TensorCV: Accelerating Inference-Adjacent Computation Using Tensor Processors

Dongho Ha
Yonsei University
Seoul, Korea
dongho.ha@yonsei.ac.kr

Won Woo Ro
Yonsei University
Seoul, Korea
wro@yonsei.ac.kr

Hung-Wei Tseng
University of California, Riverside
Rivsrside, California, USA
htseng@ucr.edu

Abstract—**The advancements in AI/ML accelerators have made the core AI/ML computation relatively insignificant in application pipelines. For example, inferencing only accounts for 3% of the latency in an image-based ML pipeline with the help of Tensor Cores. The mismatch in performance growth between ML model computation and ML-adjacent computation, the producer and consumer of ML models, will become the bottleneck leading to system inefficiency.**

This paper presents a set of innovative algorithms to allow the entire ML-based computer vision pipelines to leverage AI/ML accelerators. Our proposed algorithms feature matrix-based operations that AI/ML accelerators specialize in. Simply compiler optimizations cannot take full advantage of hardware acceleration without revisiting algorithms.

This paper implements the proposed algorithms as an open-source library, TensorCV, in a system platform with Tensor Cores. TensorCV shows a 6.12× speedup in optimized ML-adjacent functions and saves 81% energy consumption on modern heterogeneous computers. The code is available at https://github.com/escalab/TensorCV.

I. Introduction

The broad spectrum of applications that use camera and video inputs to sense the world make computer vision-related workloads one of the essential categories in artificial intelligence (AI) and machine learning (ML). Recent advancements in AI/ML hardware accelerators, including Google's Tensor Processing Units (TPUs), NVIDIA's Tensor Cores, Apple's Neural Engines, Intel's Gaussian Neural Accelerators, etc., have significantly improved the computation time directly related to the core of AI/ML, namely, inference and training. As a result, inferencing a highly optimized NN model can account for less than 3% of the time in computer vision pipelines. Instead, these non-inference code sections consume the majority of execution time in these applications nowadays.

These code sections adjacent to the core ML inference or training process of the computer vision pipeline typically perform operations that help enhance or extract the most critical part of images to enable more accurate and efficient ML results. Unfortunately, the conventional approach typically relies on CPU code whose performance can only scale with the relatively slow improvement through Moore's Law, but not the rapid growth from emerging, innovative hardware accelerators. The inefficiency of these ML-adjacent stages will lead to under-utilized hardware accelerators and become the performance bottleneck, as these stages cannot feed sufficient inputs to well-optimized ML models.

To address the above problems without increasing hardware costs, a potential solution is leveraging existing hardware accelerators (e.g., Tensor Cores, TPUs) for inference/training-adjacent stages. Using hardware accelerators can bring several benefits. (1) These accelerators' microarchitecture can directly compute on higher dimensional datasets, providing more efficient processing models for inference/training-adjacent stages. (2) Using the same training/inference hardware for adjacent computation can remove unnecessary data movement and transformation overhead. (3) The associative property of tensor algebra enables optimizations crossing the boundaries of stages in the pipeline to further reduce operations. (4) By moving more computation into AI/ML accelerators, the system can free up CPUs/GPUs for more meaningful workloads. (5) The system can reclaim the significantly wasted idle power in AI/ML accelerators [20]. (6) The application can significantly reduce the total energy consumption if the execution time is at the same level, while AI/ML accelerators consume lower power than other computing resources.

However, using AI/ML accelerators for non-training/inference tasks is still challenging for the following reasons. First, existing AI/ML accelerators take domain-specific design approaches and abstract their hardware operations in a domain-specific way. Converting existing general-purpose programming language code to use a domain-specific language interface is non-trivial. Second, and probably the most important, due to the difference in micro-architectures and execution models, existing workloads will need to change their algorithms fundamentally to make use of AI/ML accelerators.

In this paper, we demonstrate the application of AI/ML accelerators in accelerating inference/training-adjacent tasks. We revisited the design of frequently used, performance-critical inference-adjacent functions in modern computer vision (CV) pipelines. We proposed matrix/tensor-based algorithms to allow these functions to enjoy the facilities that AI/ML accelerators provide. Though these algorithms potentially have higher algorithmic complexity than existing solutions, these algorithms can still supersede the performance of optimized implementations on modern general-purpose processors since AI/ML accelerators can execute our underlying operations efficiently. Our implementation, TensorCV, shows our algorithms and implementations can achieve 6.12× and save 81% energy consumption compared to the CPU implementation on a desktop computer using the latest CPU and GPU approaches, while the existing GPU implementation shows 2.98× speedup and 64% energy saving.

In presenting TensorCV, this paper makes the following contributions. (1) It proposes a set of algorithms that enable critical image-processing functions on AI/ML accelerators. AI/ML hardware for the same functions is only possible with these algorithms. (2) It implements and evaluates the proposed algorithms on a real system platform to prove the performance benefits. (3) It provides an open-source implementation aligning with the interface of the most popular computer vision library to impact a broad spectrum of CV applications.

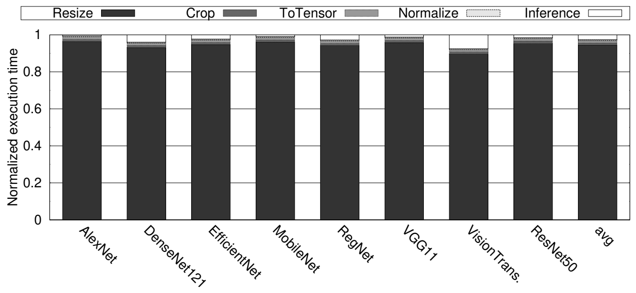

Fig. 1. Latency breakdown when running a MobileNetV3 object detection

TABLE I
OPENCV IMAGE PRE-PROCESSING FUNCTIONS

Function Name	Number of GitHub Code	Description
resize()	>1000k	Resize an image or a video frame.
cvtColor()	682k	Convert an image from one color space to another.
rect()	504k	Define a rectangular region of interest in an image.
normalize()	477k	Normalize an image or a matrix.
rotate()	295k	Rotate an image or a matrix by a specified angle.
Canny()	262k	Perform edge detection on an image.
Sobel()	188k	Perform gradient calculation on an image.
dilate()	182k	Perform morphological dilation on an image.
findContours()	169k	Find the contours in an image.
erode()	158k	Perform morphological erosion on an image.

II. BACKGROUND AND MOTIVATION

A. AI/ML accelerators

AI/ML accelerators have significant presence in modern computer systems as they are more efficient than conventional GPUs in AI/ML tasks for the following reasons. (1) AI/ML accelerators require fewer operations and cycles in performing the same task. For example, each Tensor Core operation can multiply two tiles of matrices in one cycle. In contrast, using GPUs' vector processing model, a matrix multiplication would require a vector element-wise multiplication and an accumulation operation on each pair of rows and columns. When multiplying two 8K $\times$ 8K matrices, tensor cores will take 2^{30} tile MMA operations that require 3.2 million cycles on RTX 3090. Still, conventional CUDA cores will use 2^{39} multiplications and 2^{26} accumulations that require 100 million cycles on the same GPU. (2) As the hardware is specialized for matrix operations, the same circuit area can deliver higher throughput than general-purpose architectures. (3) The design can use smaller circuit areas to reduce power consumption. However, due to AI/ML accelerators' specialization for NN/matrix operations, legacy programs cannot easily take advantage of these accelerators unless their algorithms are presented in matrix algebra.

This paper implements TensorCV algorithms on Tensor Cores for accessibility reasons. Tensor Cores are ubiquitous in NVIDIA's GPU architectures, and NVIDIA made their API available at various level programming frameworks. Conversely, high-performance TPUs are only accessible through Google's cloud services, and Apple's NPUs are only available on their machines without revealing their API to the public. However, as these AI/ML accelerators are essentially matrix processing units, we envision the same algorithm that works efficiently on Tensor Cores would work on other AI/ML accelerators with minimal modifications.

B. Modern CV pipelines and performance bottlenecks

The advantage of avoiding a tremendous amount of manual feature engineering while maintaining high accuracies in classifications and recognitions makes NNs an inevitable component in modern CV applications. In modern AI/ML-assisted CV applications, the application must standardize, shrink and clean up the content before inference because these inference-adjacent operations can reduce both computational operations and memory consumption and improve the accuracy of inference, making NN models more efficient and economically available in applications.

With AI/ML accelerators significantly improve the inference performance, inference-adjacent computation becomes more critical in CV pipelines. Figure 1 shows latency breakdown in running popular image classification applications written in PyTorch and TensorRT libraries. For these CV workloads, the inference-adjacent stages take 4032$\times$3024 images as inputs, resize images to 256$\times$256 (Resize) or crop images to 224$\times$224 (Crop), reshape the tensors (ToTensor), and normalize the images (Normalize). The result shows that inference-adjacent stages account for 97% of overall latency. These applications only spent 3% of time on Tensor Cores for inference.

III. ALGORITHM

As modern AI/ML accelerators cannot work on computation without matrix operations, the most critical task in this paper is revisiting entrenched implementations to promote the use of matrix operations. Therefore, the fundamental idea of this paper is treating each input image as an input matrix $Input$, and our algorithms dynamically create specialized matrices (i.e., matrix kernels) that the algorithm can later perform operations together with $Input$ to achieve equivalent image processing results.

Table I lists the most frequently used image pre-processing functions in Open-CV [2] ranked by their occurrence in public GitHub repositories. Considering the demands of each function in NN applications, this paper targets five functions; `resize`, `cvtColor`, `crop`, `rotate`, and `normalize`. Existing implementations of these functions target conventional CPU/GPU applications and employee scalar or vector computation that AI/ML cannot perform efficiently. The following sections elaborate on the how this paper generates appropriate matrix kernels for the corresponding algorithms.

A. Resize

1) Baseline resize algorithm: Resize is the most frequently used pre-processing function. In AI/ML-assisted CV pipelines, resize function can help the application shrink images to fit the demanding input size of the model. By shrinking input size, resizing helps a training or trained model work with various sizes of input images and, probably the most important, reduce memory consumption and execution time.

In the conventional bi-linear resizing implementation, the code will first compute the relative ratio between input and output matrix sizes, which we call row and column scales ($rowScale$ and $columnScale$). For an input image ($Input$) sized m-by-n, the input matrix size is m-by-$3n$ since each row includes R, G, and B channels. If the target output image size is m'-by-n' the output matrix size is m'-by-$3n'$. Hence, the code computes the row and column scales as follows:

$$rowScale = \frac{m'}{m}, \ \ columnScale = \frac{n'}{n} \tag{1}$$

Then, using the scales, the code iterates through every bounding box containing the source pixels to a target pixel and calculates the weighted average as the target pixel value. The following equations show how the baseline code calculates the boundary $(top_i, bot_i, left_j, right_j)$ of bounding boxes in $Input$ and weights for corresponding output pixel (i, j).

$$\begin{aligned} top_i &= \lfloor \frac{i}{rowScale} \rfloor \\ bot_i &= \lceil \frac{i}{rowScale} \rceil \\ left_j &= \lfloor \frac{\lfloor j/3 \rfloor}{columnScalme} \rfloor \\ right_j &= \lceil \frac{\lfloor j/3 \rfloor}{columnScale} \rceil \\ rowWeight_i &= \frac{i}{rowScale} - top_i, \\ colWeight_j &= \frac{\lfloor j/3 \rfloor}{columnScale} - left_j \end{aligned} \tag{2}$$

The finally, the conventional algorithm will calculates each output pixel (i, j) value as follows:

$$\begin{aligned} output_{i,j} = \\ (1 - rowWeight_i) \cdot (1 - colWeight_j) \cdot Input_{top_i, left_j} + \\ (1 - rowWeight_i) \cdot (colWeight_j) \cdot Input_{top_i, right_j} + \\ (rowWeight_i) \cdot (1 - colWeight_j) \cdot Input_{bot_i, left_j} + \\ (rowWeight_i) \cdot (colWeight_j) \cdot Input_{bot_i, right_j} \end{aligned} \tag{3}$$

However, the algorithm in OpenCV-CUDA cannot exploit Tensor Cores since the there is no matrix multiplications but only exists element-wise weighted averages.

2) TensorCV resize algorithm: TensorCV transforms the conventional implementation into two matrix multiplications. Considering an m-by-$3n$ input and m'-by-$3n'$ output images, the proposed algorithm creates an m'-by-m matrix as L^{resize} and $3n$-by-$3n'$ matrix as R^{resize}. The content of L^{resize} and R^{resize} is only related to the target image's size; therefore, TensorCV only needs to create them once for every batch. In contrast, the conventional approach not only prevents the code from using Tensor Cores but also recalculates weights for the same pixel position in each channel. The proposed algorithm will fill the content of L^{resize} using the following formula:

$$L^{resize}_{m \times m'} \ni l^{resize}_{i,j} = \begin{cases} 1 - rowWeight_j & \text{if} \quad j = top_i \\ rowWeight_j & \text{if} \quad j = bot_i \\ 0 & \text{else} \end{cases} \tag{4}$$

R^{resize} holds column-related weights, and TensorCV's algorithm fills the R^{resize} using the following formula:

$$R^{resize}_{3n \times 3n'} \ni r^{resize}_{i,j} = \begin{cases} 1 - colWeight_i & \text{if} \quad \begin{matrix} \lfloor i/3 \rfloor = left_i \\ \text{and } i \pmod 3 = j \pmod 3 \end{matrix} \\ colWeight_i & \text{if} \quad \begin{matrix} \lfloor i/3 \rfloor = right_i \\ \text{and } i \pmod 3 = j \pmod 3 \end{matrix} \\ 0 & \text{else} \end{cases} \tag{5}$$

After filling matrices L^{resize} and R^{resize} using Equation 4 and 5 at the beginning of each batch of images, the proposed algorithm can compute the output of each image resizing result as the following.

$$Output_{m' \times 3n'} = L^{resize}_{m' \times m} \cdot Input_{m \times 3n} \cdot R^{resize}_{3n \times 3n'} \tag{6}$$

B. Color space conversion

1) Baseline cvtColor algorithm: As the sensor designs vary, image sources may encode pixels differently. Therefore, CV applications must convert the color spaces between the raw image encoding to the RGB color space that computing devices most frequently use. Taking the most common conversion between YUV color space that most video encoders use to RGB color space as an example, each pixel would require several element-wise multiplications with different coefficients and accumulations of scalar values. [9]. The vectorized version in OpenCV-CUDA already expands the functions into matrix-vector multiplications with each pixel as a 1×3 vector and the coefficients (we call matrix C) as a 3×3 matrix as follows:

$$\begin{aligned} (Y \quad U \quad V) = (R \quad G \quad B) \cdot \begin{pmatrix} 0.299 & -0.147 & 0.615 \\ 0.587 & -0.289 & -0.515 \\ 0.114 & 0.436 & -0.100 \end{pmatrix} \\ (R \quad G \quad B) = (Y \quad U \quad V) \cdot \begin{pmatrix} 1 & 1 & 1 \\ 0 & -0.395 & 2.032 \\ 1.140 & -0.581 & 0 \end{pmatrix} \end{aligned} \tag{7}$$

However, the algorithm in OpenCV-CUDA still cannot fully exploit Tensor Cores since each Tensor Core Unit can work on larger sizes of matrices.

2) TensorCV cvtColor algorithm: The proposed algorithm multiplies the original m-by-$3n$ input matrix with $3n$-by-$3n$ $R^{cvtColor}$, which is similar to an identical matrix. Each element in the $R^{cvtColor}$ is filled by coefficient matrix C in the following formula:

$$R^{cvtColor}_{3n \times 3n} \ni r^{cvtColor}_{i,j} = \begin{cases} C_{i \pmod 3, j \pmod 3} & \text{if} \quad \lfloor i/3 \rfloor = \lfloor j/3 \rfloor \\ 0 & \text{else} \end{cases} \tag{8}$$

Still, each three-by-three partial matrix is a coefficient matrix for color space conversion.

$$Output_{m \times 3n} = Input_{m \times 3n} \cdot R^{cvtColor}_{3n \times 3n} \tag{9}$$

C. Cropping

1) Baseline crop algorithm: Cropping (i.e., `rect` in OpenCV) extracts an essential part within an image for the AI/ML model. For a source image with size m-by-n and target image size of m'-by-n' and offset (x, y), the conventional implementation would require at least n' memory operations where each operation copies the length of m' from an offset of x from the beginning of each row. Suppose the cropping operation occurs during the middle stage of the pipeline. In that case, the application typically has to transfer the control to the memory controller and under-utilize the AI/ML hardware.

2) TensorCV crop algorithm: The proposed algorithm keeps the matrix content in matrix units and performs matrix multiplications but requires zero memory operations. Similar to equation 6, the proposed algorithm creates two matrices, $L^{crop}_{m' \times m}$ and $R^{crop}_{3n \times 3n'}$. $L^{crop}_{m' \times m}$ contains an identity matrix sizes m-by-m starting from column y and $R^{crop}_{3n \times 3n'}$ contains an identity matrix sizes $3n$-by-$3n$ starting from row x.

D. Normalize

1) Baseline normalize algorithm: Since the AI/ML models are trained on normalized data, the models must normalize input images before inference. To normalize an image, CV applications calculate the mean ($mean$) and standard deviation ($stddev$) of the input image's pixel values, subtract $mean$ from each pixel value, and divide it by the $stddev$ for each channel. In conventional CV applications, the code calculates mean and standard deviation and conducts normalization as follows:

$$\begin{aligned} mean = \frac{\sum Input_{i,j}}{m \times n}, \quad stddev = \frac{\sqrt{\sum (Input_{i,j} - mean)^2}}{m \times n}, \\ Output_{m \times n} = \frac{Input_{m \times n} - mean}{stddev} \end{aligned} \tag{10}$$

Again, the baseline OpenCV-CUDA algorithm contains no matrix operations for Tensor Cores.

2) TensorCV normalize algorithm: To calculate the mean, TensorCV multiplies 16-by-m matrix $L^{mean}_{16\times m}$ and n-by-16 matrix $R^{mean}_{n\times 16}$ with only ones in the first row and column. We scale the row and column dimensions to $L^{mean}_{16\times m}$ and $R^{mean}_{n\times 16}$ because the current Tensor Core hardware optimizes for 16×8 matrix operations. Since normalization transposes and resizes the matrices, handle all color spaces simultaneously is challenging. Hence, each element in the kernels $L^{mean}_{16\times m}$ and $R^{mean}_{n\times 16}$ are filled in the following formula:

$$L^{mean}_{16\times m} \ni l^{norm}_{i,j} = \begin{cases} 1 & \text{if} \quad i = 0 \\ 0 & \text{else} \end{cases}, \qquad R^{mean}_{n\times 16} \ni r^{norm}_{i,j} = \begin{cases} 1 & \text{if} \quad j = 0 \\ 0 & \text{else} \end{cases} \tag{11}$$

TensorCV then multiplies matrice $L^{mean}_{16\times m}$ and $R^{mean}_{n\times 16}$ with the $Input$ and results in the sum of all pixel values in the first element of the output matrix as follows:

$$Mean_{16\times 16} = \begin{pmatrix} \sum Input_{i,j} & \cdots \\ \vdots & \ddots \end{pmatrix} = L^{mean}_{16\times m} \cdot Input_{m\times n} \cdot R^{mean}_{n\times 16} \tag{12}$$

Unlike the conventional algorithm calculating the $stddev$ as Equation 10, TensorCV calculates $stddev$ using the square root of the variance of data, allowing matrix multiplications in computing $stddev$.

$$stddev = \sqrt{\frac{\sum Input^2_{i,j}}{m \times n} - mean^2} \tag{13}$$

To compute the sum of the squared values of the input matrix, TensorCV multiplies the $Input$ and transposed one, generating the squared input values in the diagonal elements of the output as the following formula:

$$\begin{aligned} StdDev'_{m\times m} &= Input_{m\times n} \cdot Input^T_{m\times n} \\ &= \begin{pmatrix} \sum Input^2_{0,j} & \cdots & \cdots & \cdots \\ \vdots & \sum Input^2_{1,j} & \cdots & \cdots \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \cdots & \sum Input^2_{m-1,j} \end{pmatrix} \end{aligned} \tag{14}$$

To accumulate the diagonal values with matrix multiplication, the TensorCV utilizes an observation that real matrix values are stored in a single-dimension array, and users can define the length and height of the matrix before computing matrix multiplication. Thus, the algorithm add one to the height of the matrix to align the target values in the first row. Consequently, multiplying the output matrix with the transposed matrix $L^{mean}_{16\times m}$ allows calculating the sum of the squared values of the input matrix as the following formula:

$$\begin{aligned} StdDev_{m\times 16} &= \begin{pmatrix} \sum Input^2_{i,j} & \cdots \\ \vdots & \ddots \end{pmatrix} \\ &= StdDev' \begin{pmatrix} \sum Input^2_{0,j} & \sum Input^2_{1,j} & \cdots & \sum Input^2_{m-1,j} \\ \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix} \cdot L^{mean\ T}_{16\times m} \end{aligned} \tag{15}$$

After calculating the sum of pixel values and squared pixel values, the proposed algorithm normalizes $Input$ using pair-wise vector operations, again, not matrix operations as the following equation.

$$\begin{aligned} mean &= \frac{Mean_{16\times 16}[0]}{mn} \\ stddev &= \sqrt{\frac{StdDev_{m\times 16}[0]}{mn} - (\frac{Mean_{16\times 16}[0]}{mn})^2} \\ Output_{m\times n} &= \frac{Input_{m\times n} - mean}{stddev} \end{aligned} \tag{16}$$

E. Rotate

1) Baseline rotate algorithm: To acquire accurate inferencing results, the input images' layout must be exact to its original intent [22]. Thus, CV applications must be able to rotate the input matrix. Taking the most common case, the proposed algorithm supports 90, 180, and 270 degrees of counter-clockwise rotation. The existing CV applications rotate an image by allocating corresponding coordinate values to new coordinates. For example, a 90-degree rotation allocates the (x,y) coordinate values to (y,$width - 1 - x$). In that respect, 180- and 270-degree move (x,y) coordinate values to ($width - 1 - x$,$height - 1 - y$) and ($height - 1 - y$,x), respectively.

2) TensorCV rotate algorithm: We observe that rotation can be represented by swapping x and y coordinates and reversing coordinates ($1 - width - x$ and $1 - height - y$). Thus, the TensorCV algorithm swaps the coordinates by transposing the matrix and reverses by multiplying flipped identical matrices $L^{rotate}_{m\times m}$ and $R^{rotate}_{n\times n}$. The matrices $L^{rotate}_{m\times m}$ and $R^{rotate}_{n\times n}$ are filled in the following formula:

$$L^{rotate}_{m\times m} \ni l^{rotate}_{i,j} = \begin{cases} 1 & \text{if} \quad j = m-1-i, \\ 0 & \text{else} \end{cases}, \qquad R^{rotate}_{n\times n} \ni r^{rotate}_{i,j} = \begin{cases} 1 & \text{if} \quad j = n-1-i, \\ 0 & \text{else} \end{cases} \tag{17}$$

Like normalize, TensorCV performs the rotation operation on each color channel separately (using a W-H/C format) due to the transpose operation, requiring split and merge operations. The following equations depict how the proposed algorithm computes the 90-, 180-, and 270-degree rotations.

$$\begin{aligned} Output90_{n\times m} &= \left[Input_{m\times n} \cdot R^{rotate}_{n\times n}\right]^T \\ Output180_{m\times n} &= L^{rotate}_{m\times m} \cdot Input_{m\times n} \cdot R^{rotate}_{n\times n} \\ Output270_{n\times m} &= \left[L^{rotate}_{m\times m} \cdot Input_{m\times n}\right]^T \end{aligned} \tag{18}$$

F. Kernel Integration

By transforming algorithms into matrix-based ones, TensorCV can treat this series of operations as a series of matrix operations that the system can fuse these functions into a single one to further reduce matrix operations and memory usage. In TensorCV, we demonstrate the fusion of resize, cropping, color space conversion, and rotation into just two matrix multiplications. Considering a code performs resize, center crop, RGB to YUV color space conversion, and 90-degree rotation, we can represent the process as:

$$\begin{aligned} &\Big[L^{crop}_{M''\times M'} \cdot L^{resize}_{M'\times M} \cdot Input_{M\times 3N} \cdot \\ &R^{resize}_{3N\times 3N'} \cdot R^{crop}_{3N'\times 3N''} \cdot R^{cvtColor}_{3N''\times 3N''} \cdot R^{rotate}_{3N''\times 3N''}\Big]^T \end{aligned} \tag{19}$$

As mentioned in Section III-A2, the values of two left kernels and four right kernels are only related to the target image's size and function parameters, not input values. Thus, TensorCV simply needs to compute the matrix multiplications between kernels once for every batch.

$$\left[L^{integ}_{M''\times M'} \cdot Input_{M\times 3N} \cdot R^{integ}_{3N\times 3N''}\right]^T \tag{20}$$

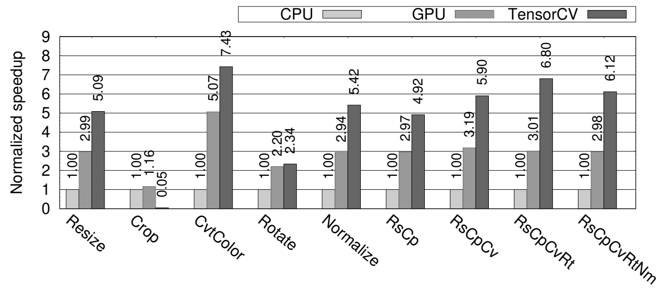

Fig. 2. Performance of TensorCV compared with baseline OpenCV implementation using CPUs and GPUs

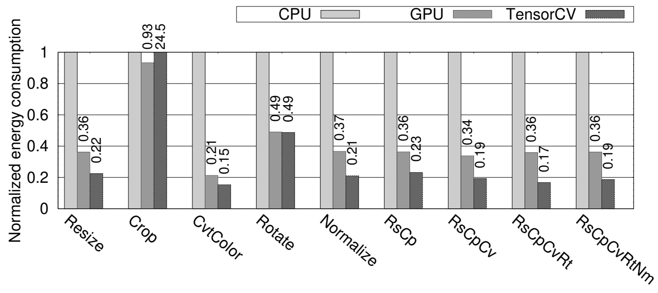

Fig. 3. Energy consumption of TensorCV compared with baseline OpenCV implementation using CPUs and GPUs

One issue is that naively extending $R^{rotate}_{N''\times N''}$ to $R^{rotate}_{3N''\times 3N''}$ generates the incorrect output image. TensorCV solves such an issue by transposing the matrix first. Then, the integrated kernels use modified $R^{cvtColor}_{3N''\times 3N''}$ to convert an RGB format matrix to VUY one. After color space conversion, $R^{rotate}_{3N''\times 3N''}$ flips VUY to YUV, and the transpose operation generates YUV format output. And finally, transpose the result back to the original format. As such, the algorithm fulfills $R^{cvtColor}_{3N''\times 3N''}$ and $R^{rotate}_{3N''\times 3N''}$ as follows:

$$\begin{aligned}
R^{cvtColor\_integ}_{3n\times 3n} \ni r^{cvtColor\_integ}_{i,j} &= \begin{cases} r^{cvtColor}_{i,3j+2} & \text{if} \quad \lfloor j/n \rfloor = 0 \\ r^{cvtColor}_{i,3(j-n)+1} & \text{if} \quad \lfloor j/n \rfloor = 1 \\ r^{cvtColor}_{i,3(j-2n)} & \text{if} \quad \lfloor j/n \rfloor = 2 \end{cases}, \\
L^{rotate\_integ}_{m\times m} \ni l^{rotate\_integ}_{i,j} &= \begin{cases} 1 & \text{if} \quad j = m-1-i \\ 0 & \text{else} \end{cases}, \\
R^{rotate\_integ}_{3n\times 3n} \ni r^{rotate\_integ}_{i,j} &= \begin{cases} 1 & \text{if} \quad j = 3n-1-i \\ 0 & \text{else} \end{cases}
\end{aligned} \tag{21}$$

IV. Experimental Methodology

We conducted experiments on a machine with an Intel Core i7-12700K processor, 64 GB DDR5 DRAM. The GPU in our experiments is an NVIDIA GeForce RTX 3090 GPU based on Ampere architecture. We implemented TensorCV is implemented using NVIDIA CUDA Toolkit 11.7 and cuBLAS in IEEE 754 half precision. The system runs a Linux 5.15.0 kernel. We ran each function with 100 batches of images, where each batch had 20 samples. The inference-adjacent tasks include resizing the image from 4032×3024 to 256×256, cropping the center of the resized image with 224×224 box, converting the color space RGB to YUV, rotating the image 90-degree counter-clockwise, and normalizing the image values. Note that the applications RsCp, RsCpCv, RsCpCvRt, and RsCpCvRtNm indicate integrated functions of Resize (Rs), Crop (Cp), Color space conversion (Cv), Rotate (Rt), and Normalize (Nm).

V. Results

This section summarizes our evaluation of TensorCV. TensorCV delivered 6.12× speedup in RsCpCvRtNm functions of CV pipelines and saved 81% of the energy.

Figure 2 compares the performance of TensorCV with conventional OpenCV implementations that can only use CPUs or CUDA cores. The baseline of Figure 2 is the CPU-based implementation. TensorCV's algorithm achieves up to 7.43× speedup in the color space conversion function. Note that TensorCV shows a performance drop in the center crop because OpenCV implements the crop with simple memory operations. However, TensorCV also can employ such an approach when it needs to run a single crop, and integrating multiple functions removes the high latency of the crop. Excepting the outlier, center crop, TensorCV achieved 4.67× speedup on average, while OpenCV-CUDA only sped up 3.14× on average.

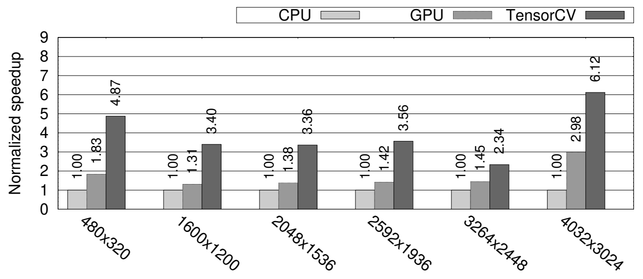

Fig. 4. Performance of RsCpCvRtNm TensorCV function in different input image sizes

Another advantage of TensorCV is the ability to optimize across functions and combine several matrix multiplications into fewer ones. We presented four different use cases that combine multiple pre-processing functions. Compared with other implementations, TensorCV achieves 5.93× geometric mean in speedup in integration functions. However, the performance is limited in the conventional GPU implementation as 3.04× speedup, where cross-function optimization is complicated.

Tensor Cores also share the power/energy advantages of other AI/ML accelerators. We use a Watts Up Power Meter to measure the system power. While running these functions using Tensor Cores, the total system power peaked at 203 W. However, the total system power reaches 192 W and 178 W when using CUDA cores and CPU only, respectively. As TensorCV reduces execution time, TensorCV receives huge benefits in energy consumption. Figure 3 compares the energy consumption of TensorCV with its counterparts. TensorCV saves 81% of energy on the RsCpCvRtNm function. In contrast, the existing GPU implementation of OpenCV-CUDA shows only 64% energy saving.

To demonstrate that TensorCV delivers performance advantages regardless of the image size, Figure 4 shows the normalized speedup of TensorCV in a variety of input image sizes. Our evaluation opts for the input image sizes from the default photo sizes of Apple iPhones. TensorCV achieves 4.04× speedup compared to OpenCV implementation on average, while OpenCV-CUDA leads to 1.70× speedup. Moreover, compared to the OpenCV-CUDA implementation, TensorCV shows at least 1.6× higher speedup in all image sizes with 2.36× speedup on average.

VI. Related Work

In addition to aforementioned related work, several other lines of TensorCV-relevant research deserve mention.

Inference-adjacent computation Prior work identified inference-adjacent computation as the bottleneck in many CV applications [3], [10], [13], [19]. However, most prior work focus on scheduling inference-adjacent stages and managing computing resources, not accelerating the inference-adjacent stage itself. Kang et al. [10] propose a runtime engine for efficient resource management and scheduling for ML-adjacent stages. DLBooster [3] offers an FPGA design to selectively offload and compute some critical decoding workloads to provide high performance in inferencing. Tf.Data [13] proposes a framework for building and executing efficient ML-adjacent stages, allowing the user to schedule, compose, and reuse the computations. FastFlow [19] proposes an ML training system offloading some ML-adjacent computations to remote CPUs to mitigate the bottlenecks.

Image processing on GPUs With the growing interest in GPUs as a general-purpose parallel computing platform, there are studies and frameworks exploiting GPUs in image processing algorithms [1], [2], [7], [15]. However, as mentioned in Section II, ML-adjacent stages can be the bottleneck of ML applications, although such parallelized algorithms, because matrix multiplication accelerators can accelerate only AI/ML parts. Although there are a few studies that employ matrix multiplication accelerators for image processing algorithms [6], [17], they focus on accelerating convolution-based algorithms only. In contrast, TensorCV proposes novel algorithms to utilize matrix multiplication accelerators on ML-related image processing algorithms.

Non-AI/ML applications on AI/ML accelerators Prior work demonstrates that exploiting matrix multiplication accelerators in non-AI/ML algorithms by rewriting the algorithms can improve their latency and throughput [4], [5], [8], [11], [12], [14], [16], [18]. They target reduction operation [4], [14], fractal processing [16], database operations [8], stencil computation [12], Fourier transform [5], [11], [18], and general tensor computation [21]. To our best knowledge, TensorCV is the first research exploiting matrix multiplication accelerator on ML-related image processing algorithms.

VII. Conclusion

This paper revisited the algorithms of several most frequently used and under-optimized inference-adjacent functions in CV pipelines. We showed an average of 7.6× compared with state-of-the-art GPU implementations. Furthermore, the energy efficiency is solid – an average of 82% energy saving. As the first work that utilizes Tensor Cores for inference-adjacent computation in CV pipelines, we envision this work would encourage revisits to existing problems and bring more discussions on related topics.

VIII. Acknowledgment

This research was supported by the Super Computer Development Leading Program of the National Research Foundation of Korea(NRF) funded by the Korean government (Ministry of Science and ICT(MSIT)) (2021M3H6A1017683), Intel, and National Science Foundation (NSF) award, CNS-2007124.

References

[1] Yannick Allusse, Patrick Horain, Ankit Agarwal, and Cindula Saipriyadarshan. Gpucv: An opensource gpu-accelerated framework forimage processing and computer vision. In *Proceedings of the 16th ACM International Conference on Multimedia*, MM '08, page 1089–1092, New York, NY, USA, 2008. Association for Computing Machinery.

[2] Gary Bradski. The opencv library. *Dr. Dobb's Journal: Software Tools for the Professional Programmer*, 25(11):120–123, 2000.

[3] Yang Cheng, Dan Li, Zhiyuan Guo, Binyao Jiang, Jinkun Geng, Wei Bai, Jianping Wu, and Yongqiang Xiong. Accelerating end-to-end deep learning workflow with codesign of data preprocessing and scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 32(7):1802–1814, 2021.

[4] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu. Accelerating reduction and scan using tensor core units. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, pages 46–57, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Sultan Durrani, Muhammad Saad Chughtai, Mert Hidayetoglu, Rashid Tahir, Abdul Dakkak, Lawrence Rauchwerger, Fareed Zaffar, and Wen-mei Hwu. Accelerating fourier and number theoretic transforms using tensor cores and warp shuffles. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 345–355, 2021.

[6] Stefan Groth, Jürgen Teich, and Frank Hannig. Efficient application of tensor core units for convolving images. In *Proceedings of the 24th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '21, page 1–6, New York, NY, USA, 2021. Association for Computing Machinery.

[7] Robert Haase, Loic A Royer, Peter Steinbach, Deborah Schmidt, Alexandr Dibrov, Uwe Schmidt, Martin Weigert, Nicola Maghelli, Pavel Tomancak, Florian Jug, et al. Clij: Gpu-accelerated image processing for everyone. *Nature methods*, 17(1):5–6, 2020.

[8] Yu-Ching Hu, Yuliang Li, and Hung-Wei Tseng. Tcudb: Accelerating database with tensor processors. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1360–1374, New York, NY, USA, 2022. Association for Computing Machinery.

[9] Jeffrey A. Clark. Python Imaging Library. https://github.com/python-pillow/Pillow/blob/main/src/libImaging/ConvertYCbCr.c, 2021.

[10] Daniel Kang, Ankit Mathur, Teja Veeramacheneni, Peter Bailis, and Matei Zaharia. Jointly optimizing preprocessing and inference for dnn-based visual analytics. *Proc. VLDB Endow.*, 14(2):87–100, oct 2020.

[11] Binrui Li, Shenggan Cheng, and James Lin. tcfft: A fast half-precision fft library for nvidia tensor cores. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–11, 2021.

[12] Xiaoyan Liu, Yi Liu, Hailong Yang, Jianjin Liao, Mingzhen Li, Zhongzhi Luan, and Depei Qian. Toward accelerated stencil computation by adapting tensor core unit on gpu. In *Proceedings of the 36th ACM International Conference on Supercomputing*, ICS '22, New York, NY, USA, 2022. Association for Computing Machinery.

[13] Derek G. Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. Tf.data: A machine learning data processing framework. *Proc. VLDB Endow.*, 14(12):2945–2958, jul 2021.

[14] Cristóbal A. Navarro, Roberto Carrasco, Ricardo J. Barrientos, Javier A. Riquelme, and Raimundo Vega. Gpu tensor cores for fast arithmetic reductions. *IEEE Transactions on Parallel and Distributed Systems*, 32(1):72–84, 2021.

[15] In Kyu Park, Nitin Singhal, Man Hee Lee, Sungdae Cho, and Chris Kim. Design and performance evaluation of image processing algorithms on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):91–104, 2011.

[16] Felipe A. Quezada, Cristóbal A. Navarro, Nancy Hitschfeld, and Benjamin Bustos. Squeeze: Efficient compact fractals for tensor core gpus. *Future Generation Computer Systems*, 135:10–19, 2022.

[17] Savvas Sioutas, Sander Stuijk, Twan Basten, Lou Somers, and Henk Corporaal. Programming tensor cores from an image processing dsl. SCOPES '20, page 36–41, New York, NY, USA, 2020. Association for Computing Machinery.

[18] Anumeena Sorna, Xiaohe Cheng, Eduardo D'Azevedo, Kwai Won, and Stanimire Tomov. Optimizing the fast fourier transform using mixed precision on tensor core hardware. In *2018 IEEE 25th International Conference on High Performance Computing Workshops (HiPCW)*, pages 3–7, 2018.

[19] Taegeon Um, Byungsoo Oh, Byeongchan Seo, Minhyeok Kweun, Goeun Kim, and Woo-Yeon Lee. Fastflow: Accelerating deep learning model training with smart offloading of input data pipeline. *Proc. VLDB Endow.*, 16(5):1086–1099, mar 2023.

[20] Abenezer Wudenhe and Hung-Wei Tseng. TPUPoint: Automatically Characterizing Hardware Accelerated Data Center Machine Learning Program Behavior. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS 2021, 2021.

[21] Yunan Zhang, Po-An Tsai, and Hung-Wei Tseng. Simd2: A generalized matrix instruction set for accelerating tensor computation beyond gemm. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 552–566, New York, NY, USA, 2022. Association for Computing Machinery.

[22] Yue Zhou, Xue Yang, Gefan Zhang, Jiabao Wang, Yanyi Liu, Liping Hou, Xue Jiang, Xingzhao Liu, Junchi Yan, Chengqi Lyu, Wenwei Zhang, and Kai Chen. Mmrotate: A rotated object detection benchmark using pytorch. In *Proceedings of the 30th ACM International Conference on Multimedia*, MM '22, page 7331–7334, New York, NY, USA, 2022. Association for Computing Machinery.