0$, $j <$image.width, $j = j + 1$)
      acc=0
      For ($k = 0$, $k <$kernel.size, $k = k + 4$)
        acc = acc+$W(k_0)I(j + k_0.x, i + k_0.y)$
        acc = acc+$W(k_1)I(j + k_1.x, i + k_1.y)$
        acc = acc+$W(k_2)I(j + k_2.x, i + k_2.y)$
        acc = acc+$W(k_3)I(j + k_3.x, i + k_3.y)$

thus, loop unrolling color loop requires the number of vector length color channels, e.g., eight channels in CPU's AVX, 16 channels in CPU's AVX, 32 channels in GPU's warp unit. For this case, we usually change the data structure from AoS to the structure of array (SoA) format. The structure has color channels in the most outer loop, e.g., data arrangement is RR...RGG...GBB...B. After SoA transformation, we can unroll data in the image-loop or kernel-loop direction. Note that in the signal channel case, which is the target of this paper, the data structure is SoA in default.

The kernel-loop unrolling merges elements in kernel loops and computes a pixel shown in Algorithm 1. The image-loop unrolling merges elements in image pixel-loops and computes the data with the same kernel value shown in Algorithm 2. Note that image size is $\mathrm{width} \times \mathrm{height}$. Kernel size is also 2D; however, the shape is arbitrary, such as square or circle. Therefore, we represent kernel loop as 1D, which collapses the 2D arbitrary kernel loop into 1D.

IV. Proposed Method

The convolution order represented using TCs is different from the SIMD cases. This paper introduces three cases: kernel-loop unrolling, kernel-image-loop unrolling, and diagonal kernel-image-loop unrolling. Here, we assume that we have 3×3 TCs for ease of explanation.

A. Kernel-loop Unrolling

We introduce the kernel-loop unrolling. The kernel-loop unrolling can be achieved by transposing the kernel matrix and the pixel value matrix as the input matrix. An example of the kernel loop unrolling for 3×3 convolution is shown in Fig. 1. First, we load 3×3 patch with the transpose of image

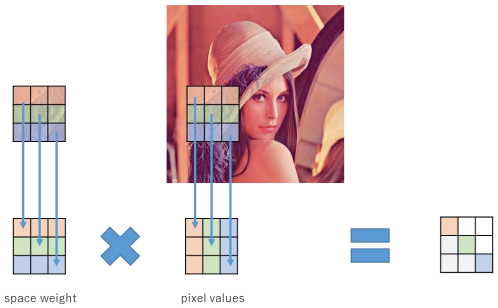

Fig. 1. Kernel-loop unrolling in TC.

and store its relative weights to TCs. Next, we perform matrix multiply to compute simultaneously. Since the corresponding elements of the kernel and pixel value at the pixel of interest are the sum of the products of the n-th row and n-th column, the calculation results of the corresponding elements appear in the diagonal components. Finally, the total of the diagonal elements is the output of the kernel loop unrolling.

Here, we explain the computational flow as a matrix multiply form: $D = A \times B + C$. Let $\boldsymbol{p}_{i,j}$ be the shifted position $\boldsymbol{p} = (x, y)$ by (i, j): $\boldsymbol{p}_{i,j} = (x + i, y + j)$. We here consider matrix-multiply elements as a convolution result. We set a transposed input weight matrix $W(\boldsymbol{p}, \boldsymbol{q})$ as A, and a input image matrix $\boldsymbol{I}(\boldsymbol{q})$ as B. Note that the accumulation matrix C is a zero matrix. The output element $d_{\boldsymbol{p}_{i,j}} \in D$ is calculated as follows:

$$d_{\boldsymbol{p}_{i,j}} = \sum_{k \in \{0,1,2\}} W(\boldsymbol{p}_{k,i}, \boldsymbol{q}_{k,j}) \boldsymbol{I}(\boldsymbol{q}_{k,j}). \tag{5}$$

In the diagonal elements cases, i.e., $i = j, i \in \{0, 1, 2\}$, the equation is as follows:

$$d_{\boldsymbol{p}_{i,i}} = \sum_{k \in \{0,1,2\}} W(\boldsymbol{p}_{k,i}, \boldsymbol{q}_{k,i}) \boldsymbol{I}(\boldsymbol{q}_{k,i}). \tag{6}$$

This means that $d_{\boldsymbol{p}_{i,i}}$ is the result of the convolution of the i-th column: $\boldsymbol{I}(\boldsymbol{p}_{1,1}) = \sum_i d_{\boldsymbol{p}_{i,i}}$.

When we replace the pixel values with the bilateral range weights or bilateral range weights multiplied by pixel values, the convolution becomes the bilateral filter's denominator or numerator, respectively. The numerator is defined as follows:

$$d_{\boldsymbol{p}_{i,i}} = \sum_{k \in \{0,1,2\}} \Big(\omega_s(\boldsymbol{p}_{k,i}, \boldsymbol{q}_{k,i})\Big)\Big(\omega_r(\boldsymbol{I}(\boldsymbol{p}_{k,i}), \boldsymbol{I}(\boldsymbol{q}_{k,i}))\boldsymbol{I}(\boldsymbol{q}_{k,i})\Big). \tag{7}$$

Also, the denominator is defined as follows:

$$d_{\boldsymbol{p}_{i,i}} = \sum_{k \in \{0,1,2\}} \Big(\omega_s(\boldsymbol{p}_{k,i}, \boldsymbol{q}_{k,i})\Big)\Big(\omega_r\big(\boldsymbol{I}(\boldsymbol{p}_{k,i}), \boldsymbol{I}(\boldsymbol{q}_{k,i})\big)\Big). \tag{8}$$

This method only uses diagonal elements; thus, computational efficiency is not high.

B. Kernel-image-loop Unrolling

We introduce the kernel-image-loop unrolling. The kernel-image-loop unrolling using TC is shown in Fig. 2. First, we

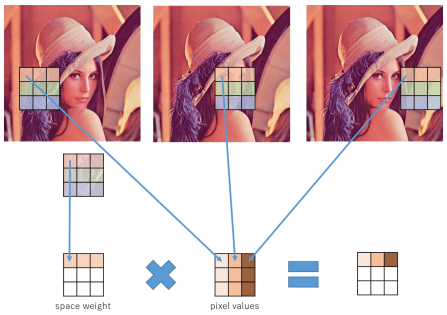

Fig. 2. Kernel-image-loop unrolling in TC.

load the top column of the weight matrix and set the top column. Next, we load the top column of the image in the kernel and then store left rows with transpose. In the next row, we load pixels in a one-pixel slide widow and then store it with transpose. The process is repeated until the matrix of pixel values is filled with all elements. Then, matrix-multiply is performed. After that, the convoluted result of top column is stored in the top column. The flow is defined as follows for $i \in \{0, 1, 2\}$:

$$d_{\boldsymbol{p}_{i,0}} = \sum_{k\in\{0,1,2\}} W(\boldsymbol{p}_{k,i}, \boldsymbol{q}_{k,0})\boldsymbol{I}(\boldsymbol{q}_{k,0_{i,k}}). \tag{9}$$

The process is repeated until the kernel column end. The stride of the unrolling is the radius for the image-loop direction.

Here, we consider the matrix layouts that can be computed with WMMA API. There are three types of FP16/FP32 mixed operations: $16 \times 16 \times 16$, $32 \times 8 \times 16$, and $8 \times 32 \times 16$. For the kernel-image-loop unrolling, only 16 elements of the first line can be used when $16 \times 16 \times 16$ case, while 32 elements of the first line can be used in the $32 \times 8 \times 16$ cases. The kernel-loop unrolling can use up to 16 elements in the $16 \times 16 \times 16$ case; thus, the kernel-image-loop unrolling with the $32 \times 8 \times 16$ tends to be more efficient.

This process can also represent bilateral filter weight. The numerator is defined as follows:

$$d_{\boldsymbol{p}_{i,0}} = \sum_{k\in\{0,1,2\}} \Big(\omega_s(\boldsymbol{p}_{k,i}, \boldsymbol{q}_{k,0})\Big) \Big(\omega_r(\boldsymbol{I}(\boldsymbol{p}_{k,i}), \boldsymbol{I}(\boldsymbol{q}_{k,0_{i,k}}))\boldsymbol{I}(\boldsymbol{q}_{k,0_{i,k}})\Big). \tag{10}$$

The denominator is defined as follows:

$$d_{\boldsymbol{p}_{i,0}} = \sum_{k\in\{0,1,2\}} \Big(\omega_s\big(\boldsymbol{p}_{k,i}, \boldsymbol{q}_{k,0}\big)\Big)\Big(\omega_r(\boldsymbol{I}(\boldsymbol{p}_{k,i}), \boldsymbol{I}\big(\boldsymbol{q}_{k,0_{i,k}}\big))\Big). \tag{11}$$

C. Diagonal Kernel-image-loop Unrolling

We introduce the diagonal kernel-image-loop unrolling, which is an extension of kernel loop unrolling. Here, we consider the convolution of an LTI filter, which has the same weight function for each pixel.

We now consider the unused elements in kernel-loop unrolling. We show the left/right next diagonal elements cases:

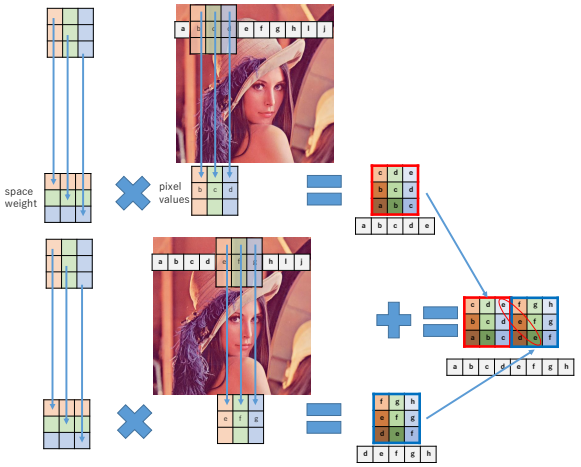

Fig. 3. Diagonal kernel-image-loop unrolling using TC.

TABLE II
RELATIONSHIP OF 3×3 MATRIX-MULTIPLY ELEMENTS BETWEEN INPUT AND REGARDED OUTPUT.

$d_{\boldsymbol{p}_{0,0}}$	$d_{\boldsymbol{p}_{0,1}}$	$d_{\boldsymbol{p}_{0,2}}$	$d_{\boldsymbol{p}_{1,0}}$	$d_{\boldsymbol{p}_{1,1}}$	$d_{\boldsymbol{p}_{1,2}}$	$d_{\boldsymbol{p}_{2,0}}$	$d_{\boldsymbol{p}_{2,1}}$	$d_{\boldsymbol{p}_{2,2}}$
$d_{\boldsymbol{p}_{0,0}}$	$d_{\boldsymbol{p}+1_{0,0}}$	$d_{\boldsymbol{p}+2_{0,0}}$	$d_{\boldsymbol{p}-1_{1,1}}$	$d_{\boldsymbol{p}_{1,1}}$	$d_{\boldsymbol{p}+1_{1,1}}$	$d_{\boldsymbol{p}-2_{2,2}}$	$d_{\boldsymbol{p}-1_{2,2}}$	$d_{\boldsymbol{p}_{2,2}}$

$j = i \pm 1, i \in \{0, 1\}$.

$$d_{\boldsymbol{p}_{i,i\pm1}} = \sum_{k\in\{0,1,2\}} W(\boldsymbol{p}_{k,i}, \boldsymbol{q}_{k,i\pm1})\boldsymbol{I}(\boldsymbol{q}_{k,i\pm1}). \tag{12}$$

$d_{\boldsymbol{p}_{0,1}}$ is the 3×1 convolution result of the top column of the next pixel. $d_{\boldsymbol{p}_{1,2}}$ is the second column of that. Also, $d_{\boldsymbol{p}_{1,0}}$ is the 3×1 convolution result of the second column of the previous pixel. $d_{\boldsymbol{p}2,0}$ is the last column of that. Note that we do not have the last column of the convolution result of the next pixel and the top column of that of the previous pixel.

Next, we consider the second left/right next diagonal elements: $j = i \pm 2, i \in \{0\}$.

$$d_{\boldsymbol{p}_{i,i\pm2}} = \sum_{k\in\{0,1,2\}} W(\boldsymbol{p}_{k,i}, \boldsymbol{q}_{k,i\pm2})\boldsymbol{I}(\boldsymbol{q}_{k,i\pm2}). \tag{13}$$

$d_{\boldsymbol{p}_{0,2}}$ is the 3×1 convolution result of the top column of the next next pixel position, and $d_{\boldsymbol{p}_{2,0}}$ is that of the last column of the previous previous pixel.

Table II shows the relationship of all the elements. To replenish the missing convolution result, we can unroll and convolution with a kernel width stride. Figure 3 show the computational flow. For obtaining the convolution result, we unroll the image-loop at the convolution radius with processing TC and sum the diagonal elements.

The process is limited for the LTI filter. Let consider the convolution of the bilateral filter, which is an LTV filter. Since the bilateral filter is time-invariant, the neighboring elements of the matrix cannot be used to calculate the neighboring pixels. Only the corresponding vectors of the matrices, the diagonal components of the matrices in the case of pixel loop, and the first row of the matrices in the case of kernel loop can be used.

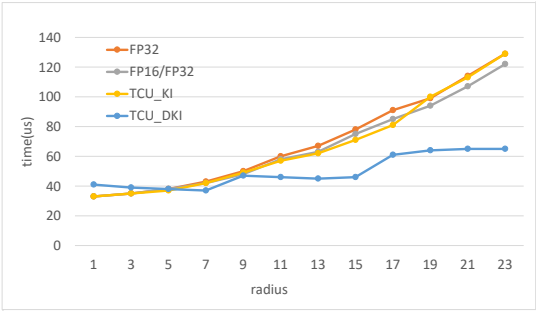

Fig. 4. Processing time of Gaussian filtering.

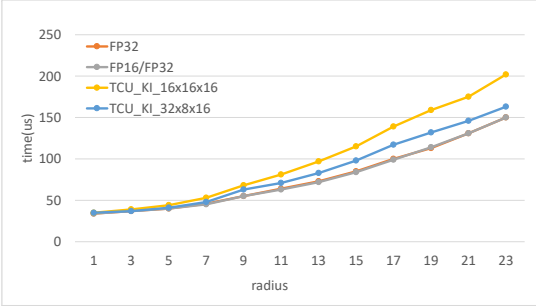

Fig. 5. Processing time of bilateral filtering.

V. Experimental Results

We verified the effectiveness of the proposed loop unrolling for TCs. We used a Gaussian filter as an LTI filter and a bilateral filter as an LTV filter to compare the different convolution methods. Note that Gaussian filtering is a separable filter; thus, we can reduce the computational order to $O(r)$ using separable convolution. However, we did not use separable convolution in our experiment to verify the performance of general LTI filters. The experimental image was a 128×128 grayscale image, and the processing time is measured while the convolution radius r is varied from 1 to 23. Note that we did not measure the convolution radius beyond this value ($r > 23$) because it is difficult to maintain the accuracy in FP16/FP32 mixed-precision arithmetic. The computer used for the measurement was Ryzen9 5900X CPU and GeForce RTX3090 GPU. The accuracy of the data was calculated using a mixed-precision calculation of FP16 input and FP32 accumulation.

For Gaussian filtering, we used the matrix layout of $16 \times 16 \times 16$ with diagonal kernel-image-loop unrolling. Since the LTI filter can compute neighboring pixels simultaneously, the kernel size was used as the stride width in the column direction, and the row direction was computed for all rows. In addition, we use kernel-image-loop unrolling as a competitive.

The bilateral filtering is compared using a kernel-image-loop unrolling that uses only the first row of the matrix in two layouts, $16 \times 16 \times 16$ and $32 \times 8 \times 16$. Since the LTV filter cannot compute adjacent pixels, it is computed for all pixels. In both cases, when $r > 7$, the matrix was split between the loop, and multiple operations were performed using TC.

The execution time of the Gaussian filter with varying kernel size is shown in Fig. 4. The time of diagonal kernel-image (DKI) loop unrolling using TC is shorter than using CUDA Core 16/32-bit cases from $r \geq 7$. In particular, there is a large difference for $r = 7, 15$. The kernel-image (KI) loop unrolling using TC is not efficient since the utilization of TC is not complete.

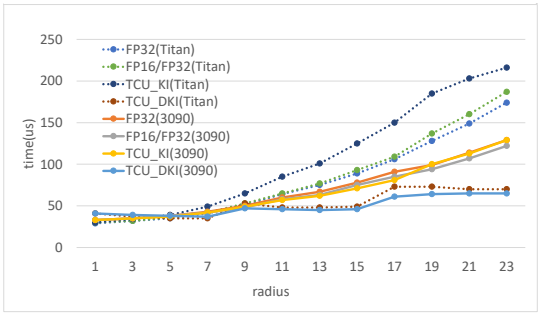

Fig. 6. Processing time of Gaussian filtering on various GPU.

TABLE III
GPU spec. For FP16, FP32, and TC, unit is TFLOPS. #CC and #TC are the number of CUDA Cores and Tensor Cores.

Model	Arch.	Clock	#CC	#TC	FP16	FP32	TC
RTX 3090	Ampere	1.70	10496	328	35.7	35.7	142.7
Titan RTX	Turing	1.77	4608	576	32.6	16.3	130.5

The bilateral filter cases is shown in Fig. 5. Implementing the matrix size of $32 \times 8 \times 16$ was faster than the implementation using the matrix size of $16 \times 16 \times 16$; however, the usual implementation with CUDA core is faster than the TC implementation.

Figure 6 shows the difference between the GeForce RTX3090 (Ampare GA102) and GeForce TITN RTX (Turing). Table III shows the spec of each GPU and the Peak TFLOPS is computed as follows (CC is CUDA Core):

- TuringCC: 4608cores $\times$ 1.77GHz$\times$ 2FMA= 16.312
- AmpereCC: 10496cores $\times$ 1.70GHz$\times$ 2FMA= 35.686
- TuringTC: 576cores $\times$ 1.77GHz$\times$ 2FMA$\times$64[1]= 130.499
- AmpereTC: 328cores $\times$1.70GHz$\times$ 2FMA$\times$128[2]= 142.746

For each case, DKI loop unrolling is the fastest. The difference between TC implementation and CC implementation becomes smaller than the previous generation of GPU due to the ratios of peak FLOPS (TC/CC) are 8 (Turing) and 4 (Ampere).

VI. Conclusion

In this paper, we proposed three loop unrolling method for TCs and verified these effectiveness for image filtering of LTI and LTV filters: Gaussian filtering and bilateral filtering as examples. Time-invariant Gaussian filter has high using rate of the TC with diagonal kernel-image-loop unrolling. The TC calculation results was faster than the CUDA Core calculation for the part with a large kernel radius. In addition, it is shown that the execution time can be shortened by changing the matrix size of TC in the bilateral filter because the matrix can be used efficiently; however, usual CUDA Core parallelization is more efficient than using TC. As our future work, we will develop a DSL for recursive filtering, such as [42], with TC.

[1] 1st/2nd generation TC computes 64 16-bit-elements with FMA per a clock.
[2] 3rd generation TC for RTX3090 computes 128 16-bit-elements with FMA per a clock.

REFERENCES

[1] D. Ziou and S. Tabbone. Edge detection techniques-an overview. *attern Recognition and Image Analysis: Advances in Mathematical Theory and Applications*, 8(4):537–559, 1998.

[2] A. P. Witkin. Scale-space filtering. In *Readings in Computer Vision*, pages 329–332. Elsevier, 1987.

[3] V. Aurich and J. Weule. Non-linear gaussian filters performing edge preserving diffusion. In *Mustererkennung*, pages 538–545. Springer, 1995.

[4] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *IEEE International Conference on Computer Vision (ICCV)*, pages 839–846, 1998.

[5] A. Buades, B. Coll, and J. M. Morel. A non-local algorithm for image denoising. In *Proc. IEEE Computer Vision and Pattern Recognition (CVPR)*, pages 60–65, 2005.

[6] J.-S. Lee. Digital image enhancement and noise filtering by use of local statistics. *IEEE transactions on pattern analysis and machine intelligence*, (2):165–168, 1980.

[7] E. Reinhard, W. Heidrich, P. Debevec, S. Pattanaik, G. Ward, and K. Myszkowski. *High dynamic range imaging: acquisition, display, and image-based lighting*. Morgan Kaufmann, 2010.

[8] R. Fattal. Single image dehazing. *ACM transactions on graphics (TOG)*, 27(3):1–9, 2008.

[9] N. Fukushima, K. Sugimoto, and S. Kamata. Guided image filtering with arbitrary window function. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2018.

[10] S. Paris, S. W. Hasinoff, and J. Kautz. Local laplacian filters: Edge-aware image processing with a laplacian pyramid. *ACM transactions on graphics (TOG)*, 30(4):68, 2011.

[11] T. Matsuo, N. Fukushima, and Y. Ishibashi. Weighted joint bilateral filter with slope depth compensation filter for depth map refinement. In *Proc. International Conference on Computer Vision Theory and Applications (VISAPP)*, 2013.

[12] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[13] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the h. 264/avc video coding standard. *IEEE Transactions on circuits and systems for video technology*, 13(7):560–576, 2003.

[14] G. J. Sullivan, J.-R. Ohm, W.-J. Han, and T. Wiegand. Overview of the high efficiency video coding (hevc) standard. *IEEE Transactions on circuits and systems for video technology*, 22(12):1649–1668, 2012.

[15] B. Bross, J. Chen, J.-R. Ohm, G. J. Sullivan, and Y.-K. Wang. Developments in international video coding standardization after avc, with an overview of versatile video coding (vvc). *Proceedings of the IEEE*, 2021.

[16] R. Deriche. Recursively implementating the gaussian and its derivatives. In *Proc. IEEE International Conference on Image Processing (ICIP)*, pages 263–267, 1992.

[17] L. J. vanVliet, I. T. Young, and P. W. Verbeek. Recursive gaussian derivative filters. In *Proceegings of International Conference on Pattern Recognition (ICPR)*, 1998.

[18] K. Sugimoto and S. Kamata. Fast gaussian filter with second-order shift property of dct-5. In *Proceedings of IEEE International Conference on Image Processing (ICIP)*, 2013.

[19] T. Otsuka, N. Fukushima, Y. Maeda, K. Sugimoto, and S. Kamata. Optimization of sliding-dct based gaussian filtering for hardware accelerator. In *Proc. International Conference on Visual Communications and Image Processing (VCIP)*, 2020.

[20] F. Durand and J. Dorsey. Fast bilateral filtering for the display of high-dynamic-range images. *ACM Transactions on Graphics*, 21(3):257–266, 2002.

[21] K. N. Chaudhury, D. Sage, and M. Unser. Fast o(1) bilateral filtering using trigonometric range kernels. *IEEE Transactions on Image Processing*, 20(12):3376–3382, 2011.

[22] K. Sugimoto and S. Kamata. Compressive bilateral filtering. *IEEE Transactions on Image Processing*, 24(11):3357–3369, 2015.

[23] Y. Maeda, N. Fukushima, and H. Matsuo. Effective implementation of edge-preserving filtering on cpu microarchitectures. *Applied Sciences*, 8(10), 2018.

[24] K. Sugimoto, N. Fukushima, and S. Kamata. 200 fps constant-time bilateral filter using svd and tiling strategy. In *IEEE International Conference on Image Processing (ICIP)*, 2019.

[25] Y. Sumiya, N. Fukushima, K. Sugimoto, and S. i. Kamata. Extending compressive bilateral filtering for arbitrary range kernel. In *Proc. IEEE International Conference on Image Processing (ICIP)*, 2020.

[26] S. Oishi and N. Fukushima. Clustering-based acceleration for high-dimensional gaussian filtering. In *Proc. Signal Processing and Multimedia Applications (SIGMAP)*, 2021.

[27] H. Tajima, N. Fukushima, Y. Maeda, and T. Tsubokawa. Local lut upsampling for acceleration of edge-preserving filtering. In *Proc. International Conference on Computer Vision Theory and Applications (VISAPP)*, 2019.

[28] H. Tajima, T. Tsubokawa, Y. Maeda, and N. Fukushima. Fast local lut upsampling. In *Proc. International Conference on Computer Vision Theory and Applications (VISAPP)*, 2020.

[29] S. Markidis, S. W. Der Chien, E. Laure, I. B. Peng, and J. S. Vetter. Nvidia tensor core programmability, performance & precision. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531. IEEE, 2018.

[30] A. Sorna, X. Cheng, E. D'azevedo, K. Won, and S. Tomov. Optimizing the fast fourier transform using mixed precision on tensor core hardware. In *IEEE International Conference on High Performance Computing Workshops (HiPCW)*, pages 3–7. IEEE, 2018.

[31] R. Chowdhury, F. Silvestri, and F. Vella. A computational model for tensor core units. In *Proc. ACM Symposium on Parallelism in Algorithms and Architectures*, page 519–521. Association for Computing Machinery, 2020.

[32] A. Dakkak, C. Li, J. Xiong, I. Gelado, and W.-m. Hwu. Accelerating reduction and scan using tensor core units. In *Proceedings of the ACM International Conference on Supercomputing*, pages 46–57, 2019.

[33] N. Jouppi, C. Young, N. Patil, and D. Patterson. Motivation for and evaluation of the first tensor processing unit. *IEEE Micro*, 38(3):10–19, 2018.

[34] M. H. Ionica and D. Gregg. The movidius myriad architecture's potential for scientific computing. *IEEE Micro*, 35(1):6–14, 2015.

[35] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics*, 31(4):32:1–32:12, 2012-07.

[36] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proc. ACM Programming Language Design and Implementation (PLDI)*, pages 519–530, 2013.

[37] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics*, 35(4):83:1–83:11, 2016.

[38] S. Sioutas, S. Stuijk, T. Basten, L. Somers, and H. Corporaal. Programming tensor cores from an image processing dsl. In *Proc. nternational Workshop on Software and Compilers for Embedded Systems*, page 36–41. Association for Computing Machinery, 2020.

[39] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro*, 41(2):29–35, 2021.

[40] M. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[41] Y. Maeda, N. Fukushima, and H. Matsuo. Taxonomy of vectorization patterns of programming for fir image filters using kernel subsampling and new one. *Applied Sciences*, 8(8), 2018.

[42] H. Takagi and N. Fukushima. An efficient description with halide for iir gaussian filter. In *Proc. Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA)*, 2020.