Optimizing Batch HGEMM on Small Sizes Using Tensor Cores

Ahmad Abdelfattah, Stanimire Tomov, and Jack Dongarra

ICL · THE UNIVERSITY OF TENNESSEE KNOXVILLE

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 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 precision matrix multiply is at the core of the AI revolution and other areas such as:

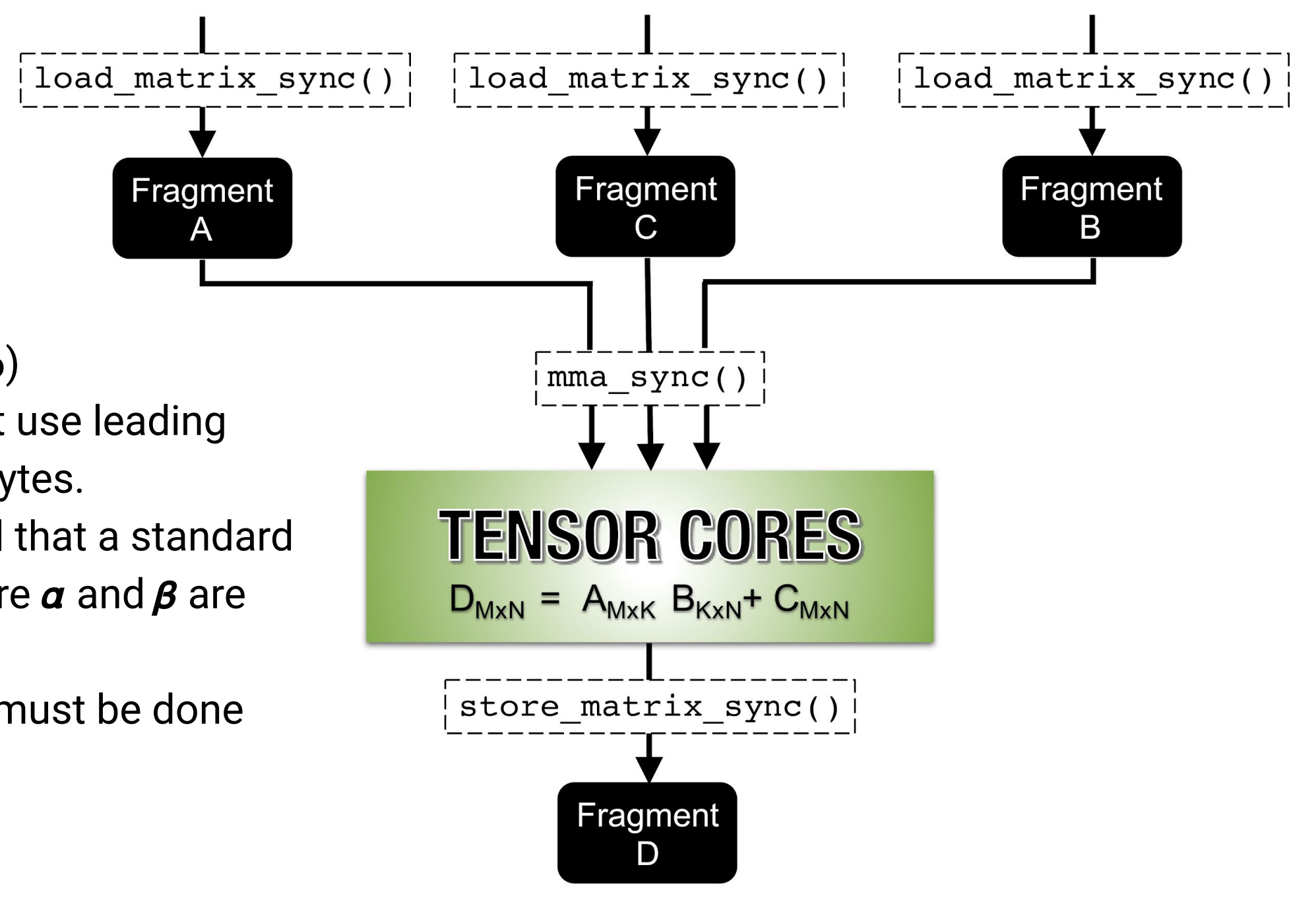
- Deep learning
- CNN algorithms based on minimal filtering [2]
- Mixed precision algorithms and linear solvers [3]

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, N, K) are limited to (16, 16, 16), (32, 8, 16), and (8, 32, 16)
- Loading and storing fragments must use leading dimensions that are multiple of 16 bytes.
- No direct support for scaling. Recall that a standard GEMM performs: C = αAB + βC, where α and β are scalars.
- All operations (load/store/multiply) must be done using a full warp

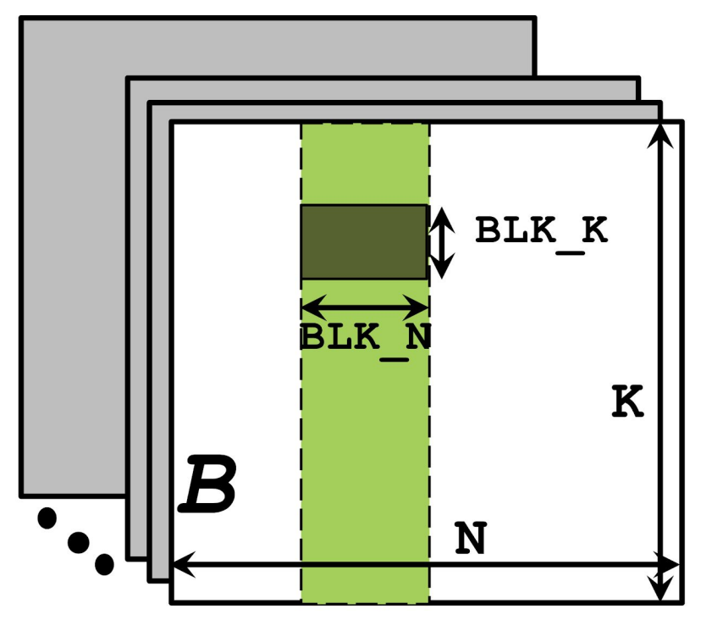


Design Strategy

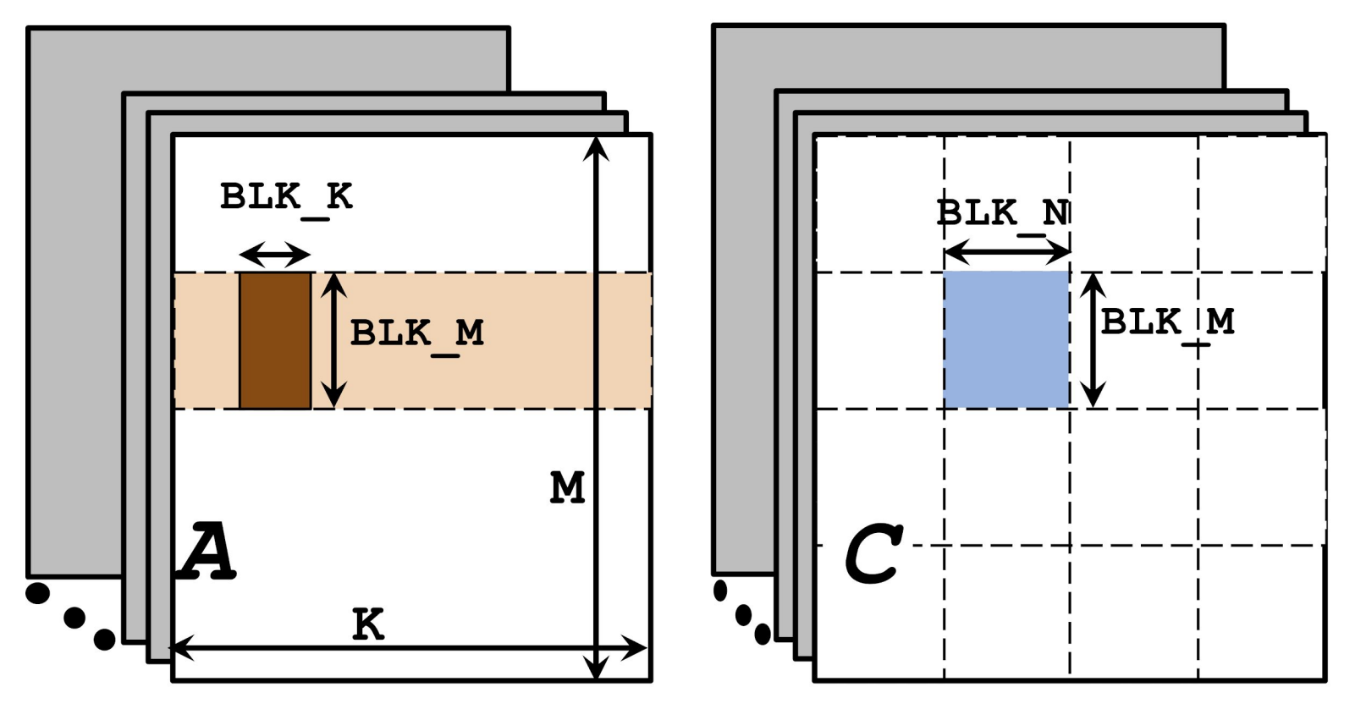
The developed kernel performs a standard GEMM operation C = αAB + βC, on a batch of relatively small matrices, all having the same dimensions. The size of the batch is called `batchCount`.

Grid Design

The output matrices are subdivided into small blocks of size `BLK_M` x `BLK_N`. The grid configuration of the kernel is a 3D thread block array of size (⌈`M÷BLK_M`⌉, ⌈`N÷BLK_N`⌉, `batchCount`). The grid is organized as a 1D array of subgrids, each having a unique identifier called `batchid`.



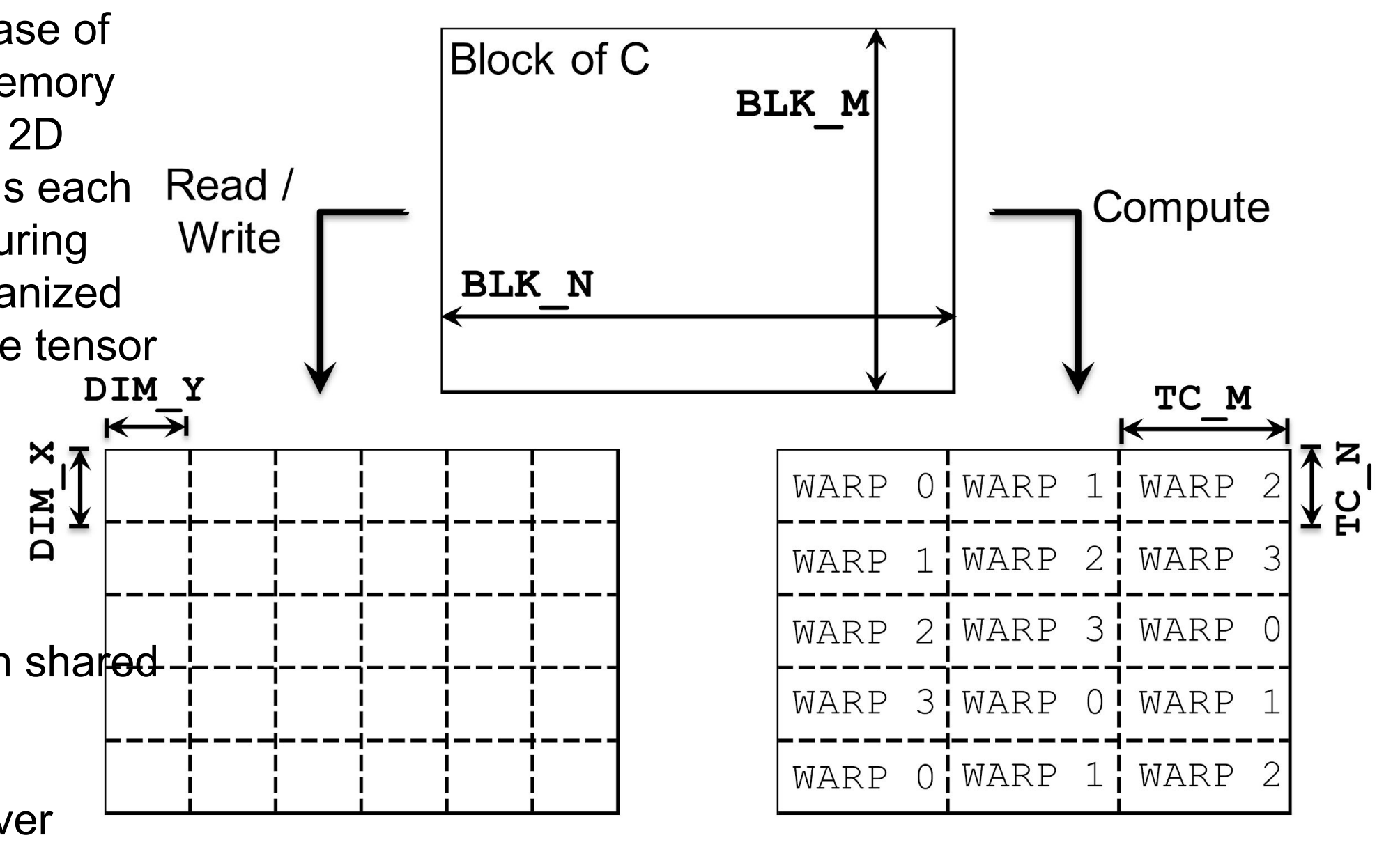
Within every subgrid, each TB is responsible for computing a block of the corresponding output matrix by reading a block row of A and a block column of B.



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 `DIM_X` x `DIM_Y` configuration that reads each data block using a double loop-nest. During computation, 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 (`TC_M`, `TC_N`, `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.



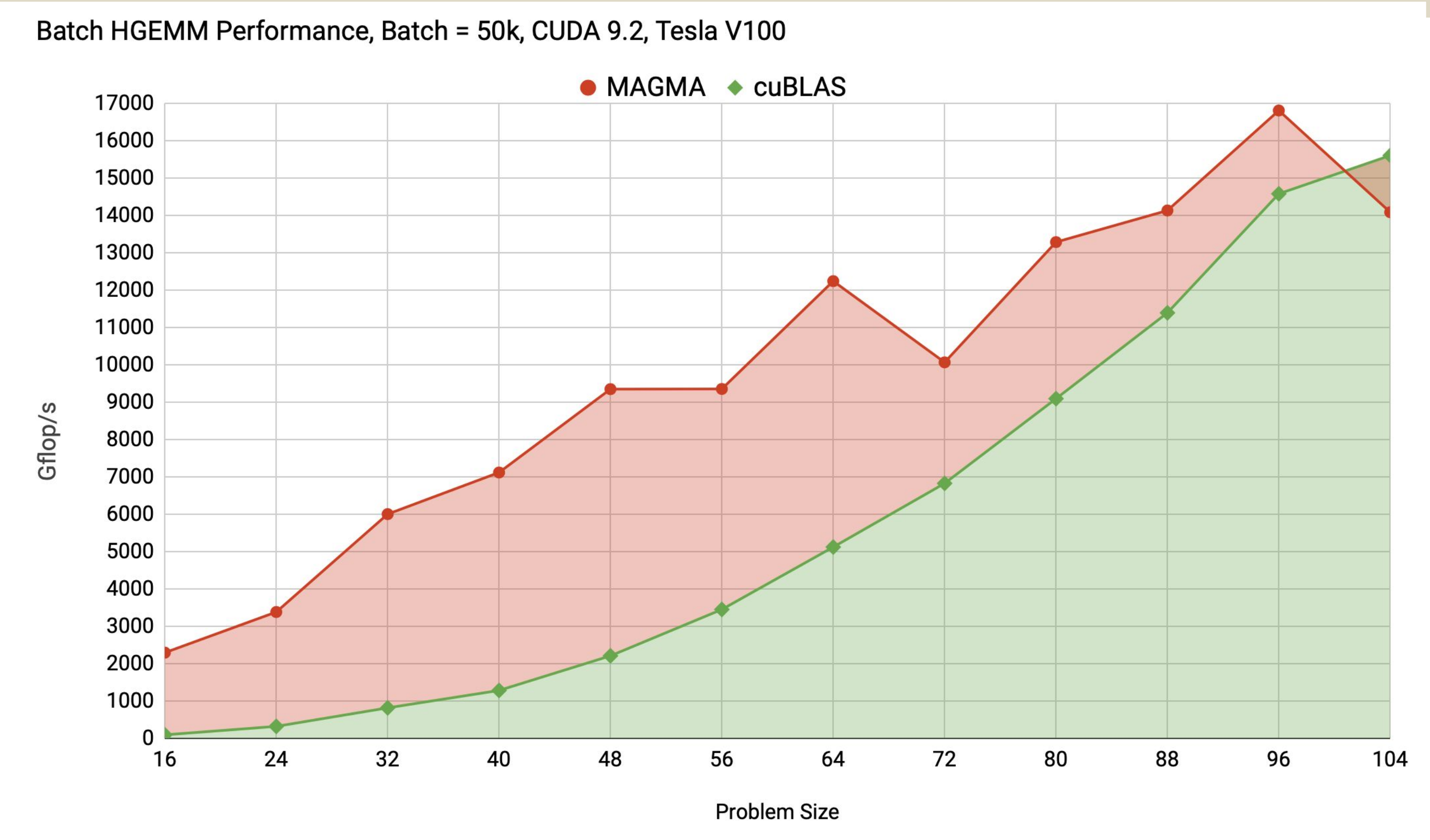
Autotuning and Results on Medium Sizes

Kernel Autotuning

- CUDA C++ templates are used
- 8 tuning parameters in total: (`DIM_X`, `DIM_Y`, `BLK_M`, `BLK_N`, `BLK_K`, `TC_M`, `TC_N`, `TC_K`).
- Using "soft constraints", the number of eligible kernel instances is reduced from about 15,000 to 4,948
- Automated scripts are used to generate the sample space, evaluate each kernel instance, and analyze the collected results to choose the winning kernels
- The performance of MAGMA is compared against cuBLAS (CUDA 9.2), with `CUBLAS_TENSOR_OP_MATH` turned on

Performance summary

- MAGMA outperforms cuBLAS for sizes less than 100
- Speedup ranges between 1.2x and 10.5x
- The smaller the size, the larger the speedup
- cuBLAS has the advantage asymptotically



Optimization Techniques for Extremely Small Sizes

Three different designs for very small sizes

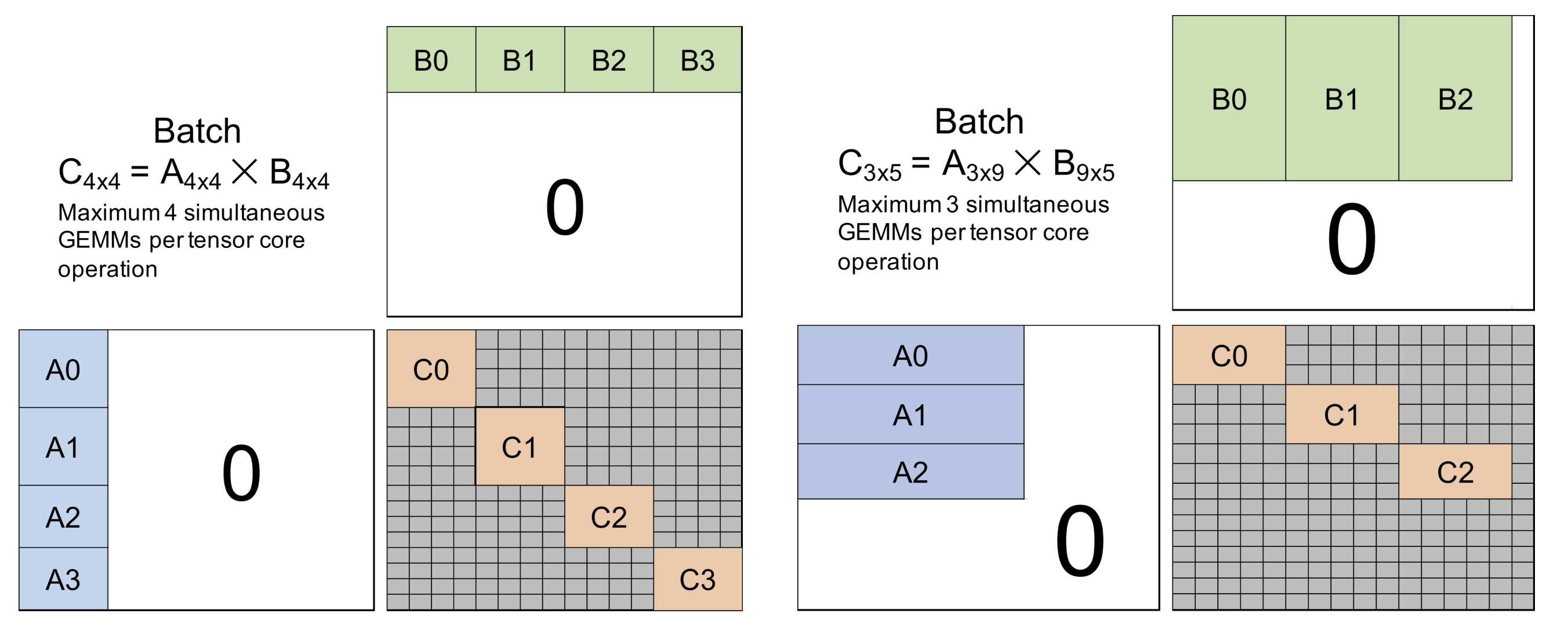
- Mainly target sizes less ⩽ 16. These sizes do not fully occupy the tensor core units when using `mma_sync()`
- A different design approach is required

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(⌊`TC_M ÷ M`⌋, ⌊`TC_N ÷ N`⌋)
- Matrices stored in fragments as shown below, assuming (`TC_M`, `TC_N`, `TC_K`) = (`16`, `16`, `16`)

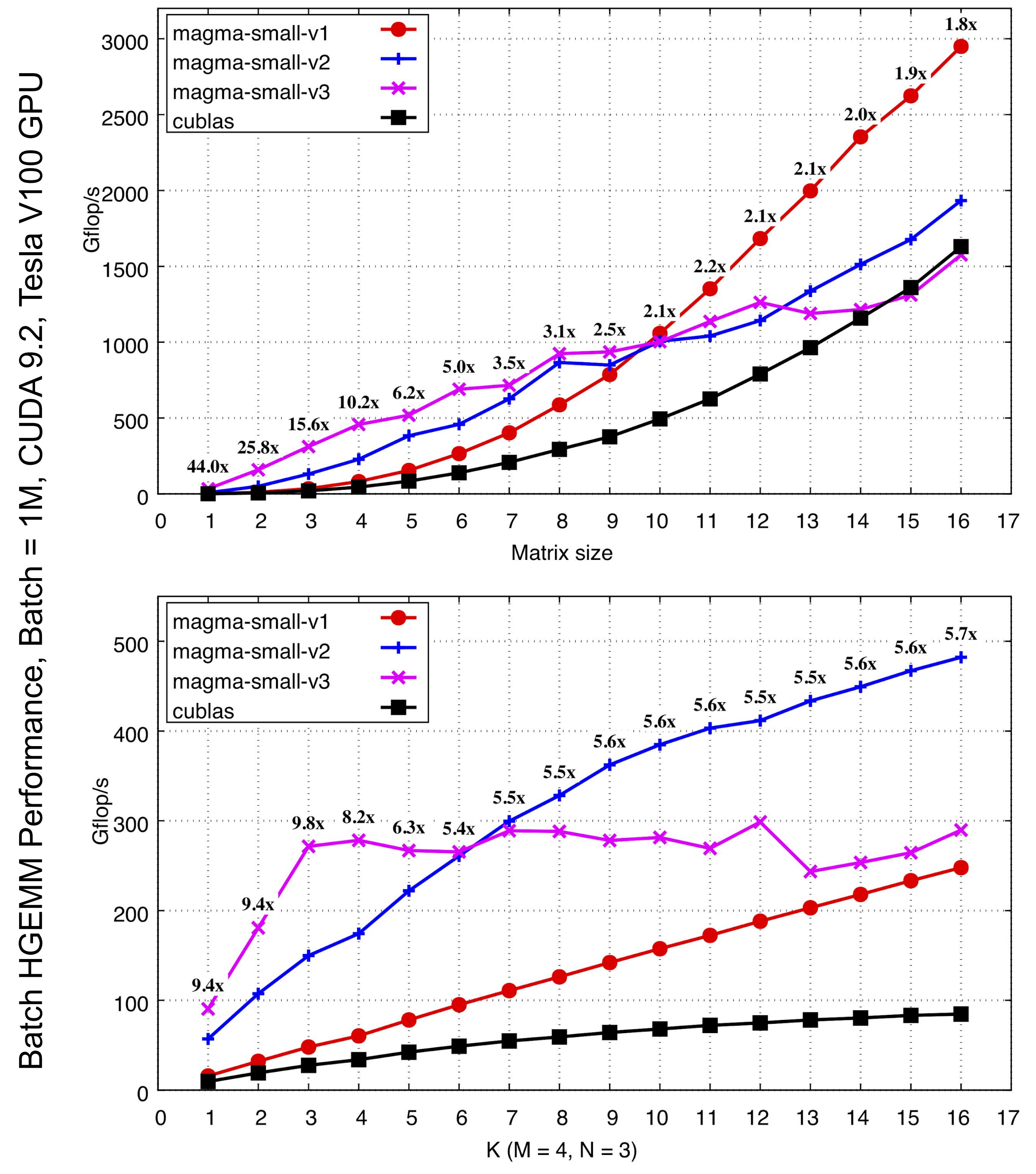

Batch
$C_{4x4} = A_{4x4} \times B_{4x4}$
Maximum 4 simultaneous GEMMs per tensor core operation
B0 B1 B2 B3
0
A0 A1 A2 A3
0
C0 C1 C2 C3
Batch
$C_{3x5} = A_{3x9} \times B_{9x5}$
Maximum 3 simultaneous GEMMs per tensor core operation
B0 B1 B2
0
A0 A1 A2
0
C0 C1 C2

MAGMA-SMALL-V3

- Does not use tensor cores
- Automatic code generation for every size using C++ templates
- If the percentage utilization of the tensor core compute power in **MAGMA-SMALL-V2** is less than ~**25%**, then **MAGMA-SMALL-V3** can outperform **MAGMA-SMALL-v2** (e.g. for square sizes ⩽ 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 (**125.0 Tflop/s**) on the V100 GPU

Performance Results

- **MAGMA-SMALL-V1** is a clear winner for square sizes > 10, with about 2x speedup against cuBLAS (top figure)
- **MAGMA-SMALL-V3** outperforms **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 **N** are less than 9, while having a relatively large K (bottom figure). Small **M** and **N** ensure multiple GEMMs per a tensor core operation. A large value of **K** leads to better utilization of the tensor cores compute power.



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

Acknowledgement 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.

REFERENCES: [1] Matrix Algebra on GPU and Multicore Architectures (MAGMA), https://icl.utk.edu/magma/
[2] S. Winograd, S. for Industrial, A. Mathematics, C. B. of the Mathematical Sciences, and N. S. F. E. U. d'Am`erica) *Arithmetic Complexity of Computations*, ser. CBMS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics, 1980.
[3] A. Haidar, S. Tomov, J. Dongarra, and N. Higham. 2018. *Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision* iterative *refinement solvers*. SC'18