TOPS: Accelerating Reduction Using Tensor Core Units

Abdul Dakkak*, Cheng Li*, Jinjun Xiong†, Wen-Mei Hwu*
{dakkak, cli99, w-hwu}@illinois.edu, jinjun@us.ibm.com
*University of Illinois Urbana-Champaign, †IBM Research Yorktown

center for cognitive computing systems research
IBM | ILLINOIS

Motivation

- Driven by deep learning, there has been a surge of specialized processors for matrix multiplication — Tensor Core Units (TCUs), e.g. Nvidia Tensor Cores, Google's TPU, Intel KNL's AVX, Apple A11's Neural Engine
- TCUs are used to accelerate convolutional (CNN) and recurrent neural networks (RNN) in deep learning workloads
- TCUs 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 TCUs to express reduction show the benefits — in terms of program simplicity, efficiency, and performance compared to start-of-the-art reduction methods on the GPU

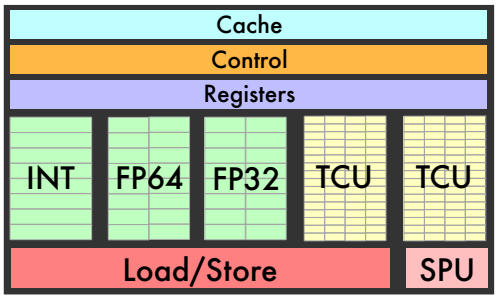


Figure 1: Each processing block (subcore) in the NVIDIA Tesla V100 PCI-E architecture contains 2 TCUs. Currently algorithms other than GEMM do not utilize the TCUs - resulting in low chip utilization.

Nvidia Tensor Cores

- Tensor Cores have been only used to accelerate GEMM operations, most prominently through NVIDIA's CUDA libraries — cuBLAS [1], cuDNN [2] and CUTLASS [3]
- NVIDIA also provides a CUDA C++ Warp Matrix Multiply and Accumulate (WMMA) API to program the Tensor Cores directly
- WMMA only supports warp-level matrix multiplication with dimensions <16, 16, 16>, <8, 16, 32>, <32, 16, 8>

```
1  #include <mma.h>
2  using namespace nvcuda::wmma;
3  __global__ void dot_wmma_16x16(half *a, half *b, half *c) {
4     fragment<matrix_a, 16, 16, 16, half, col_major> a_frag;
5     fragment<matrix_b, 16, 16, 16, half, row_major> b_frag;
6     fragment<accumulator, 16, 16, 16, half> c_frag;
7     load_matrix_sync(a_frag, a, /* leading dim */ 16);
8     load_matrix_sync(b_frag, b, /* leading dim */ 16);
9     fill_fragment(c_frag, 0.0f);
10    mma_sync(c_frag, a_frag, b_frag, c_frag);
11    store_matrix_sync(c, c_frag, 16, row_major);
12 }
```

Figure 2: A simple CUDA kernel performing <16, 16, 16> matrix multiplication (C = A . B + C) within a warp in half precision using the CUDA WMMA API.

The current WMMA API provides **warp-level matrix operations** for

- matrix load (load_matrix_sync)
- matrix store (store_matrix_sync)
- 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.

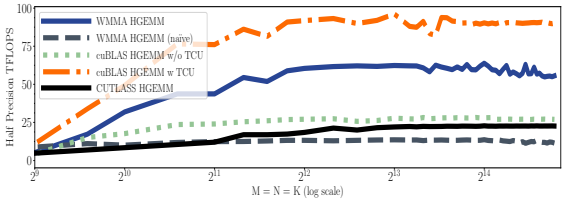
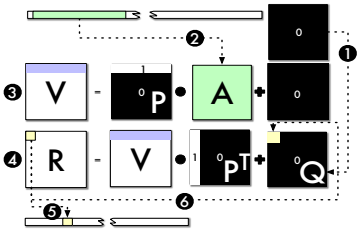


Figure 3: GEMM performance using Tensor Cores on a V100 PCI-E GPU with a 113 TFLOPS peak performance. The inputs are square matrices with variable <M , N , K> dimensions.

- We evaluated the GEMM performance using Tensor Cores through cuBLAS, CUTLASS (version 0.1.1), and hand written kernels using the WMMA API
- We see that the matrix-multiplication performance of NVIDIA TCUs is high enough to tolerate resource and computation waste in algorithms.
- Driven by this observation, we formulate the mapping between the widely used reduction collective and TCUs

Map Reduction onto TCUs



We developed a library of warp-, block-, and grid-level primitives which can be auto-tuned for different architectures and will plan on releasing in the near future

Figure 4: The work-inefficient Reduction256N algorithm (1) initializes the Q matrix with all zeros and (2) loads the 256 input elements into a matrix A in column major order. (3) A dot product V = P.A + 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. (4) the dot product R = V.PT + Q reduces the first row into a scalar. (5) 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 (6) the first element of R matrix is used as the first element of the Q matrix and the procedure is iterated starting from step (2).

Evaluation

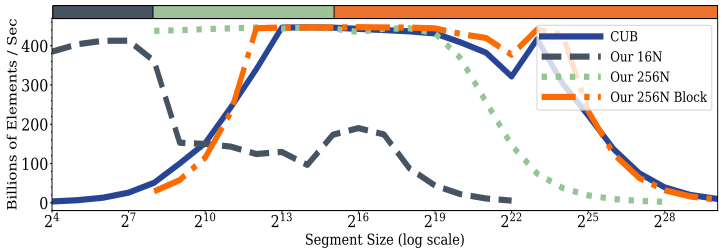


Figure 5 Segmented reduction for the algorithms presented on different segment sizes (between 16 and 230) for a fixed 2^30 element list. The bar on top of the figure shows the best performing algorithm for each range of segment sizes.

- We evaluated our TCU reduction algorithm against the state-of-the-art implementation from CUB [5] on different segment sizes for a fixed 2^{30} element list
- Through a combination of the algorithms presented, we are able to achieve within 90% and 98% of ideal throughput
- Our algorithm achieves this while decreasing the power consumption by up to 22%

Conclusion

Although this work targets GPUs, the motivation, methods, and observations are applicable to a wide number of TCU implementations and microarchitectures.

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.

References

[1] NVIDIA cuBLAS. https://developer.nvidia.com/cublas.
[2] NVIDIA cuDNN. https://developer.nvidia.com/cudnn.
[3] NVIDIA CUTLASS. https://devblogs.nvidia.com/cutlass-linear-algebra-cuda.
[4] NVIDIA Tensor Cores. https://www.nvidia.com/en-us/data-center/tensorcore.
[5] D Merrill. CUB v1.8.0: CUDA Unbound, a library of warp-wide, blockwide, and device-wide GPU parallel primitives. NVIDIA Research, 2018.

Contact me at dakkak@illinois.edu