Efficient Tensor Cores support in TVM for Low-Latency Deep learning

Wei Sun, Savvas Sioutas, Sander Stuijk, Andrew Nelson, Henk Corporaal
Eindhoven University of Technology
{w.sun, s.sioutas, s.stuijk, a.t.nelson, h.corporaal}@tue.nl

***Abstract*— Deep learning algorithms are gaining popularity in autonomous systems. These systems typically have stringent latency constraints that are challenging to meet given the high computational demands of these algorithms. Nvidia introduced Tensor Cores (TCs) to speed up some of the most commonly used operations in deep learning algorithms. Compilers (e.g., TVM) and libraries (e.g., cuDNN) focus on the efficient usage of TCs when performing batch processing. Latency sensitive applications can however not exploit large batch processing. This paper presents an extension to the TVM compiler that generates low latency TCs implementations, particularly for batch size 1. Experimental results show that our solution reduces the latency on average by 14% compared to the cuDNN library on a Desktop RTX2070 GPU, and by 49% on an Embedded Jetson Xavier GPU.**

I. Introduction

Deep learning algorithms [14], especially Convolutional Neural Networks (CNNs), have demonstrated a great ability to solve challenging computer vision tasks like image classification, object detection, and semantic segmentation.

Modern autonomous systems such as autonomous driving vehicles, Cyber-Physical Systems, and unmanned aerial vehicles have low latency requirements, which means that CNNs have to process the input image and produce the outputs within the shortest possible response time. To this end, instead of processing larger batch sizes for high data parallelism to get high throughput, the batch size is typically small for these latency-sensitive autonomous applications.

Nvidia introduced Tensor Cores (TCs) to meet the increasing computation demands of modern CNNs [17]. Programming convolution layers on a TCs is nontrivial. To improve the programmability of TCs and make them more accessible to users without enough CUDA expertise, some effort has been invested to enable automatic code generation for TCs. [22] extends the Halide DSL [21] to support TCs for GEMM applications. Recently, the TVM [5] open-source compiler community [19] added support for convolution on TCs. However, the current TVM solution can only support input batch sizes with multiples of eight, which is not suitable for low-latency applications where the input batch size is typically one. This paper presents an extension of the TVM compiler to automatically generate low-latency convolution code for TCs. Contributions of this work are:

- Implementation description of an extension based on the TVM open-source compiler stack to automatically generate low-latency convolution CUDA code for small input batch sizes.
- Performance analysis of our solution in comparison to the Nvidia cuDNN library when targeting TCs.
- Analysis and discussion of the limitations of both our solution and cuDNN for small and large batch sizes.

This work is funded by the NWO Perspectief program ZERO, project P3

II. Background

A convolution layer can be expressed as $Output = Conv(Input, Weight)$. $Input$ stands for the input images or intermediate feature map from the previous layer. $Weight$ stands for the weights of convolutional filters learned from the training phase. The typical data layout of $Input$ and $Weight$ are (N, C, H, W) and $(Cout, C, Kh, Kw)$ respectively, where N stands for batch size, $C/Cout$ stands for the number of $Input/Output$ channels, H/W is height/width of the input and Kh/Kw is the height/width of the convolutional filters. This data layout is supported by all mainstream deep learning frameworks including Pytorch [20] and TensorFlow [1]. Although there are different data layout settings which can affect the run-time performance of CNNs [15, 16], this work focuses on this mainstream data layout.

The convolution layer is the most important and expensive layer in a CNN. How to implement convolution layers efficiently has been extensively investigated over the past years [6, 13, 18]. **Direct convolution** implements convolution in a straightforward way which consists of 7 nested loops. Alternatively, **Im2col convolution** first lowers the convolution problem into the GEMM problem, which then can be solved using highly optimized linear algebra software libraries (e.g. cuBLAS) or a GEMM hardware accelerator (e.g. Tensor Cores). Explicit im2col convolution will introduce extra memory footprint for the lowered matrices and additional memory transfers between GPU global memory and GPU streaming multiprocessors (SMs) shared memory. In this paper, we will address this extra memory footprint by introducing **implicit im2col**, which avoids the explicit construction of the im2col matrices in memory.

III. Related Work

CuDNN provides highly optimized GPU kernel functions for TCs developed by experts [12]. However, cuDNN is closed-source, blocking researchers to study the optimization techniques exploited in cuDNN functions. Furthermore, a black-box library prevents layer fusion and joint optimizations which provide significant performance gains [2].

Recently, automatic code generation for Tensor Cores has been an active research topic which aims to provide better programmability and transparency while keeping comparable

978-3-9819263-5-4/DATE21/©2021 EDAA

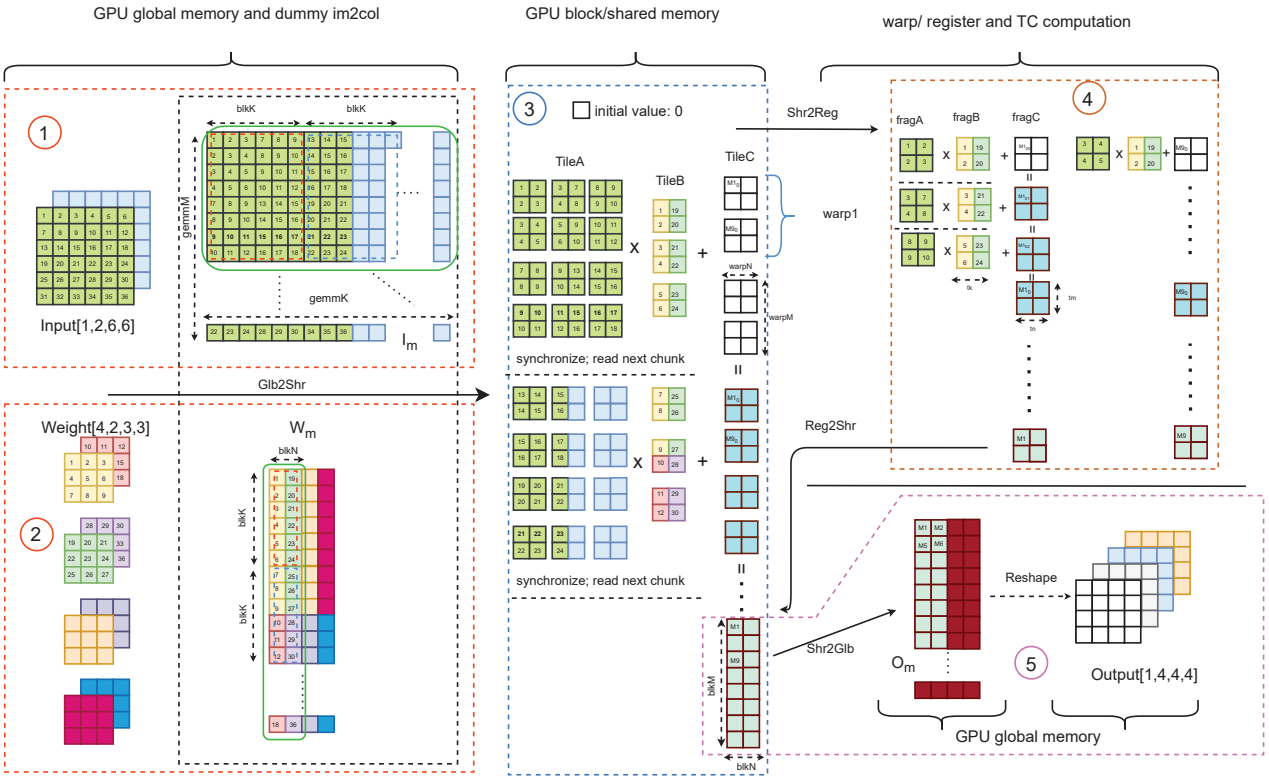

Fig. 1. Computation schedule of convolution for Tensor Cores. ① and ② show $Input$ tensor, $Weight$ tensor and corresponding implicit im2col transformations (I_m and W_m). ③ and ④ show the computation schedule at block-level and warp-level respectively. ⑤ is result construction step.

performance to black-box libraries. [22] extended the Halide Domain Specific Language (DSL) to generate PTX codes on Tensor Cores which can achieve performance reasonably close to manually tuned implementations provided by cuBLAS. [3] proposes a polyhedral approach based on the Diesel [8] DSL compiler framework to generate efficient CUDA kernels for matrix multiplication by using inline assembly instructions for Tensor Cores. Inspired by Halide and TVM, [9] proposes the Fireiron scheduling language for GPUs as well as Tensor Cores, which is claimed to outperform expert-written advanced matrix multiplication.

The above efforts have demonstrated that compilers are able to match the performance of vendor libraries developed by human experts on matrix multiplication applications. This paper aims at the efficient use of TCs for CNN convolution layers with small batch sizes. TVM supports multiple DL frameworks (like Tensorflow [1] and Pytorch [20]) generating code for multiple HW target platforms, including GPUs. However, TVM only uses TCs for certain large batch sizes [19] which cannot be exploited for low-latency applications.

IV. Exploit Tensor Cores for Low-Latency Inference

Implicit im2col requires calculating GEMM indices at run-time to access the matrices without restructuring. In comparison to explicit im2col, we trade extra computation for a reduction in memory accesses and storage. This section explains the computation schedule, memory allocation, and data movement for implicit im2col convolution with a simple convolution example as shown in Figure 1, where Input and Weight tensors are shown in box ① and ②. The data in $Input$ and $Weight$ tensors will be moved directly from GPU global memory to GPU shared memory as demonstrated in box ③ without explicitly constructing im2col matrices I_m and W_m. The size of $TileA$ and $TileB$ vary across different shapes of $Input$ and $Weight$ tensors. For visualization, we assume the size of $TileA$ and $TileB$ are (8,4) and (4,2). The matrix multiplication in box ③ is tiled into several sub-matrix multiplications which can be handled by Tensor Cores through a set of WMMA CUDA primitives [17] by a GPU thread warp as demonstrated in ④. For simplification, we assume the shape of $fragA$, $fragB$ and $fragC$ are (2,2).

The partial matrix multiplications computed by Tensor Cores are first stored in registers before being transferred to shared memory to construct the result $TileC$ in box ③. Finally, the $Output$ tensors are constructed in GPU global memory as shown in box ⑤.

To achieve a good performance, it is always crucial to optimize the memory allocation and data movement across the GPU memory hierarchy from global memory to registers. The computation schedule demonstrated in Figure 1 contains two categories of data communication. 1) Data movement

between Global memory and Shared memory (Glb2Shr and Shr2Glb): data in $Input$ and $Weight$ are moved to shared memory in GPU SM. In this step, vectorization is applied to achieve high memory bandwidth. A vectorized load/store instruction can perform n load/store operations in a single instruction, where n is the vectorization factor. 2) Data movements between Shared memory and registers (Shr2Reg and Reg2Shr), the keypoint to speed-up GPU shared memory accesses is to avoid shared memory bank conflicts. This can be achieved by applying a certain offset when allocating shared memory in GPU blocks at the cost of extra wasted memory space. For instance, after applying offset (m), the shape of shared memory for TileA will increase from (8, 4) to (8, 4+m).

The computation schedule, memory allocation, and vectorization together introduce a set of hyper-parameters which need to be tuned for different input sizes. We expose these hyper parameters to the auto-tuner [4] of the TVM compiler to perform design space exploration. Note that this will cause extra compilation time, but it will not negatively affect the compiled program during run-time.

V. Experiments

This section presents the run-time performance of our method on convolution layers in ResNet18 [10] and SqueezeNet [11] on an Embedded Jetson Xavier GPU and a Desktop RTX2070 GPU. In total we benchmark 33 distinctive convolution layers which cover most of real-life convolution settings [7]. Note that this work only discusses the per-layer performance with the standard NCHW data layout.

We disabled dynamic frequency switching on the Jetson Xavier board to get stable results. We use the TVM auto-tuning auxiliary tool [4] for finding the optimal tunable parameters among the search space when evaluating our work and the TVM baseline solution. The reported latencies of the TVM-based solutions are averaged over 1000 measurements using the TVM estimator function. When evaluating cuDNN[1], we use the **cudnnFindConvolutionForwardAlgorithm** cuDNN built-in API to try all available algorithms and find the best one with respect to latency. cuDNN also supports half precision convolution without using TCs, we filtered out non-tensor-core results by setting **mathType** to **1**. The selected cuDNN implementation is profiled by the same TVM estimator to get the latency over 1000 measurements.

We first present the latency achieved by our solution and cuDNN of the 33 convolution layers with batch size equal to one on both experimental platforms. The existing TVM solution can only support large batch sizes which are multiples of eight, so we are not able to add this baseline for batch size is equal to one experiment. Secondly, we compare our method with the existing TVM solution and the cuDNN library on two representative convolution layers with larger batch sizes to discuss the limitations of our method.

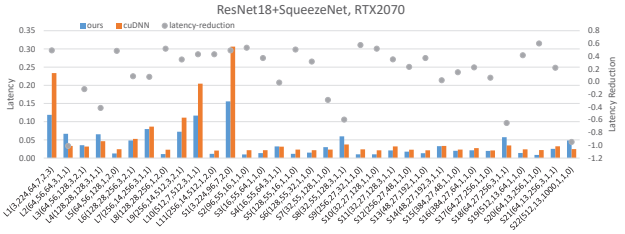

Fig. 2. Latency (ms) of convolution layers on an Embedded Jetson Xavier GPU. Format of convolution layer names: $Lx(Sx)$ $(C, H/W, Cout, Kh/Kw, Sh/Sw, Ph, Pw)$, where Lx stands for ResNet18 layer x and Sx stands for SqueezeNet layer x.

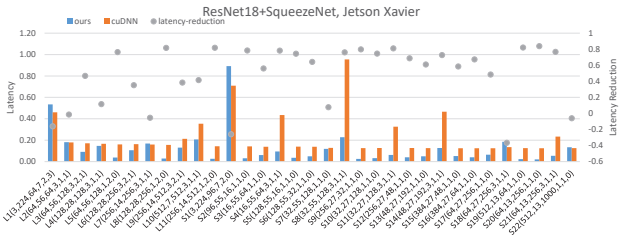

Fig. 3. Latency (ms) of convolution layers on an Embedded Jetson Xavier GPU

A. Compare to cuDNN under batch size = 1

Figure 2 and 3 show the averaged latency of our work compared to the cuDNN library on a Desktop RTX2070 GPU and an Embedded Jetson Xavier GPU. On the RTX2070 GPU, our method reduces the latency on average by **14%**, with the best case S20 (**60%**) and the worst case L2 (**-101%**), where minus means our method has a higher latency. The extreme outliers can be explained by the fact that cuDNN has perfect pre-configured kernels for these specific workloads (cuDNN performs better), or does not have a suitable function and relies on zero padding (cuDNN performs worse).

On the Embedded Jetson Xavier GPU, the best case is L8 (**82%**) and the worst case is S18 (**-37%**). Our method reduces the latency on average by **49%** which is 3.5x better than the latency reduction our method achieved on the Desktop GPU. One reason is that the cuDNN implementations rely on the usage of large extra global memory as indicated by the cuDNN built-in profiling function **cudnnConvolutionFwdAlgoPerf_t**. The RTX2070 GPU equips dedicated global memory GDDR6, but the Jetson GPU shares a unified memory LPDDR4x with the CPU. The bandwidth of dedicated GDDR6 (448GB/s) is higher than that of the Jetson Xavier LPDDR4x (137GB/s) which results in more memory access penalty when running on a Jetson Xavier. Since our solution requires fewer memory accesses, it is less penalized by this effect.

B. Study large batch sizes under an Embedded Jetson GPU

This section presents additional experiments on large batches on an Embedded Jetson GPU platform. We present results for two representative layers to demonstrate the

[1]cuDNN v7

limitations of our method. Figure 4 shows the performance of our method, cuDNN and the TVM-baseline (TVM existing solution) on convolution layer L5 with different batch sizes. L5 is a convolution layer with a 1x1 kernel (i.e. Kh = Kw = 1) where the TVM-baseline direct convolution solution shows better performance under larger batch sizes. For small batch sizes, our method is a better choice. Figure 5 compares these three methods on convolution layer L10 with a 3x3 kernel. Unlike the results of layer L5, the TVM baseline solution can not achieve a satisfactory performance. Our method is still better than cuDNN for small batch sizes, although cuDNN outperforms our method for batch size of 4 and larger.

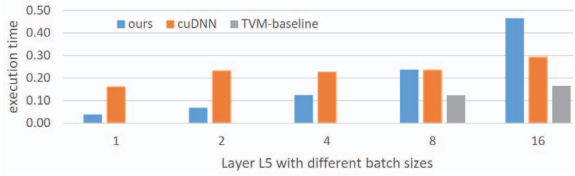

Fig. 4. Average execution time (ms) of L5 under different batch sizes on the Desktop RTX2070 GPU

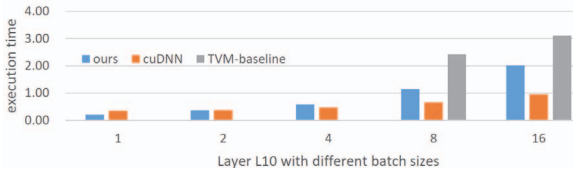

Fig. 5. Average execution time (ms) of L10 under different batch sizes on the Embedded Jetson Xavier GPU

VI. Conclusion and Future work

This paper introduces a code generation method for convolutional layers targeting TCs with NCHW standard data layout. The experiments show that our solution can reduce the latency of convolutional layers compared to the Nvidia cuDNN library, especially for a batch size of one. Our technique achieves average performance increases of 14% and 49% for desktop and embedded targets, respectively. We conclude that our work is therefore a complementary solution to the state-of-the-art cuDNN library for latency-sensitive applications.

We acknowledge that the current solution is not able to match the expertly hand-written libraries in many cases, and there are still some possibilities for improving our method further. Therefore, we propose some future work directions: 1) introduce lower-level optimizations for the TVM compiler. Our method relies on the TVM code generator to generate CUDA code, although CUDA code has better readability/-portability than PTX code or Shader ASSembly (SASS) code, some lower level optimizations cannot be applied at the CUDA language level. One option would be to implement micro-kernels based on PTX or SASS as external function calls to the generated CUDA code and integrate the micro-kernels with compiler passes. 2) investigate different data layouts. This work only discussed the standard NCHW data layout, cuDNN convolutions on Tensor Cores can perform better under NHWC layout [12]. It would be interesting exploring the efficient convolution code generation strategy for small batch sizes under NHWC data layout.

References

[1] M. Abadi et al. “TensorFlow: A System for Large-Scale Machine Learning”. In: OSDI’16. Savannah, GA, USA: USENIX Association, 2016, pp. 265–283.

[2] M. Alwani et al. “Fused-layer CNN accelerators”. In: *Annual IEEE/ACM International Symposium on Microarchitecture*. 2016, pp. 1–12.

[3] Somashekaracharya G. Bhaskaracharya, Julien Demouth, and Vinod Grover. *Automatic Kernel Generation for Volta Tensor Cores*. 2020.

[4] Tianqi Chen et al. “Learning to Optimize Tensor Programs”. In: NIPS’18. 2018.

[5] Tianqi Chen et al. “TVM: An Automated End-to-End Optimizing Compiler for Deep Learning”. In: Carlsbad, CA, USA: USENIX Association, 2018.

[6] Minsik Cho and Daniel Brand. “MEC: Memory-Efficient Convolution for Deep Neural Network”. In: ICML’17. Sydney, NSW, Australia, 2017.

[7] Marat Dukhan. “The Indirect Convolution Algorithm”. In: (2019). arXiv: 1907.02129.

[8] Venmugil Elango et al. “Diesel: DSL for Linear Algebra and Neural Net Computations on GPUs”. In: MAPL. 2018.

[9] Bastian Hagedorn et al. *Fireiron: A Scheduling Language for High-Performance Linear Algebra on GPUs*. 2020.

[10] K. He et al. “Deep Residual Learning for Image Recognition”. In: *2016 IEEE CVPR*.

[11] Forrest N Iandola et al. “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size”. In: *arXiv preprint arXiv:1602.07360* (2016).

[12] M. Jordà, P. Valero-Lara, and A. J. Peña. “Performance Evaluation of cuDNN Convolution Algorithms on NVIDIA Volta GPUs”. In: *IEEE Access* (2019).

[13] A. Lavin and S. Gray. “Fast Algorithms for Convolutional Neural Networks”. In: *2016 IEEE CVPR*.

[14] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep Learning”. In: *Nature* (2015).

[15] Mingzhen Li et al. “The Deep Learning Compiler: A Comprehensive Survey”. In: (2020). arXiv: 2002.03794v3.

[16] Yizhi Liu et al. “Optimizing CNN Model Inference on CPUs”. In: USENIX ATC ’19. Renton, WA, USA.

[17] S. Markidis et al. “NVIDIA Tensor Core Programmability, Performance Precision”. In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops*.

[18] Michael Mathieu, Mikael Henaff, and Yann LeCun. “Fast training of convolutional networks through ffts”. In: *arXiv preprint arXiv:1312.5851* (2013).

[19] *Optimization of CNNs on Tensor Core*. URL: https://discuss.tvm.ai/t/rfc-tensor-core-optimization-of-cnns-on-tensor-core/6004.

[20] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019.

[21] Jonathan Ragan-Kelley et al. “Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines”. In: PLDI ’13. Seattle, Washington, USA, 2013.

[22] Savvas Sioutas et al. “Programming Tensor Cores from an Image Processing DSL”. In: SCOPES ’20. Association for Computing Machinery, 2020.