Accelerating Reduction Using Tensor Core Units

Extended Abstract

Abdul Dakkak
University of Illinois Urbana-Champaign
Urbana, Illinois
dakkak@illinois.edu

Cheng Li
University of Illinois Urbana-Champaign
Urbana, Illinois
cli99@illinois.edu

Jinjun Xiong
IBM T. J. Watson Research Center
Yorktown Heights, New York
jinjun@us.ibm.com

Wen-mei Hwu
University of Illinois Urbana-Champaign
Urbana, Illinois
w-hwu@illinois.edu

Abstract

Driven by deep learning, there has been a surge of specialized processors for matrix multiplication, referred to as Tensor Core Units (TCUs). These TCUs come under the guise of different marketing terms and are capable of performing matrix multiplications on small matrices (usually 4x4 or 16x16) to accelerate the convolutional and recurrent neural networks in deep learning workloads. Although TCUs are prevalent and promise increase in performance and/or energy efficiency, they suffer from over specialization — with only general matrix-matrix multiplication (GEMM) being supported. This limits their applicability to general algorithms and makes them confined to narrowly specialized libraries and application domains. In this work, we leverage NVIDIA's TCU to express reduction in terms of matrix multiplication and show the benefits — in terms of program simplicity, efficiency, and performance compared to start-of-the-art reduction methods on the GPU. Although this work targets GPUs, the motivation, methods, and observations are applicable to a wide number of TCU implementations and microarchitectures

Deep learning's reliance on matrix-multiplication for compute has driven both research and industry to develop matrix-multiplication accelerator hardware — collectively called Tensor Core Units (TCUs) in this paper. TCUs come under the guise of different marketing terms, be it NVIDIA's Tensor Cores [1], Google's Tensor Processing Unit [2], Intel KNL's AVX extensions [3], Apple A11's Neural Engine [4], or ARM's Machine Learning Processor [5]. TCUs are designed to accelerate Multilayer Perceptrons (MLP), Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), or Deep Neural Network (DNN) in general TCUs vary in implementation [6, 7, 1, 8, 3, 9, 10, 11, 12, 13, 14, 15], and are prevalent [16, 17, 18, 19, 2, 20, 21, 22] in edge devices, mobile, and the cloud.

Although TCUs are prevalent and promise increase in performance and/or energy efficiency, they suffer from over specialization — with only matrix-multiplication operations being supported. This limits their applicability to general algorithms and makes them confined to narrowly specialized libraries and application domains. Take NVIDIA's Tensor Cores, for example. For matrix multiplication, the NVIDIA Volta architecture achieves a 8× throughput increase — with each Streaming Multiprocessor (SM) capable of performing 1024 half precision operations per cycle using the TCUs compared to 128 half precision operations per cycle without the TCUs. This is enabled by the fact that NVIDIA's Volta GPUs dedicate a large portion of the SM processing unit (or subcore) chip area to TCUs, shown in Figure 1. Currently algorithms other than general matrix-matrix multiplication (GEMM) do not utilize the TCUs — resulting in an idle TCU and low chip utilization for these algorithms.

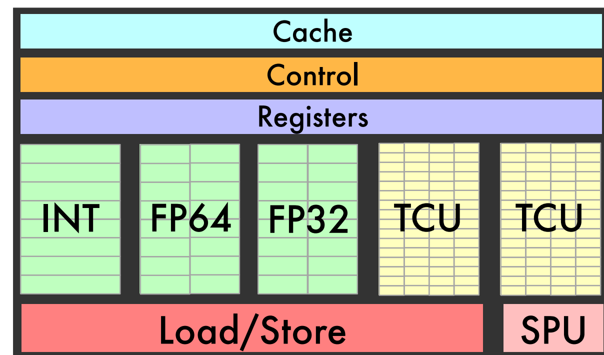

Figure 1. Each processing block (subcore) in the NVIDIA Tesla V100 PCI-E architecture contains 2 TCUs. In total, 640 TCUs are available — achieving a theoretical peek of 113 TFLOPS.

The objective of the paper is to expand the class of algorithms that can execute on TCUs — enabling the TCU to be used for non-GEMM kernels. We choose reduction, since a large body of work [23, 24, 25, 26, 27, 28, 29, 30] has shown it is a key primitives of data parallel implementations of radix sort, quicksort, string comparison, lexical analysis, stream compaction, polynomial evaluation, solving recurrence equations, and histograms. We formulate a simple mapping between reduction and TCUs. Then we develop a library of warp-, block-, and grid-level primitives for reduction

that utilize the NVIDIA TCUs, and present how they can be used in concert to achieve near-ideal performance. Although we target GPUs, the motivation, methods, and observations are applicable to a wide number of TCU implementations.

Tensor Cores have been only used to accelerate GEMM operations, most prominently through NVIDIA's CUDA libraries — cuBLAS [31], cuDNN [32] and CUTLASS [33]. NVIDIA also provides a CUDA C++ Warp Matrix Multiply and Accumulate (WMMA) [4] API to program the Tensor Cores directly. The current WMMA API provides warp-level matrix operations for matrix load (`load_matrix_sync`), matrix store (`store_matrix_sync`), and matrix multiply and accumulate (`mma_sync`). These APIs operate on a special thread-local data type (fragment), which holds a matrix tile in thread-local registers. A helper function to fill a matrix fragment with a scalar constant (`fill_fragment`) is provided as well.

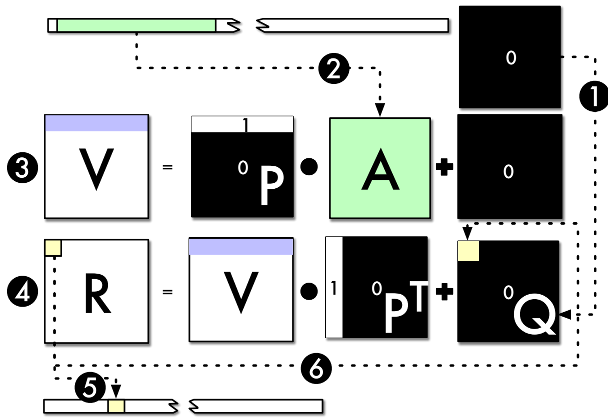

Figure 2. The work-inefficient $Reduction_{256N}$ algorithm ❶ initializes the Q matrix with all zeros and ❷ loads the 256 input elements into a matrix A in column major order. ❸ A dot product $V = P.A + \underline{\mathbf{0}}$ where the P matrix has the first row as ones and the rest of the values are zeros is performed to reduce each row into a scalar. ❹ the dot product $R = V.P^T + Q$ reduces the first row into a scalar. ❺ If the segmented reduction size is equal to the matrix size (i.e. $N = 1$) or for the last iteration, then the first element of the R matrix is stored in the output array, otherwise ❻ the first element of R matrix is used as the first element of the Q matrix and the procedure is iterated starting from step ❷.

Intuitively, reduction can be represented as a special case of a matrix multiplication, since

$$Reduction([a_1, a_2, \ldots, a_n]) =$$

$$\begin{pmatrix} 1 & 1 & \cdots & 1 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix} \cdot \begin{pmatrix} a_1 & a_2 & \ldots & a_n \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix}^T = \begin{pmatrix} \sum_{i=1}^{n} a_i & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix}$$

The challenge is to map generic input sizes onto the fixed tensor shapes supported by the TCU. Other configurations can be used in a similar manner to perform segmented reduction for multiples of 8 or 32. We use $Reduction_K$ to represent a K regular segmented reduction — partial reductions of the input uniformly partitioned into K element subsets. We will also use P as the matrix which has the first row set to 1 and the rest to 0 — i.e. $(P)_{r,c} = \begin{cases} 1 & \text{if } r = 0 \\ 0 & \text{if } r \neq 0 \end{cases}$, and the notation $\underline{\mathbf{X}}$ to denote a matrix where all the elements are the constant value X.

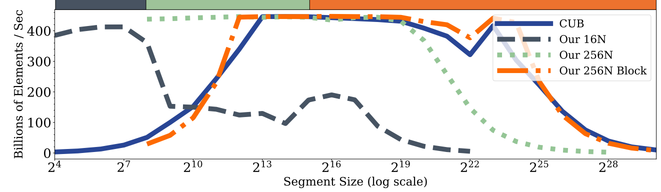

Figure 3. We evaluate the segmented reduction for the algorithms presented on different segment sizes (between 16 and 2^{30}) for a fixed 2^{30} element list. Through a combination of the algorithms presented, for the range between 16 and 2^{24} we are able to achieve throughput within 90% and 98% of ideal throughput (the theoretical peak is 450 billion half precision elements per second). The bar on top of the figure shows the best performing algorithm for each range of segment sizes.

Figure 2 shows a naïve implementation of 256N segmented reduction by first reducing each row of the 16x16 matrix in each iteration and use the row of reduced values as an accumulator. In the final iteration, the final row is reduced into a scalar. We develop a library of warp-, block-, and grid-level primitives which can be auto-tuned for difference microarchitectures and will plan on releasing in time of conference.

We evaluate our TCU reduction algorithm against the state-of-the-art implementation from CUB [34] on different segment sizes for a fixed 2^{30} element list — shown in Figure 3. Through a combination of the algorithms presented, we are able to achieve within 90% and 98% of ideal throughput (the theoretical peak is 450 billion half precision elements per second), and is orders of magnitude faster for small segment sizes — common in machine learning and scientific applications. Our algorithm achieves this while decreasing the power consumption by up to 22% (average power within the execution phase of the kernel reported by NVVP).

Collectives are never performed in isolation, instead they are used to summarize and exchange data that's produced by a much larger kernel. These larger kernels typically make extensive use of the integer and floating point ALUs. By performing the collective operations on TCUs, we alleviate the pressure on general purpose floating point ALUs — thus decreasing stalls because of hardware resource contention. Future work would leverage the techniques described to examine the impact of using TCU collectives on large applications

and see what else can be mapped to utilize the TCUs. We have identified some candidate primitives that can be mapped: such as transcendental and special functions (such as Tanh and Log), since the NVIDIA special function units have been observed to be the bottleneck in HPC applications. As well as certain DNN layers which are known to be bottlenecks. We plan on exposing the math functions and DNN layer TCU implementations using a libm and cuDNN interface respectively.

References

[1] [n. d.] NVIDIA Tensor Cores. https://www.nvidia.com/en-us/data-center/tensorcore. Accessed: 2018-7-24. (). https://www.nvidia.com/en-us/data-center/tensorcore.

[2] [n. d.] Google Cloud TPU. https://cloud.google.com/tpu. Accessed: 2018-7-24. (). https://cloud.google.com/tpu.

[3] Andres Rodriguez, Eden Segal, Etay Meiri, Evarist Fomenko, Young Jim Kim, Haihao Shen, and Barukh Ziv. 2018. Lower Numerical Precision Deep Learning Inference and Training. Tech. rep. Intel, (Jan. 2018).

[4] [n. d.] Apple A11 Bionic. https://www.apple.com/iphone-x. Accessed: 2018-7-24. ().

[5] [n. d.] Arm Machine Learning Processor. https://developer.arm.com/products/processors/machine-learning/arm-ml-processor. Accessed: 2018-7-24. (). https://developer.arm.com/products/processors/machine-learning/arm-ml-processor.

[6] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark A Horowitz. 2013. Convolution engine: balancing efficiency & flexibility in specialized computing. In *ACM SIGARCH Computer Architecture News* number 3. Vol. 41. ACM, 24–35.

[7] Norman P Jouppi et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*. IEEE, 1–12.

[8] Yuhao Zhu, Matthew Mattina, and Paul Whatmough. 2018. Mobile machine learning hardware at arm: a systems-on-chip (soc) perspective. *arXiv preprint arXiv:1801.06274*.

[9] Brandon Reagen, Robert Adolf, Paul Whatmough, Gu-Yeon Wei, and David Brooks. 2017. Deep learning for computer architects. *Synthesis Lectures on Computer Architecture*, 12, 4, 1–123.

[10] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. 2016. Minerva: enabling low-power, highly-accurate deep neural network accelerators. In *ACM SIGARCH Computer Architecture News* number 3. Vol. 44. IEEE Press, 267–278.

[11] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2017. Eyeriss: an energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52, 1, 127–138.

[12] Zidong Du et al. 2017. An accelerator for high efficient vision processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36, 2, 227–240.

[13] Yao Fu, Ephrem Wu, Ashish Sirasao, Sedny Attia, Kamran Khan, and Ralph Wittig. 2016. Deep learning with int8 optimization on xilinx devices. *White Paper*.

[14] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. 2016. Isaac: a convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH Computer Architecture News*, 44, 3, 14–26.

[15] Miao Hu et al. 2016. Dot-product engine for neuromorphic computing: programming 1t1m crossbar to accelerate matrix-vector multiplication. In *Proceedings of the 53rd annual design automation conference*. ACM, 19.

[16] [n. d.] Azure IoT Edge. https://azure.microsoft.com/en-us/services/iot-edge. Accessed: 2018-7-24. (). https://azure.microsoft.com/en-us/services/iot-edge.

[17] [n. d.] Google Edge TPU. https://cloud.google.com/edge-tpu. Accessed: 2018-7-24. (). https://cloud.google.com/edge-tpu.

[18] [n. d.] Snapdragon Artificial Intelligence. https://www.qualcomm.com/snapdragon/artificial-intelligence. Accessed: 2018-7-24. (). https://www.qualcomm.com/snapdragon/artificial-intelligence.

[19] Paavo Pärssinen. [n. d.] Modern mobile graphics processors. *Science: Internet, Data and Things (CS-E4000), Spring 2018*, 211.

[20] [n. d.] FPGA Cloud server. https://cn.aliyun.com/product/ecs/fpga. Accessed: 2018-7-24. (). https://cn.aliyun.com/product/ecs/fpga.

[21] [n. d.] FPGA Cloud Compute. https://cloud.baidu.com/product/fpga.html. Accessed: 2018-7-24. (). https://cloud.baidu.com/product/fpga.html.

[22] [n. d.] Amazon EC2 Instances with Up to 8 NVIDIA Tesla V100 GPUs (P3). https://goo.gl/GyrFWN. Accessed: 2018-7-24. (). https://goo.gl/GyrFWN.

[23] Guy E Blelloch. 1989. Scans as primitive parallel operations. *IEEE Transactions on computers*, 38, 11, 1526–1538.

[24] Guy E Blelloch, Michael A Heroux, and Marco Zagha. 1993. Segmented operations for sparse matrix computation on vector multiprocessors. Tech. rep. Carnegie-Mellon Univ Pittsburgh PA School of Computer Science.

[25] Guy E Blelloch. 1990. Prefix sums and their applications. Tech. rep. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University.

[26] Michael D McCool, Arch D Robison, and James Reinders. 2012. *Structured parallel programming: patterns for efficient computation*. Elsevier.

[27] Jean-Baptiste Tristan, Joseph Tassarotti, and Guy Steele. 2015. Efficient training of lda on a gpu by mean-for-mode estimation. In *International Conference on Machine Learning*, 59–68.

[28] Xiaolong Xie, Yun Liang, Xiuhong Li, and Wei Tan. 2018. Culda_cgs: solving large-scale lda problems on gpus. *arXiv preprint arXiv:1803.04631*.

[29] Rory Mitchell and Eibe Frank. 2017. Accelerating the xgboost algorithm using gpu computing. *PeerJ Computer Science*, 3, e127.

[30] Timothy M Chan. 2010. More algorithms for all-pairs shortest paths in weighted graphs. *SIAM Journal on Computing*, 39, 5, 2075–2089.

[31] [n. d.] NVIDIA cuBLAS. https://developer.nvidia.com/cublas. Accessed: 2018-7-24. (). https://developer.nvidia.com/cublas.

[32] [n. d.] NVIDIA cuDNN. https://developer.nvidia.com/cudnn. Accessed: 2018-7-24. ().

[33] [n. d.] NVIDIA CUTLASS. https://devblogs.nvidia.com/cutlass-linear-algebra-cuda. Accessed: 2018-7-24. ().

[34] D Merrill. 2018. CUB v1.8.0: CUDA Unbound, a library of warp-wide, blockwide, and device-wide GPU parallel primitives. *NVIDIA Research*.