Computing Large 2D Convolutions on GPU Efficiently with the im2tensor Algorithm

Mickael Seznec, Nicolas Gac, François Orieux, Alvin Sashala Naik

Noname manuscript No.
(will be inserted by the editor)

Computing Large 2D Convolutions on GPU Efficiently with the im2tensor Algorithm

Mickaël Seznec · Nicolas Gac · François Orieux · Alvin Sashala Naik

the date of receipt and acceptance should be inserted later

Abstract Attaining the best possible throughput when computing convolutions is a challenge for signal and image processing systems, be they HPC (High-Performance Computing) machines or embedded real-time targets. This importance is highlighted by the numerous methods and implementations available, often optimized for particular settings: small batched kernels or very large kernels, for example. In the meantime, GPUs (Graphics Processing Units) have become a first-class architecture for real-time and embedded processing. The power offered by those chips stems from their parallel nature, and this advantage has been exploited for convolutions in several libraries. Even more recently, the introduction of tensor cores on NVIDIA GPUs has opened up new limits in terms of attainable FLOPS (Floating-Point Operations per Second). For reaching that performance, GPU applications must use GEMMs (GEneral Matrix Multiplications), that tensor cores accelerate. We then developed an efficient GEMM-based 2D convolution algorithm in a general setting. On relatively large kernels (30 $\sim$ 50-pixel wide), *im2tensor* is, to the best of our knowledge, the fastest method for computing 2D convolutions. We provide detailed performance analysis for different scenarios: small (1024×1024) and large (4096×4096) images, with convolutions kernels of sizes ranging 1 to 60-pixel wide, on two GPU cards: Jetson AGX Xavier (embedded) and Titan V (server-class). Moreover, the accuracy of *im2tensor* surpasses non-GEMM based methods, thanks to the larger-precision registers used by tensor cores for intermediate representations.

Keywords Image Convolution · Hardware Acceleration · GPU Optimisation · Image Processing Systems · GPU Tensor Cores

1 Introduction

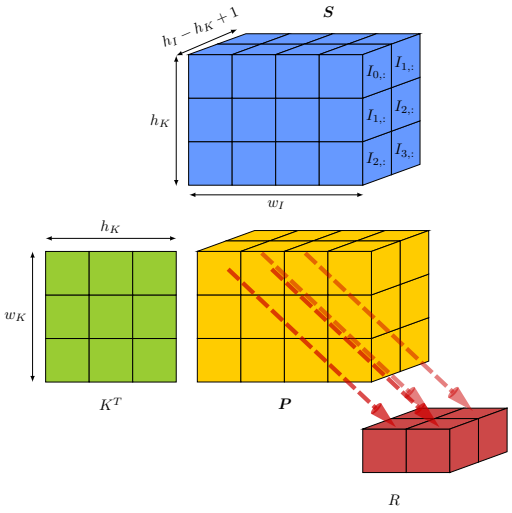

Fig. 1: *im2tensor* achieves 2D convolution of image I and kernel K with a matrix-tensor product $\boldsymbol{P} = K^T \boldsymbol{S}$ then sums along its diagonals (red arrows).

Convolutions are a fundamental tool of signal processing. When applied to images, they serve for template-matching methods [7], edge detection [31], or noise reduction [28]. Convolution is such a ubiquitous operation that much work has been devoted to speeding up its execution on modern computers.

Mickaël Seznec · Alvin Sashala Naik
Thales Research and Technology. Palaiseau, France

Mickaël Seznec · Nicolas Gac · François Orieux
Laboratoire des Signaux et Systèmes, Université Paris-Saclay, CNRS, CentraleSupélec. Gif-Sur-Yvette, France

In its most basic form, computing a 2D convolution can be done with nested loops that perform a multiply-and-add routine for each resulting coefficient. This strategy is simple but has a computational complexity growing in $\mathcal{O}(N^2M^2)$, with an $N \times N$ image and a $M \times M$ kernel. Another method, based on the convolution theorem in the Fourier space, has a $\mathcal{O}(N^2 \log(N))$ complexity. While more sustainable for large kernels, that strategy introduces significant overhead for smaller convolutions [26]. Finding more efficient ways of doing 2D convolution is then a challenging topic that would benefit many real-time image processing applications.

In the meantime, new processor architectures, such as GPUs (Graphics Processing Units), have been widely featured on embedded systems [**?**]. Those chips are used extensively for compute-heavy workloads, and the current trend is to feature accelerators for specific operations [13], such as hardware matrix-multiplication units[**?**]. On GPUs, tensor cores have been introduced starting on NVIDIA Volta and accelerate applications such as DNNs (Deep Neural Networks) or, more generally, BLAS (Basic Linear Algebra Set) operations [6].

More than just plain GEMM programs, other algorithms may benefit from tensor cores by being re-expressed as matrix-multiplications. In that regard, a recent corpus of work has been devoted to using tensor cores, either for parallel primitives [19,9,12], image processing via a DSL (Domain-Specific Language) [27] or CT reconstruction [20], for example.

Regarding convolutions as GEMMs, algorithms exist but target a specific setting, where a convolution between a 3D tensor and multiple small kernels is performed. This situation is usually found within CNNs (Convolutional Neural Networks). *im2col*, explained in [**?**], targets this very setting. It has been implemented in the CUDNN library and leverages tensor cores. However, non-deep learning applications, like large spatial gradients or Gaussian blurs, do not use batches of kernels at once. In those cases, *im2col* might not be the most efficient method.

In this article, we then introduce *im2tensor*, shown on fig. 1, a new 2D convolution algorithm designed to take advantage of tensor cores in a general setting. Our contributions are the following:

- A description of *im2tensor* algorithm for 2D convolutions. It is a composition of a sequence of matrix multiplications and summations on the diagonals. Expressed in this form, the 2D convolution can leverage matrix-multiplication units.
- A CUDA implementation on Nvidia Titan V and Jetson Xavier. It serves to demonstrate the soundness of the algorithm under a GPU environment. We show the different challenges raised by using tensor cores and how to get the maximum performance out of them.
- A comparaison with state-of-the-art methods such as CUDNN, CUFFT, and ArrayFire [33]. We compare *im2tensor* results in terms of speed and accuracy. Our proposed algorithm is fastest on a large range of kernel sizes and is one of the most accurate methods.

Section 2 presents related work on convolution algorithms and an overview of GPU programming and tensor cores. Section 3 explains in detail the *im2tensor* algorithm and provides a proof that is it equivalent to a 2D convolution. Section 4 goes through the details of the CUDA implementation. It reviews the different strategies used to minimize the runtime of the algorithm. Section 5 provides a comparison between our method and several state-of-the-art implementations. The evaluation is done in two different contexts: embedded (30W) and desktop (500W). Results are given in terms of speed and accuracy with respect to a reference implementation. Finally, section 6 concludes this paper and offers directions to follow for further work.

2 Background

2.1 2D Convolutions

Convolutions are a fundamental tool of signal processing. When applied to images, it serves for template-matching methods [7], edge detection [31], or noise reduction [28]. Convolution is such a ubiquitous operation that a lot of work has been devoted to speed up its execution on modern computers:

Separable convolutions If the convolution kernel K can be written as the outer product of two vectors $K = \boldsymbol{k}_1 \boldsymbol{k}_2^T$, the convolution can be performed in two steps: $R = (I * \boldsymbol{k}_1) * \boldsymbol{k}_2$. This technique reduces the overall memory pressure but is restricted to particular kernels.

Convolutions in the Fourier space A convolution can be computed with an element-wise multiplication of the Fourier transforms of the image and the kernel. This product should then undergo an inverse Fourier transform. This technique is asymptotically faster than convolutions in the direct space but may be slower for large images and small kernels [26].

Winograd convolutions This category groups several methods that build optimal algorithms in terms

of arithmetic complexity. In [17], Lavin *et al.* first presented a GPU implementation said to be Winograd-based [32]. Like the Fourier method, it requires the input image and kernel to be transformed, pointwise multiplied and then be inverse-transformed. It has been shown to perform well in DNN for small convolution kernels but is also sensitive to numerical instability [4].

Overlap and Add This technique follows the divide-and-conquer strategy: first, divide the input image into smaller images. Then, compute the convolution of all smaller images with the original kernel. Then recombine the full image and sum where the results overlap. It can be applied to different algorithms [2].

GEMM-based techniques GEMM strategies are motivated by heavily optimized libraries for matrix multiplication (openBLAS, cuBLAS). *im2col* is one such GEMM-based algorithm. First, it flattens the kernel into a vectorn. Second, it constructs a matrix based the image's coefficients so that the vector-matrix product effectively computes the convolution [8, **?**]. This method yields best performance when processing multiple kernels at once. The vector representations are stacked into a matrix. Computing multiple convolutions at once is hence transformed into a matrix-matrix multication, that leverages more of the GPU compared to vecor-matrix. Anderson *et al.* [3] extend the *im2col* idea to different layouts.

The design of *im2tensor* has been guided by other GEMM-based techniques. The unique feature of our algorithm is that it exhibits matrix-matrix multiplications even with a single kernel. This situation is found in several computer vision algorithms, such as Harris corner detection or SURF features that use differential or gaussian kernels [**?**,**?**].

2.1.1 Notations

In this article, a lowercase a denotes a coefficient, the bold lowercase $\boldsymbol{a}$ is a vector, the uppercase A is a matrix, and the bold uppercase $\boldsymbol{A}$ is a three-dimensional tensor. $A_{i,j}$ denotes the coefficient in the i-th row, j-th column of A. The colon notation selects all elements in a dimension: $A_{i,:}$ is the entire i-th row of A.

Let I be an image of size (h_I, w_I) and K be the kernel of size (h_K, w_K). Let R, of shape (h_R, w_R), be the convolution of I by K, defined by:

$$\forall i \in [\![0, h_I - h_K]\!], \forall j \in [\![0, w_I - w_K]\!], \tag{1}$$

$$R_{i,j} = \sum_{y=0}^{h_K-1} \sum_{x=0}^{w_K-1} K_{y,x} I_{i+y,j+y} \tag{2}$$

Formally, this definition is a cross-correlation. For the sake of simplicity, it is, anyway, called a convolution throughout this article. The *real* convolution can be computed by cross-correlating the image with the reversed kernel. With our definition, the result's dimensions are $(h_R, w_R) = (h_I - h_K + 1, w_I - w_K + 1)$. To adhere to numpy and Matlab conventions, this is the so-called "valid" convolution. A "full" convolution yields a $(h_I + h_K - 1, w_I + w_K - 1)$ output while "same" produces a (h_I, w_I) result. The latter two methods require conditions on the borders. They can be computed by doing a *valid* convolution on a pre-padded image. Figure 2 introduces the above notations.

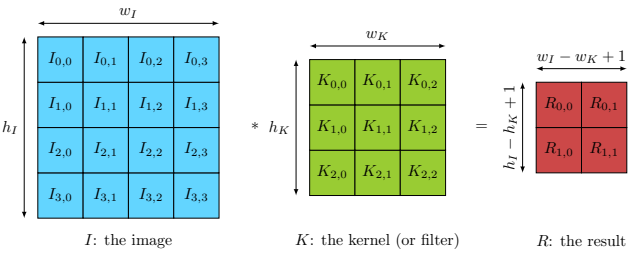

Fig. 2: The *valid* convolution between I and K.

2.2 GPU Programming

GPUs were initially designed for efficient production and display of images on computer screens. They first achieved this goal by means of hardware-fixed functions such as rasterization and pixel shading. With the ever-growing interest in such a powerful processor, GPUs became more flexible and open to general computation. In 2007, Nvidia released the CUDA language that made GPUs handy as a compute platform.

From the software perspective, CUDA bases its programming model on a SIMT (Single Instruction, Multiple Threads) paradigm, a variant of SIMD (Single Instruction, Multiple Data). The programmer writes a single program and specifies how many threads should run it. Threads are partitioned into Thread Blocks (TB) of a customizable size where threads can be synchronized using barrier instructions and share data efficiently through shared memory. This memory location is used in our implementation to share partial matrix multiplication results.

See [16, 25, 15] for more information about GPU and CUDA programming.

2.3 Tensor Cores

Tensor cores are recent additional hardware built into the *Volta* (2017) and later GPU architectures [21–23].

These special units compute a matrix multiplication and accumulation: $D = AB + C$ as shown in fig. 3. While tensor cores operate on 4×4 matrices at the hardware level, the ISA (Instruction Set Architecture) of NVIDIA GPUs provides instructions for larger operand sizes. This is made possible by combining block matrices operations.

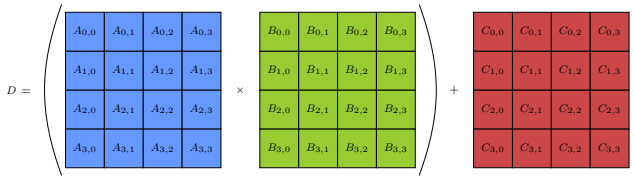

Fig. 3: The matrix multiplication-and-accumulation operation made by a tensor core.

The operands A, B, C, and D may be stored with several numerical precisions. The supported formats depend on the generation of the GPU. Volta GPUs brought the first generation of tensor cores and only supported *fp16* precision (IEEE 754 *binary16*). Turing and Ampere added support for other input types while keeping backward compatibility for already supported precisions. The output type is usually the same as the input matrices, except for *fp16* inputs, where the user chooses between an *fp16* or *fp32* output.

Several authors have explored the arithmetic accuracy of such reduced or mixed-precision operations [14, 18, 5]. Several experiments are conducted later in this article to assess those effects on our convolution algorithm.

3 The im2tensor algorithm

This section introduces our novel algorithm to compute a *valid* convolution between an image I and a kernel K, as defined in section 2.1.1. The algorithm has two main components, the definition of a 3D tensor $\boldsymbol{S}$ that holds row selections of I and a sum operation along its diagonal with the $\mathbf{Tr}_+$ operation.

First, let $\boldsymbol{S}$ be a tensor of shape $(h_S, w_S, d_S) = (h_K, w_I, h_I - h_K + 1)$, such that:

$$\forall i \in [\![0, h_K - 1]\!], \forall j \in [\![0, w_I - 1]\!], \forall k \in [\![0, h_I - h_K]\!], \quad \boldsymbol{S}_{i,j,k} = I_{i+k,j} \tag{3}$$

Then, we define the $\mathbf{Tr}_+$ operation, a sum along the tensor diagonals, like:

$$\mathbf{Tr}_+ : \mathbb{R}^{h\times w\times d} \to \mathbb{R}^{(w-h+1)\times d} \tag{4}$$

$$\boldsymbol{A} \mapsto A, A_{i,j} = \sum_{k=0}^{h-1} A_{k,i+k,j}, \tag{5}$$

$$\forall i \in [\![0, w - h]\!], \forall j \in [\![0, d]\!] \tag{6}$$

Let us now show that $\mathbf{Tr}_+(K^T\boldsymbol{S})$ is, indeed, the "valid" convolution between I and K:

Proof

$$\left[\mathbf{Tr}_+(K^T\boldsymbol{S})\right]_{i,j} = \sum_{x=0}^{w_K-1} (K_T\boldsymbol{S})_{x,i+x,j} \tag{7}$$

$$= \sum_{x=0}^{w_K-1}\sum_{y=0}^{h_K-1} K^T_{x,y}\boldsymbol{S}_{y,i+x,j} \tag{8}$$

$$= \sum_{x=0}^{w_K-1}\sum_{y=0}^{h_K-1} K_{y,x}I_{j+y,i+x} \qquad \square \tag{9}$$

Figure 1 illustrates the $K^T\boldsymbol{S}$ product and the application of the $\mathbf{Tr}_+$ operation, shown with red arrows. We call *im2tensor* the algorithm that uses such a tensor $\boldsymbol{S}$ and the $\mathbf{Tr}_+$ operation to compute a convolution. It is well-suited for parallel computation and exhibits a matrix-tensor product that may leverage specialized hardware units such as tensor cores.

4 Efficient GPU implementation

The previous section introduced the mathematical reasoning behind *im2tensor*. Many hardware architectures could benefit from computing convolution as a matrix-tensor multiplication. In this section, we restrict ourselves to using NVIDIA GPUs and their tensor cores. It allows us to verify the interest of the algorithm on a readily available platform.

4.1 Overview

For the initial version, we implement *im2tensor* with two CUDA kernels. One for the matrix-tensor product, one for the $\mathbf{Tr}_+$ operation. We divide those two tasks into independent pieces of work, handled by CUDA thread blocks (TBs). For the matrix-tensor operation, the $K^T\boldsymbol{S}$ tensor is split into different sub-tensors (see fig. 4) called $\boldsymbol{P}_{\text{block}}$. For $\mathbf{Tr}_+$, we process all diagonals in parallel. Algorithm 1 details the structure of the algorithm.

An important implementation feature is to avoid the explicit construction of $\boldsymbol{S}$. This is made possible by

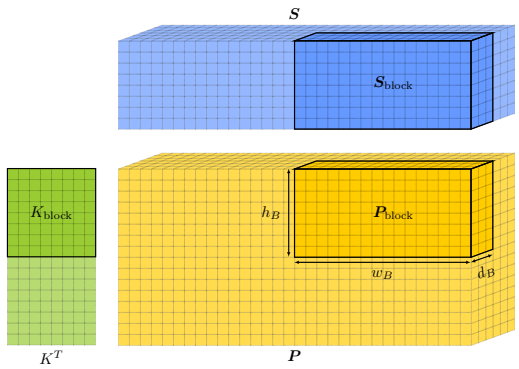

Fig. 4: A $\boldsymbol{P}_{\text{block}}$ is handled by a thread block. To compute its values, the TB must read the associated K_{block} and $\boldsymbol{S}_{\text{block}}$.

```
1  Function Im2Tensor
       input  : An image I, a kernel K
       output: R the 2D valid convolution
2      K^T ← transpose(K)
3      begin CUDA Kernel
4          while any P_block remains do
5              b_idx ← getPBlockIdx (TB_idx)
6              K_block ← readLines (K^T, b_idx)
7              S_block ← readAndBuildSubTensor (I, b_idx)
8              P_block ← matTensorMultiply (K_block, S_block)
9          end
10     end
11     begin CUDA Kernel
12         while any diagonal remains do
13             d^(k), d^(l) ← getDiagIds (Thread_idx)
14             s ← sumDiagonal (P, d^(k), d^(l))
15             storePixel (R, s)
16         end
17     end
```

Algorithm 1: Pseudo-code description of im2tensor.

reading directly rows of I when a TB needs them for a matrix-matrix product (algorithm 1, line 6).

4.2 Performance optimization

4.2.1 Tensor cores considerations

im2tensor is designed to run efficiently on matrix multiplication units, such as tensor cores, presented in section 2.3. The first part of *im2tensor* computes $K^T\boldsymbol{S}$, a matrix-tensor product, which is, in essence, a collection of matrix-matrix multiplications. In turn, each of these products can be further decomposed into smaller block-matrices products that are accelerated by tensor cores.

Since tensor cores only accept matrices of a limited choice of dimensions as inputs, our implementation must pad block-matrices to work around this constraint. A bad choice for the input dimensions may lead to significant overhead, so it is in our best interest to find a sensible choice of those tensor cores input shapes.

Let the tensor cores input dimension be $(m_{\text{tc}}, n_{\text{tc}}, k_{\text{tc}})$. For a convolution kernel $(h_K, w_K) = (5, 5)$ and tensor core input dimension of size $(32, 8, 16)$, the K^T must be padded to an height of 32. With this choice of shape, tensor cores mostly operates on zero padding.

For *im2tensor* to operate efficiently with tensor core on a diversity of images' and convolution kernels' shapes, we explore how to choose the best set of input shapes. First, we introduce the notation $\lceil m \rceil_{(n)}$, the multiple of n directly higher than m:

$$\forall n \in \mathbb{N}^*, \forall m \in \mathbb{N},\ \lceil m \rceil_{(n)} \mapsto n\lceil \frac{m}{n} \rceil \tag{10}$$

Zero-padded matrices use a hat in their name, such that:

$$\hat{w}_K = \lceil w_K \rceil_{(m_{\text{tc}})},\ \hat{h}_K = \lceil h_K \rceil_{(k_{\text{tc}})},\ \hat{w}_I = \lceil w_I \rceil_{(n_{\text{tc}})} \tag{11}$$

$$\hat{K}^T = \begin{bmatrix} K^T & 0 \\ 0 & 0 \end{bmatrix} \in \mathbb{R}^{\hat{w}_K \times \hat{h}_K} \tag{12}$$

$$\forall k \in [\![0,\ d_S - 1]\!],\ \hat{\boldsymbol{S}}_{:,:,k} = \begin{bmatrix} \boldsymbol{S}_{:,:,k} & 0 \\ 0 & 0 \end{bmatrix} \in \mathbb{R}^{\hat{h}_K \times \hat{w}_I} \tag{13}$$

Now, we have $\hat{\boldsymbol{P}} = \hat{K}^T\hat{\boldsymbol{S}}$ of shape $(\hat{w}_K, \hat{w}_I, h_I - h_K + 1)$.

4.2.2 Computational Complexity

Let us show the impact of padding on the algorithm by deriving its computational complexity. Our method uses a simple matrix multiplication algorithm, without Strassen-like methods' sophistication. Then, its complexity is:

$$\mathcal{O}\left(\hat{w}_K\hat{w}_I\hat{h}_K(h_I - h_K + 1)\right) \tag{14}$$

$$\text{i.e., } \mathcal{O}\left(\lceil w_K \rceil_{(m_{\text{tc}})}\lceil w_I \rceil_{(n_{\text{tc}})}\lceil h_K \rceil_{(k_{\text{tc}})}(h_I - h_K + 1)\right) \tag{15}$$

Three terms are directly impacted by the padding introduced by tensor cores' input shapes. In our setting, the image is significantly larger than the kernel: $w_I \gg w_K$, $w_I \gg h_K$. This assumption guide indicates that that k_{tc} and m_{tc} should prioritly be kept small.

4.2.3 Arithmetic Intensity

The arithmetic intensity of a computer program that transfers Q bytes of data to execute W operations is defined as: $\text{ai} = \frac{W}{Q}$.

This measure, central to the roofline model analysis [30], must be high enough to claim using the GPU efficiently [10]. It is usually measured at execution time through performance counters. We, however, propose an *a priori* estimation of the arithmetic intensity to guide our design choices.

We are interested in the computation of a P_{block}, of size (h_B, w_B, d_B), as shown in fig. 4. Let us see how these three dimensions should be chosen to maximize the arithmetic intensity.

As for data transfers, this operation fetches data from $\hat{K}^T$: a $(h_B, \hat{h}_K)$ sub-image; and from $\boldsymbol{S}$: a $(\hat{h}_K, w_B, d_B)$ sub-tensor. Due to duplicate rows in $\boldsymbol{S}_{\text{block}}$, this sub-tensor only requires reading a $(\hat{h}_K + d_B - 1, w_B)$ sub-image from I. Memory writes are not counted in this analysis.

Regarding operations, the matrix-tensor products for $\boldsymbol{P}_{\text{block}}$ involve $h_B \hat{h}_K w_B d_B$ multiplication additions. Then, we have:

$$\text{ai}(h_B, w_B, d_B) = \frac{h_B \hat{h}_K w_B d_B}{h_B \hat{h}_K + (\hat{h}_K + d_B - 1) w_B} \tag{16}$$

h_B	w_B	d_B	$\text{ai}(h_B, w_B, d_B)$
16	16	16	103.7
32	16	16	147.6
16	32	16	130.0
16	16	32	172.5

Table 1: Value of ai for some choices of h_B, w_B, and d_B, with $\hat{h}_K$ set to 32.

The analysis of numerical results in table 1 show that it is preferable to increase d_B as a priority, then h_B, and finally w_B. One must still consider that increasing d_B and h_B means reading more rows of I, which is usually slower than reading longer rows (by increasing w_B) due to the memory layout of the image.

In our experiments, with $(m_{\text{tc}}, n_{\text{tc}}, k_{\text{tc}}) = (8, 32, 16)$, we set (h_B, w_B, d_B) to $(8, 32, 32)$ as it provided the best results.

4.2.4 Fusing the operators

Section 4.2.3 showed how to choose adequate dimensions for $\boldsymbol{P}_{\text{block}}$ to get a sufficient arithmetic intensity. However, the number of writes in memory to store the intermediate tensor $\boldsymbol{P}$ was not considered.

To mitigate overutilization of the memory bandwidth, a technique called kernel fusion can be quite efficient [11]. Here, we propose to fuse the computation of $\boldsymbol{P}_{\text{block}}$ and the sums on the diagonals of $\boldsymbol{P}$.

We can take advantage of the fact that a $\boldsymbol{P}_{\text{block}}$ is computed by a TB to store it on the shared memory first, then sum along its diagonals, and finally, write those sums back to main memory. By doing so, the storage space requirement and main memory bandwidth usage drop from $h_B w_B d_B$ to $(h_B + w_B - 1) d_B$.

The proposed algorithm is then modified to include this $\boldsymbol{P}_{\text{block}}$ partial reduction. The sum on diagonals is now split into two passes: the first one within the shared memory for a $\boldsymbol{P}_{\text{block}}$, then recombination of the partial sums from the different $\boldsymbol{P}_{\text{blocks}}$. The pseudo-code in algorithm 2 summarizes this idea.

```
1  Function Im2TensorFused
       input  : An image I, a kernel K
       output: R the 2D valid convolution
2      K^T ← transpose(K)
3      begin CUDA Kernel
4          while any P_block remains do
5              b_idx ← getPBlockIdx (TB_idx)
6              K_block ← readLines (K^T, b_idx)
7              S_block ← readAndBuildSubTensor (I, b_idx)
8              P_block ← matTensorMultiply (K_block, S_block)
9              d_partial ← sumTensorDiagonals (P_block)
10             store (P_partial, d_partial)
11         end
12     end
13     begin CUDA Kernel
14         while any diagonal remains do
15             d^(k), d^(l) ← getDiagIds (Thread_idx)
16             s ← sumDiagonal (P_partial, d^(k), d^(l))
17             storePixel (R, s)
18         end
19     end
```

Algorithm 2: The fused im2tensor variant.

Finding an efficient data structure to hold the partial sums of a tensor is not straightforward. The perhaps simplest idea would be to store partial sums of a diagonal directly on the pixel of R it contributes to. This requires atomic sums for all thread blocks to work concurrently. In the following, this strategy is named *atomic*. On the one hand, it removes the need for an intermediate buffer. On the other hand, it makes the algorithm slower, as atomic memory accesses are serialized and do not fully utilize the available bandwidth.

For better performance, we developed a method that does not rely on atomic operations. Let us present it in a simple case. A is a (h_A, w_A) matrix on which Tr_+

should be applied. A is partitioned by the submatrices $\{M^{(i,j)}\}$ of size (h_M, w_M) as shown in fig. 5.

$$A = \begin{bmatrix} M^{(0,0)} & \dots & M^{(0,w_A/w_M)} \\ \vdots & \ddots & \vdots \\ M^{(h_A/h_M,0)} & \dots & M^{(h_A/h_M,w_A/w_M)} \end{bmatrix} \tag{17}$$

Let us explore a method for computing sums on the $M^{(i,j)}$ concurrently. This would translate into each GPU TB being affected to a $M^{(i,j)}$ block in our implementation. To do so, a 2D buffer C is used to store partial sums computed in each submatrix. Each column of C is reserved for a diagonal of A. Each $M^{(i,j)}$ writes its partial results on a row of C. In fig. 5, $M^{(0,1)}$ computes partial sums of A-diagonals $d^{(3)}$ to $d^{(9)}$. The results are written to $C_{0,3:9}$.

Reserving one row of C per B-block would waste a lot of space. We adopt two strategies to reduce the memory footprint needed for C. First, blocks on the same block-antidiagonal (i.e. $\forall n$, $\{M^{(i,j)}$, s.t. $i + j = n\}$) do not have any diagonal of A in common to sum. This way, they can safely use the same row C, as the columns they use don't overlap. In fig. 6, $M^{(1,1)}$ and $M^{(0,2)}$ use the third row of C.

Second, let's examine a block-antidiagonal's memory footprint on C: it is a segment of length $h_A - 1 + w_M \min(\frac{h_A}{h_M}, \frac{w_A}{w_M})$. Moreover, the gap in the diagonal indices treated by two consecutive block-antidiagonals is w_M. Thus, when two antidiagonals are sufficiently apart from each other, they may safely use the same row of C. With antidiagonals n and m (with $m > n$) this happens when:

$$(m-n)w_M > h_A - 1 + w_M \min\left(\frac{h_A}{h_M}, \frac{w_A}{w_M}\right) \tag{18}$$

$$m - n \geq \lceil \frac{h_A - 1}{w_M} \rceil + \min\left(\frac{h_A}{h_M}, \frac{w_A}{w_M}\right) \tag{19}$$

In fig. 6, this is shown with $M^{(0,0)}$ and $M^{(1,2)}$, from antidiagonals 0 and 3, using the same row of C.

With these two techniques, the required number of rows for C is reduced from $(\frac{h_A}{h_M}\frac{w_A}{w_M})$ to $\left(\lceil \frac{h_A - 1}{w_M} \rceil + \min(\frac{h_A}{h_M}, \frac{w_A}{w_M})\right)$.

Compared to the initial algorithm, the fused version limits the memory footprint by avoiding the creation of $\boldsymbol{P}$. It also limits the main memory bandwidth through partial summations in the $\boldsymbol{P}_{\text{block}}$.

In a setting where the image is 1024×1024, the kernel 32×32 and $w_B = h_B = 32$, the fused algorithm reduces by $\sim 15\times$ its memory bandwidth.

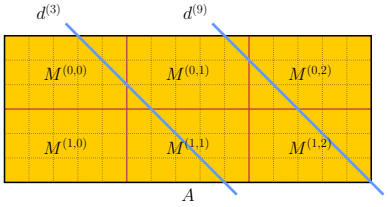

Fig. 5: The matrix A is divided into submatrices $\{M^{(i,j)}\}$. $M^{(0,1)}$ computes partial sums for $d^{(3)}$ to $d^{(9)}$.

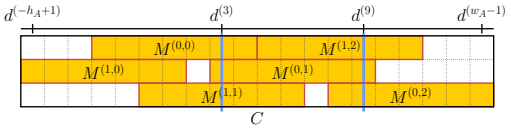

Fig. 6: Each $M^{(i,j)}$ writes its results to a segment of C. To get the full sum for $d^{(3)}$, one should add the partial sums from $M^{(0,0)}$, $M^{(0,1)}$, and $M^{(1,1)}$.

5 Results

5.1 Introduction

5.1.1 Experimental setup

This section presents the results of several experiments. They were run in two environments, as detailed in table 2. We are interested in two facets of a convolution's performance: the speed and the accuracy of its results.

	Machine #1: desktop	Machine #2: embedded (Jetson AGX Xavier)
OS	Ubuntu 16.04	Ubuntu 18.04
Linux Kernel	4.15.0	4.9.140
CUDA	11.2	10.2
NVIDIA Driver	450	JetPack 4.4
CPU	Intel i7-3820	8-core ARM 64bits
GPU	Titan V (arch. 7.0)	Xavier (arch. 7.2)
TDP	~500W	~30W

Table 2: Environments of the experiments.

For speed tests, we run the implementations on the *cameraman* image (see fig. 7), resized to (1024×1024) or (4096×4096) pixels. This setting has been chosen to mimic an industrial context where an image is to be preprocessed by a large Gaussian kernel.

We summarize the results by taking the median execution time over 20 runs. Our benchmark program measures performance with the help of *cudaEvents*. The reported timings measure the GPU kernel's execution time. CPU/GPU transfers are left aside, as we assume that they all already live in the GPU memory.

For accuracy, we rely on the median absolute percentage error (median APE, or MAPE). For a pixel p_i,

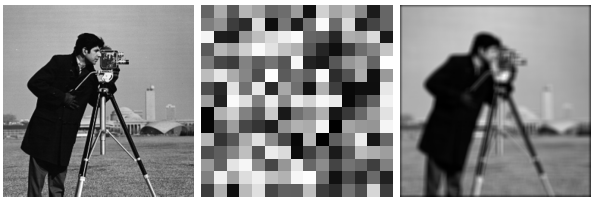

Fig. 7: Left, the original image. Middle, a 15×15 random kernel. Right, the "same" convolution.

generated by the algorithm under test and compared to r_i, the pixel from the reference implementation, the APE is defined as:

$$\mathrm{APE}(p_i) = \begin{cases} |\frac{p_i - r_i}{r_i}|, & \text{if } r_i \neq 0 \\ 0, & \text{if } r_i = 0 \end{cases} \quad (20)$$

We aggregate the results on 39 images from the *Miscellaneous* USC-SIPI dataset [29]. Kernels' coefficients are randomly chosen in the interval $[0, 1)$. We use scipy's *correlate2d* with float64 numbers to generate reference results. All images are also stored on disk in float64 precision using the FITS format.

When the precision used by an algorithm is less than fp64, we initially convert the image and kernel to the lower precision. That lower precision is subsequently used for all computations and storage. It is finally promoted to fp64 to compute its *APE*.

We also consider mixed-precision: in fp16fp32, the algorithm accesses data in fp16, does the computation in fp32, and store the final result in fp16 (which will later be promoted to fp64 for file storage). For tensor core implementations, the situation is slightly more subtle: with fp16 input, tensor cores always use fp32 internal registers for intermediate results (see fig. 8). In what we call fp16 implementation, an fp16 output is requested from the tensor core. For fp16fp32, we use fp32 results from the tensor core.

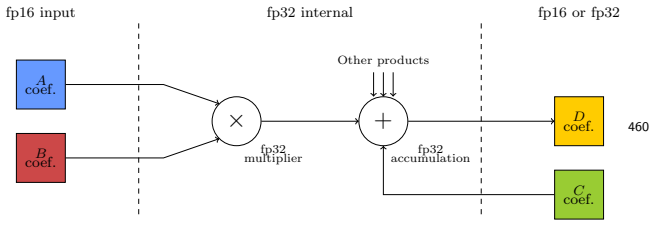

Fig. 8: Internal decomposition of a tensor core operation. Based on: `https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/`.

The exclusive mode is requested for the GPU under test, which means that no other programs will interfere with its execution. Moreover, the GPU clocks are set to fixed values to mitigate dynamic frequency scaling effects.

5.1.2 Implementations Under Test

In this benchmark, we try to cover a broad range of algorithms. However, to make the comparison fair, we restricted ourselves to general implementations, not dedicated to one kernel size. While those specialized programs may reach 2-5× speedups compared to general algorithms, they need to produce an executable for each kernel size. Thus, the compilation time increases, and the final binary is large, making it less prone to general-purpose library integration.

We compared two families of in-house implementations ("naive", *im2tensor*) with first-party NVIDIA libraries (CUFFT, CUDNN, NPP) and a third-party library (ArrayFire). Let us give a brief description of each algorithm:

im2tensor This is the algorithm explained in this article. The *+ shmem* versions use the shared memory for efficient reuse of $\boldsymbol{S}_{block}$ and K_{block}. The *+ fused* versions use the optimization explained in section 4.2.4. Finally, the *via CUBLAS* version builds the $\boldsymbol{S}$ tensor explicitly and uses CUBLAS to perform matrix multiplications between K^T and the slices of $\boldsymbol{S}$.

"Naive" This is the classical approach for convolutions on GPU: each GPU thread is assigned to computing a resulting pixel. Therefore, each thread loops over the kernel and image pixels to multiply and sum. In the *+ shmem* version, threads in the same thread block use the shared memory to reuse image pixels. Most of the code is inspired by CUDA samples [24].

CUFFT This algorithm performs convolutions in the Fourier domain. The time to do the Fourier transform of the kernel is not counted, as it could easily be precomputed and stored in a real-world application. What counts for this implementation is the Fourier transform of the image, the pointwise multiplication in the Fourier domain, and the inverse Fourier transform.

CUDNN This library, used for deep neural networks, features the *im2col* algorithm [8]. We used "cudnnConvolutionForward" with the "CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM" setting on version 8.1.1 (2021).

NPP NPP are NVIDIA Performance Primitives. They contain many utility functions for signal and image processing.

ArrayFire This general-purpose GPU-accelerated library features convolution implementations based on Fourier transform (ArrayFire Freq.) or similar to naive + shmem (ArrayFire Spatial) [33]. Used on version 3.8.0 (2021).

5.2 Performance

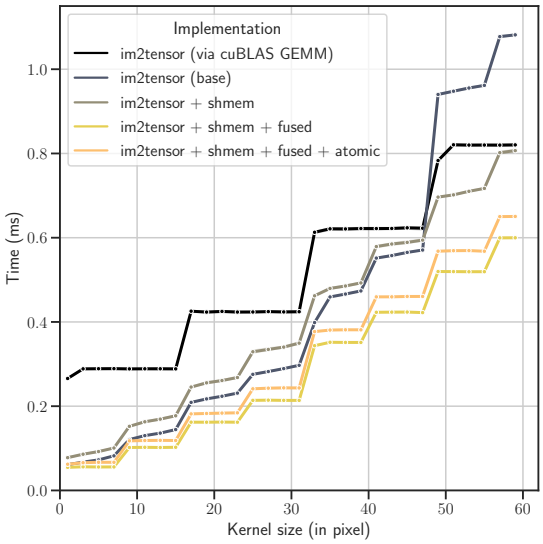

Fig. 9: Effects of im2tensor *(fp16)* optimizations on Titan V on 1024 × 1024 images.

Figure 9 shows the results for different implementations of the im2tensor algorithm. The CUBLAS version creates the whole $\boldsymbol{S}$ tensor before using the GEMM CUBLAS implementation for multiple matrix-matrix multiplications, as explained in section 5.1.2. Because the time to create $\boldsymbol{S}$ is not counted, it only serves as a reference for the other implementations. Aside from that, even if the CUBLAS library is highly optimized for GEMMs, it is still penalized, with respect to other implementations, by the large data movement that results from fetching $\boldsymbol{S}$ entirely.

The base im2tensor algorithm already achieves satisfying results. Nevertheless, we applied the optimizations discussed previously. The *shmem* version performs slightly worse, we suppose it is mainly caused by bank conflicts in the shared memory and to the L1 cache being already as efficient as using shared memory.

By adding the *fused* optimization, though, the initial algorithm is outperformed. This confirms the efficiency of reusing data as much as possible once they have been moved to the thread block. At last, the additional *atomic* optimization slows down the runtime slightly, but does not rely on any intermediate buffer.

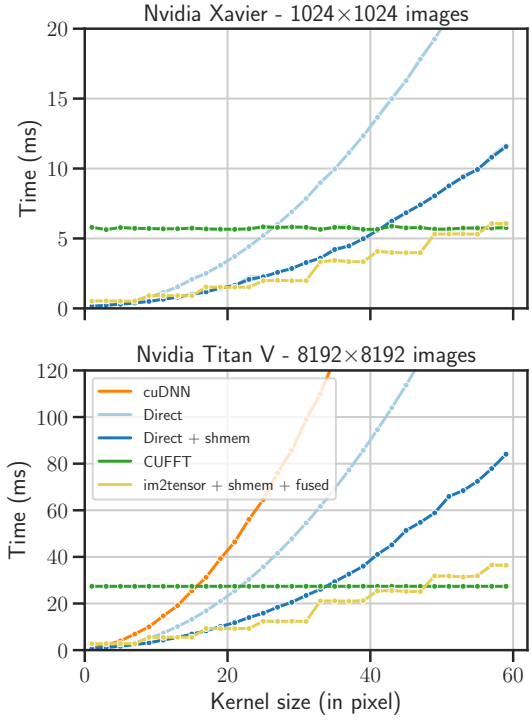

Fig. 10: Comparison of *fp16* algorithms in two contexts: (1024 × 1024) images on Jetson Xavier and (8192 × 8192) images on Titan V. Bands represent the 95% confidence interval.

In fig. 10, the *im2tensor* algorithm is compared with other *fp16* implementations. CUDNN proves to be rather slow in with a single kernel setting. Naive implementations perform well for relatively small kernels. The usage of shared memory is beneficial for larger kernels. Note that both programs use the vectorized *half2* data type to maximize compute-throughput. The Fourier implementation, CUFFT, is almost constant with respect to the kernel size. The overhead for small kernels is prohibitive but becomes less of a problem with large kernels (> 40 pixels).

Our algorithm behaves quite well for all kernel sizes. On small kernels, it is on par with the best implementations. The overhead due to padding for tensor cores does not allow it to be the fastest. As the kernel grows in size, the im2tensor execution time curve is less steep than "naive" implementations. Thus, it is the fastest for kernels between 15 and ~ 50 pixels in size.

For very large kernels, CUFFT remains the fastest. This seems sensible, as the algorithmic complexity is

asymptotically better for convolutions in the Fourier space rather than in the direct space.

Additionally, we collected several runtime metrics to understand the implementations' bottlenecks. For a kernel of size 40, the "naive" + shmem implementation is compute-limited. It uses more than 95% of the GPU's ALU throughput. That limitation is also present for CUDNN, which uses about 80%. Conversely, CUFFT is mostly bandwidth-limited: it attains 85% of the theoretical memory bandwidth. Finally, our *im2tensor* versions are mainly occupancy-limited. This restriction is caused by their large register footprints and required shared memory (for versions that use it).

Implementation	Precision	Kernel Size				
		3	15	25	35	55
"Naive"	fp16	**0.21**	3.11	8.37	16.17	41.27
"Naive" + shmem		0.31	1.64	3.71	7.43	16.48
CUFFT		5.95	5.96	6.08	5.95	**5.96**
im2tensor + shmem + fused		0.63	**1.23**	**2.70**	**4.64**	7.02
im2tensor + shmem + fused + atomic		0.82	1.50	3.04	4.86	7.24
"Naive"	fp16fp32	**0.39**	6.62	17.91	34.72	84.96
"Naive" + shmem		0.47	3.06	7.21	13.55	47.06
im2tensor + shmem + fused		1.07	**2.30**	**4.70**	**8.37**	**12.55**
im2tensor + shmem + fused + atomic		1.44	2.86	5.73	9.51	14.04
"Naive"	fp32	0.41	6.16	16.71	32.31	168.22
"Naive" + shmem		0.51	**2.47**	**6.38**	14.78	38.12
ArrayFire (Freq.)		10.79	10.79	10.78	10.79	10.79
ArrayFire (Spatial)		0.40	2.68	—	—	—
CUFFT		9.65	9.64	9.64	**9.64**	**9.64**
NPP		**0.25**	2.74	7.18	14.01	34.34
"Naive"	fp64	0.52	6.46	17.53	59.79	214.71
"Naive" + shmem		0.69	3.21	**10.82**	**14.89**	45.86
ArrayFire (Freq.)		21.51	21.51	21.51	21.52	**21.52**
ArrayFire (Spatial)		0.56	**2.84**	—	—	—
CUFFT		24.59	24.59	24.58	24.59	24.59
NPP		**0.47**	5.71	14.74	28.46	70.60

Table 3: Median execution time (in ms) on 4096×4096 images vs. size of kernel. Best time per category is highlighted.

Table 3 provides timings for various implementations and several precisions. It shows once again that im2tensor is the fastest method for most kernel sizes. Our algorithm also performs well in the mixed fp16fp32 case. Note that *im2tensor* algorithms are only implemented in fp16 and fp16fp32, due to a limition in NVIDIA's API. In fp32 and fp64, *ArrayFire (Spatial)* cannot handle large kernels. For those cases, the result is marked "—".

Implementation	Extra memory requirement (MB)
"Naive"	0
CUDNN	260
CUFFT	1,680
im2tensor	516
im2tensor + shmem + fused	98
im2tensor + shmem + fused + atomic	0

Table 4: Algorithms' required memory for a 15×15 convolution with a 4096 × 4096 image. Add 423MB to account for image, kernel, and result storage as well as CUDA runtime.

Finally, table 4 shows the VRAM used by different algorithms to perform their operations. The "naive" version does not need additional storage: its output is computed and directly stored in the result buffer. CUFFT is memory intensive due to the intermediate Fourier transforms. Also, the different im2tensor optimizations prove to have a significant impact on the space requirements. The "shmem + fused" version reduces the baseline im2tensor algorithm by more than 80% as shown in fig. 6. In contrast, the atomic variant eliminates any additional memory.

5.3 Accuracy

The previous section highlighted the difference in speed across floating-point formats. There is, however, a trade-off between speed and accuracy when it comes to float operations. Figure 11 compares some implementations' accuracy. The results are averaged over the whole USC-SIPI (Misc) database, with 95% confidence bands. The naive implementation grows from 0.02% MAPE eq. (20) to about 3% for kernels from 3 to 60. It means that for a large kernel, you can expect a 3% inaccuracy for each pixel.

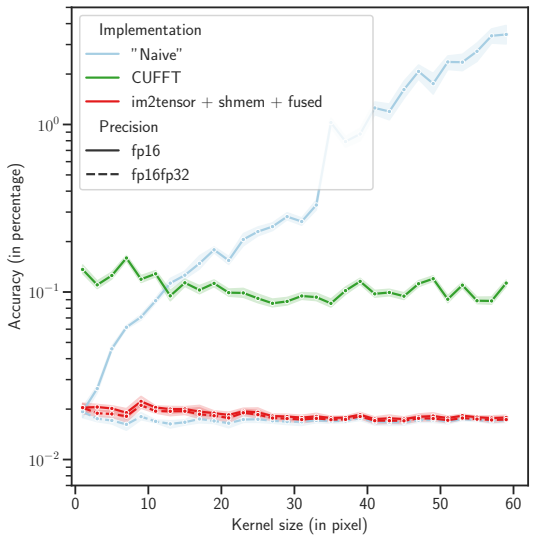

Fig. 11: MAPE accuracy of algorithms in *fp16* (solid) or *fp16fp32* (dashed). One color per algorithm. Lower is better.

This inaccuracy might be too large in some contexts [26]. Fortunately, other implementations perform better. CUFFT is almost constant at 0.1%. im2tensor algorithms in fp16 reach a constant 0.02% inaccuracy, whatever the kernel size. Finally, the performance of

the "naive" algorithm meets those of im2tensor when it is executed in fp32fp16 mixed precision.

The remarkable performance of im2tensor is explained by the use of tensor cores. As fig. 8 showed, even in fp16 precision, the intermediate results of the tensor core are computed with fp32 numbers [14, 1]. Given that the accuracy of im2tensor (fp16) is about the same as the mixed (fp16fp32) versions of "naive" and im2tensor, we can deduce that the accuracy is further limited by the storage type rather than the type used for intermediate computations.

Implementation	Precision	Kernel Size				
		3	15	25	35	55
"Naive"	fp16	**2.73e-02**	1.21e-01	2.08e-01	1.04	2.91
CUFFT		1.06e-01	1.12e-01	8.75e-02	8.37e-02	8.57e-02
im2tensor + shmem + fused		2.09e-02	**2.03e-02**	**1.83e-02**	**1.77e-02**	**1.78e-02**
"Naive"	fp16fp32	**1.76e-02**	**1.69e-02**	**1.69e-02**	**1.69e-02**	**1.70e-02**
im2tensor + shmem + fused		1.90e-02	1.97e-02	1.78e-02	1.72e-02	1.72e-02
"Naive"	fp32	**3.54e-06**	**1.48e-05**	2.42e-05	3.39e-05	5.30e-05
ArrayFire (Freq.)		2.55e-05	2.89e-05	2.51e-05	2.70e-05	2.76e-05
ArrayFire (Spacial)		3.61e-06	**1.48e-05**	—	—	—
CUFFT		1.98e-05	1.82e-05	**1.99e-05**	**1.93e-05**	**1.80e-05**
NPP		3.63e-06	**1.48e-05**	2.42e-05	3.39e-05	5.30e-05
"Naive"	fp64	**0**	**0**	**0**	**0**	**0**
ArrayFire (Freq.)		1.93e-14	3.40e-14	4.96e-14	6.59e-14	1.02e-13
ArrayFire (Spacial)		1.11e-14	3.83e-14	—	—	—
CUFFT		2.11e-14	3.38e-14	4.91e-14	6.59e-14	1.02e-13
NPP		1.11e-14	3.83e-14	6.29e-14	8.76e-14	1.37e-13

Table 5: Median accuracy (in percentage) of convolutions vs. size of kernel.

For reference, accuracy results are included for higher precision in table 5. In fp32, the trend is the same, with the "naive" growing with the kernel size. Most results stay within a $10^{-5}, 10^{-6}\%$ accuracy.

In fp64, the "naive" implementation is bit-accurate, hence the 0 precision. This is due to the reference scipy version making the same sequence of operations to compute the convolution. Other implementations are very close, about $10^{-14}\%$. Since the reference implementation also cannot be perfectly precise, it is hard to conclude for an implementation to be more accurate than another in fp64.

6 Conclusion

In this article, we have proposed a new algorithm for 2D convolutions, *im2tensor*, that exhibits matrix-multiplication operations. We implemented it on NVIDIA tensor cores, that allow modern GPUs to feature extensive FLOP/s capabilities.

We conducted an analysis of the algorithm in terms of algorithmic complexity and arithmetic intensity. This helped us make the best choice of parameters for implementing our algorithm based on matrix-tensor multiplication.

To prove the relevance of this new method, we have benchmarked several well-known implementations on GPUs. For completeness, we compared in-house implementations with first and third-party libraries. The effects of floating-point precision on the accuracy of the computation were also reviewed.

We evaluated those methods on two different setups: embedded ($\sim$ 30W) with an NVIDIA Jetson Xavier and desktop ($\sim$ 500W) with an NVIDIA Titan V. Based on our experiments, it appears that our optimized method for computing convolution via matrix-tensor multiplication with tensor cores is competitive for a large range of kernel sizes.

For small kernels ($\leq$ 20-pixel wide), it is on par with shared memory "naive" implementations and 10$\times$ faster than Fourier transforms. For large kernels ($\sim$ 50 pixels), our method is as fast as Fourier transforms and 2$\times$ faster than shared-memory "naive".

Regarding accuracy, compared to other fp16-only methods, our algorithm is 5$\times$ more precise than Fourier transforms and 100$\times$ as good as "naive" implementation for large kernels, in terms of MAPE. This gain is directly provided by tensor cores, as they use extended-precision intermediate registers.

In the future, the *im2tensor* algorithm could be extended to the newer Ampere architecture. Further optimization could be conducted to use the low-level *mma* interface to tensor cores and attain maximum performance. Non-GPU targets may also benefits from *im2tensor*, such as Google's TPU or Habana's Gaudi [**?**].

References

1. A. Abdelfattah, H. Anzt, E. G. Boman, E. Carson, T. Cojean, J. Dongarra, M. Gates, T. Grützmacher, N. J. Higham, S. Li, N. Lindquist, Y. Liu, J. Loe, P. Luszczek, P. Nayak, S. Pranesh, S. Rajamanickam, T. Ribizel, B. Smith, K. Swirydowicz, S. Thomas, S. Tomov, Y. M. Tsai, I. Yamazaki, and U. M. Yang, *A Survey of Numerical Methods Utilizing Mixed Precision Arithmetic*, arXiv:2007.06674 [cs, math], (2020), `https://arxiv.org/abs/2007.06674`.
2. K. Adámek, S. Dimoudi, M. Giles, and W. Armour, *GPU Fast Convolution via the Overlap-and-Save Method in Shared Memory*, arXiv:1910.01972 [cs], (2020), `https://arxiv.org/abs/1910.01972`.
3. A. Anderson, A. Vasudevan, C. Keane, and D. Gregg, *Low-memory GEMM-based convolution algorithms for deep neural networks*, arXiv:1709.03395 [cs], (2017), `https://arxiv.org/abs/1709.03395`.
4. B. Barabasz, A. Anderson, and D. Gregg, *Improving The Accuracy of Winograd Convolution for Deep Neural Networks*, (2018), p. 18.
5. P. M. Basso, F. F. dos Santos, and P. Rech, *Impact of Tensor Cores and Mixed Precision on the Reliability of Matrix Multiplication in GPUs*, IEEE Transactions on Nuclear Science, 67 (2020), pp. 1560–1565, `https://doi.org/10.1109/TNS.2020.2977583`.
6. S. G. Bhaskaracharya, J. Demouth, and V. Grover, *Automatic Kernel Generation for Volta Tensor Cores*,

arXiv:2006.12645 [cs], (2020), https://arxiv.org/abs/2006.12645.

7. R. Brunelli, *Template Matching Techniques in Computer Vision: Theory and Practice*, John Wiley & Sons, Apr. 2009.
8. S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, *cuDNN: Efficient Primitives for Deep Learning*, arXiv:1410.0759 [cs], (2014), https://arxiv.org/abs/1410.0759.
9. A. Dakkak, C. Li, I. Gelado, J. Xiong, and W.-m. Hwu, *Accelerating Reduction and Scan Using Tensor Core Units*, Proceedings of the ACM International Conference on Supercomputing, (2019), pp. 46–57, https://doi.org/10.1145/3330345.3331057, https://arxiv.org/abs/1811.09736.
10. N. Ding and S. Williams, *An Instruction Roofline Model for GPUs*, in 2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), Nov. 2019, pp. 7–18, https://doi.org/10.1109/PMBS49563.2019.00007.
11. J. Filipovic and S. Benkner, *OpenCL Kernel Fusion for GPU, Xeon Phi and CPU*, in 2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Oct. 2015, pp. 98–105, https://doi.org/10.1109/SBAC-PAD.2015.29.
12. J. S. Firoz, A. Li, J. Li, and K. Barker, *On the Feasibility of Using Reduced-Precision Tensor Core Operations for Graph Analytics*, in 2020 IEEE High Performance Extreme Computing Conference (HPEC), Sept. 2020, pp. 1–7, https://doi.org/10.1109/HPEC43674.2020.9286152.
13. A. González, *Trends in Processor Architecture*, in Harnessing Performance Variability in Embedded and High-performance Many/Multi-core Platforms: A Cross-layer Approach, W. Fornaciari and D. Soudris, eds., Springer International Publishing, Cham, 2019, pp. 23–42, https://doi.org/10.1007/978-3-319-91962-12.
14. A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, *Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers*, in SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, Nov. 2018, pp. 603–613, https://doi.org/10.1109/SC.2018.00050.
15. M. Khairy, A. G. Wassal, and M. Zahran, *A survey of architectural approaches for improving GPGPU performance, programmability and heterogeneity*, Journal of Parallel and Distributed Computing, 127 (2019), pp. 65–88, https://doi.org/10.1016/j.jpdc.2018.11.012.
16. D. B. Kirk and W. H. Wen-Mei, *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan kaufmann, 2016.
17. A. Lavin and S. Gray, *Fast Algorithms for Convolutional Neural Networks*, in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 4013–4021.
18. D. Mukunoki, K. Ozaki, T. Ogita, and T. Imamura, *DGEMM Using Tensor Cores, and Its Accurate and Reproducible Versions*, in High Performance Computing, P. Sadayappan, B. L. Chamberlain, G. Juckeland, and H. Ltaief, eds., vol. 12151, Springer International Publishing, Cham, 2020, pp. 230–248, https://doi.org/10.1007/978-3-030-50743-512.
19. C. A. Navarro, R. Carrasco, R. J. Barrientos, J. A. Riquelme, and R. Vega, *GPU Tensor Cores for fast Arithmetic Reductions*, arXiv:2001.05585 [cs], (2020), https://arxiv.org/abs/2001.05585.
20. M. Nourazar and B. Goossens, *Accelerating iterative CT reconstruction algorithms using Tensor Cores*, Journal of Real-Time Image Processing, (2021), https://doi.org/10.1007/s11554-020-01069-5.
21. NVIDIA, *V100 GPU Architecture: The world's most advanced datacenter GPU*, tech. report, Tech. Rep., NVIDIA, 2017.
22. NVIDIA, *NVIDIA Turing GPU Architecture: Graphics Reinvented*, tech. report, Tech. Rep., NVIDIA, 2018.
23. NVIDIA, *NVIDIA A100 Tensor Core GPU Architecture: Unprecedented Acceleration at Every Scale*, tech. report, Tech. Rep., NVIDIA, 2020.
24. V. Podlozhnyuk, *CUDA Samples Documentation: convolutionSeparable*, tech. report, Tech. Rep., NVIDIA, 2007.
25. J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Addison-Wesley Professional, 2010.
26. M. Seznec, N. Gac, A. Ferrari, and F. Orieux, *A Study on Convolution using Half-Precision Floating-Point Numbers on GPU for Radio Astronomy Deconvolution*, in 2018 IEEE International Workshop on Signal Processing Systems (SiPS), Cape Town, Oct. 2018, IEEE, pp. 170–175, https://doi.org/10.1109/SiPS.2018.8598342.
27. S. Sioutas, S. Stuijk, T. Basten, L. Somers, and H. Corporaal, *Programming tensor cores from an image processing DSL*, in Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems, SCOPES '20, New York, NY, USA, May 2020, Association for Computing Machinery, pp. 36–41, https://doi.org/10.1145/3378678.3391880.
28. S. W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*, California Technical Publishing, 1997.
29. A. G. Weber, *The USC-SIPI image database version 5*, USC-SIPI Report, 315 (1997).
30. S. Williams, A. Waterman, and D. Patterson, *Roofline: An insightful visual performance model for multicore architectures*, Communications of the ACM, 52 (2009), pp. 65–76, https://doi.org/10.1145/1498765.1498785.
31. H. Winnemöller, J. E. Kyprianidis, and S. C. Olsen, *XDoG: An eXtended difference-of-Gaussians compendium including advanced image stylization*, Computers & Graphics, 36 (2012), pp. 740–753, https://doi.org/10.1016/j.cag.2012.03.004.
32. S. Winograd, *Arithmetic Complexity of Computations*, SIAM, Jan. 1980.
33. P. Yalamanchili, U. Arshad, Z. Mohammed, P. Garigipati, P. Entschev, B. Kloppenborg, J. Malcolm, and J. Melonakos, *ArrayFire - A High Performance Software Library for Parallel Computing with an Easy-to-Use API*, AccelerEyes, Atlanta, 2015.