Optimizing Batch HGEMM on Small Sizes Using Tensor Cores
Ahmad Abdelfattah, Stanimire Tomov, and Jack Dongarra

Abstract
We present an optimized batch GEMM for FP16 arithmetic using NVIDIA's Tensor Core Technology. The proposed design strategy takes advantage of the low-level warp matrix functions to provide a highly flexible GPU kernel that provides a lot of controls to the developer. We also pay particular attention to the multiplications of very small matrices that cannot fully occupy the tensor core units. Our results show that the proposed design can outperform cuBLAS for sizes up to 100 by factors between 1.2x and 10x using a Tesla V100 GPU. For very small matrices, the observed speedups range between 1.8x and 26x. This development is part of the MAGMA library [1].

Motivation
Half-pitch matrix multiply is at the center of the AI revolution and other areas such as:
- Deep learning
- CNN algorithms based on minimal filtering [2]

Design Challenges
CUDA provides a set of Warp Matrix Functions that can be used to program the tensor core units. Tensor cores inputs and outputs are stored in fragments.

Known Constraints
- Fragments are opaque objects of restricted sizes.
- The sizes (M, K, N) are limited to (16, 16, 16), (32, 16, 8), and (8, 32, 16).
- Loading and storing fragments may cause leading dimensions that are multiples of 16 bytes.
- No direct support for scaling. Recall that a standard GEMM performs: \( C = a \cdot A \cdot B + C \), where \( a \) and \( b \) are scalars.
- All operations (load/store/multiply) must be done using a full warp.

Design Strategy
The developed kernel performs a standard GEMM operation \( C = A \cdot B + C \), on a batch of relatively small matrices, all having the same dimensions. The size of the batch is called batchcount.

Grid Design
The output matrices are subdivided into small blocks of size \( \text{BLK}_M \times \text{BLK}_N \times \text{BLK}_K \). The grid configuration of the kernel is a 3D thread block array of size \( [\text{DIM}_X, \text{DIM}_Y, \text{DIM}_Z] \). The grid is organized as a 1D array of subgrids, each having a unique identifier called batchid.

Within every subgrid, each TB is responsible for computing a block of the corresponding output matrix by reading a block row of \( A \) and a block column of \( B \).

Thread Block Design
Thread blocks utilize a double-sided recursive blocking technique which recursively subdivides data blocks from \( A, B, \) or \( C \) in two different ways, depending on which phase of the kernel is being executed. During memory operations, threads are organized as a 2D \( \text{DIM}_X \times \text{DIM}_Y \) configuration that reads each data block using a double loop-nest. During compute, however, threads are organized as individual warps in order to utilize the tensor core units. During the latter stage, the data blocks are subdivided using permissible tensor core sizes \( (\text{TC}_M, \text{TC}_N, \text{TC}_K) \).

Data blocks from \( A \) and \( B \) are cached in shared memory and are loaded into fragments by each warp as required. Warps in the same thread block loop over the data blocks of \( C \) in a round-robin style.

Three different designs for very small sizes
- Mainly target sizes less \( \leq 16 \). These sizes do not fully occupy the tensor core units when using cuBLAS.
- A different design approach is required.

MAGMA-SMALL-V1
- One warp performs one GEMM using tensor cores.
- Assign multiple warps per thread block to improve latency hiding.

MAGMA-SMALL-V2
- One warp performs simultaneous GEMMs using one tensor core operation.
- Maximum number of simultaneous GEMMs is: \( \min(\text{TC}_M \times \text{TC}_N \times \text{TC}_K) / (16, 16, 16) \).
- Matrices stored in fragments as shown below, assuming \( (\text{TC}_M, \text{TC}_N, \text{TC}_K) = (16, 16, 16) \).

MAGMA-SMALL-V3
- Does not use tensor cores.
- Automatic code generation for every size using C++ templates.
- The percentage utilization of the tensor core compute power in MAGMA-SMALL-V2 is less than \(-25\%\), so MAGMA-SMALL-V3 can outperform MAGMA-SMALL-V2 (e.g. for square sizes \( \leq 10 \)).
- Such a ratio is equivalent to the peak FP16 performance without tensor cores (31.7 Tflop/s) to the peak performance with tensor cores (125.8 Tflop/s) on the V100 GPU.

Conclusions
- The proposed batch HGEMM outperforms cuBLAS for relatively small sizes.
- Speedups range between 1.2x and 26x.
- A novel approach for addressing tensor core utilization for extremely small problems.

Future directions
- Comprehensive autotuning for various shapes and transpositions.
- Support for variable size batches.
- Engagement with application developers.

Acknowledgments
This work was supported by the Exascale Computing Project, a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This work was also partially supported by the National Science Foundation under Grant OAC-1740250 and NVIDIA. This work was also partially supported by the National Science and the National Nuclear Security Administration.

Performance Results
- MAGMA-SMALL-V1 is a clear winner for square sizes \( \geq 10 \), with about 2x speedup against cuBLAS (top figure).
- MAGMA-SMALL-V2 performs MAGMA-SMALL-V2 for sizes less than 10 (top figure). It is up to 25.8x faster than cuBLAS.
- While MAGMA-SMALL-V2 has no winning scenario for square sizes, it can be the best-performing kernel if \( M \) and \( K \) are less than 9, while having a relatively large \( K \) (bottom figure). Small \( M \) and \( K \) ensure multiple GEMMs per a tensor core operation. A large value of \( K \) leads to better utilization of the tensor cores compute power.

Optimization Techniques for Extremely Small Sizes
- Support for variable size batches.
- CUDA provides a set of Warp Matrix Functions that can be used to program the tensor core units. Tensor cores inputs and outputs are stored in fragments.

References