Efficient Quantized Sparse Matrix Operations on Tensor Cores

Shigang Li*, Kazuki Osawa*, Torsten Hoefler+

*School of Computer Science, Beijing University of Posts and Telecommunications
+Department of Computer Science, ETH Zurich
Efficient Quantized Sparse Matrix Operations on Tensor Cores
Shigang Li, Kazuki Osawa, Torsten Hoefler
SC22, Dallas, TX, USA
Nov. 2022
Model size is growing exponentially

- Even if the model can be fitted in a single GPU (for example, by swapping parameters between host and device memory), the high number of compute operations required can result in unrealistically long training times without parallelization. For example, training a GPT-3 model with 175 billion parameters would take 36 years on eight V100 GPUs, or seven months with 512 V100 GPUs.
Models are also compressible

- **Sparsification**
  - **SpMM**
    1. Self-attention in sparse Transformers
    2. Forward pass of pruned models
    ...
  - **SDDMM**
    1. Attention score in sparse Transformers
    2. Backward pass of pruned models
    ...

- **Quantization**
  - Uniform symmetric quantization
    - \(-\max(|W_i|)\) to 0.0 to \(\max(|W_i|)\)
    - real (fp16)
    - integer (int8)
  - Uniform asymmetric quantization
    - \(-\min(W_i)\) to 0.0 to \(\max(W_i)\)
    - real (fp16)
    - integer (int8)

- **Combining sparsification with quantization**
  - Mart van Baalen et al., Bayesian bits: Unifying quantization and pruning, NeurIPS 2020
  - H. Yang et al., Automatic neural network compression by sparsity-quantization joint learning: A constrained optimization based approach, CVPR 2020
  - S. Han et al., Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, ICLR 2016
  ...

**Sparsity in scientific:** > 99%

**Sparsity in DL:** 50% ~ 90%
Tensor cores for deep learning acceleration

Pascal, 2016
(18.7 TFLOPS for fp16)

Volta, 2017
(125 TFLOPS for fp16)

Turing, 2018
(261 TOPS for int8)

Ampere, 2020
(624 TOPS for int8)

Hopper, 2022
(2,000 TOPS for int8)

Images and GIFs in this slide are from https://www.nvidia.com/en-us/data-center/tensor-cores/
Challenges

(1) How to achieve practical speedup in a large range of sparsity ratio, e.g., 50% ~ 98%?

(2) How to efficiently support sparse workloads with mixed precision (two input matrices with different precision), e.g., 8-bit weights and 4-bit activation?

How to bridge the big gap?

SpMM with 2x1 block sparsity on A100 tensor cores, using 1,536 sparse matrices from DL domain

| Supported Tensor Core precisions | Hopper | Ampere | Turing | Volta |
|---------------------------------|--------|--------|--------|-------|
| FP64, TF32, bfloat16, FP16, FP8, INT8 | FP64, TF32, bfloat16, FP16, INT8 | FP16, INT8, INT4, INT1 | FP16 |

Two input matrices must be the same precision

Sparse self-attention with mixed precision
Libraries of sparse matrix computation

1 Mixed precision means two input matrices with different precision

| Library          | Precision | Sparsity     | Tensor Core |
|------------------|-----------|--------------|-------------|
|                  | fp16     | int8 | int4 | mixed | granularity | DL-friendly? |
| cuSPARSE [10]    | ✓         | ✓   | ✓   | ✓    | fine-grained block | ✓     | ✓     |
| cuSPARSELt [11]  | ✓         | ✓   | ✓   | ✓    | 2:4 structured | ✓     | ✓     |
| Sputnik [13]     | ✓         | x   | x   | x    | fine-grained   | ✓     | ✓     |
| vectorSparse [14]| ✓         | x   | x   | x    | 1-D block     | ✓     | ✓     |
| **Magicube (ours)** | ✗        | ✓   | ✓   | ✓    | 1-D block     | ✓     | ✓     |

Sparse matrix with 1-D non-zero blocks

2×1 1-D block

Columns

Rows
Data layout of $m8n8k16$ for \textit{int8 mma} on Tensor Cores

**Thread0** provides $b_{00}$, $b_{10}$, $b_{20}$, and $b_{30}$, and each $b_{xx}$ is an 8-bit integer.

**Thread0** provides $a_{00}$, $a_{01}$, $a_{02}$, and $a_{03}$, and each $a_{xx}$ is an 8-bit integer.

**A** (row-major)

| Col | Row | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   | 10  | 11  | 12  | 13  | 14  | 15  |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | T0: | {a00, a10, a20, a30} | T1: | {a01, a11, a21, a31} | T2: | {a02, a12, a22, a32} | T3: | {a03, a13, a23, a33} |
| 1   | T4: | {a04, a14, a24, a34} | T5: | {a05, a15, a25, a35} | T6: | {a06, a16, a26, a36} | T7: | {a07, a17, a27, a37} |
| ... |    |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| 7   | T28:| {a08, a18, a28, a38} | T29:| {a09, a19, a29, a39} | T30:| {a10, a20, a30, a40} | T31:| {a11, a21, a31, a41} |

**B** (column-major)

| Col | Row | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   | 10  | 11  | 12  | 13  | 14  | 15  |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | T0: | {b00, b10, b20, b30} | T1: | {b11, b21, b31}   | T2: | {b12, b22, b32}   | T3: | {b13, b23, b33}   |
| 1   | T4: | {b14, b24, b34}   | T5: | {b15, b25, b35}   | T6: | {b16, b26, b36}   | T7: | {b17, b27, b37}   |
| ... |    |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| 7   | T28:| {b28, b29, b30}   | T29:| {b29, b30}   | T30:| {b30}   | T31:| {b30}   |

$m = 8$, $n = 8$, $k = 16$, $c_{xx}$ is an int32
SR-BCRS sparse matrix format

Sparse matrix with 1-D block non-zeros, the length of the 1-D block = 2, 4, or 8

SR-BCRS (ours) is more friendly to Tensor Cores

Row pointers = [0, 4, 10, 13]
Column indices = [1, 3, 6, 8, 1, 2, 5, 7, 8, 11, 0, 4, 9]
Values = a b c e g i k m o q s t u v w x y z

Row pointers = [0, 4, 4, 10, 12, 15]
Values = [1, 3, 6, 8, 1, 2, 5, 7, 8, 11,   ,   , 0, 4, 9,   ]
Column indices =

T0: \{(a, an, an, an)\} T1: \{(an, an, an, an)\} T2: \{(an, an, an, an)\} T3: \{(an, an, an, an)\}
T4: \{(an, an, an, an)\} T5: \{(an, an, an, an)\} T6: \{(an, an, an, an)\} T7: \{(an, an, an, an)\}
T8: \{(an, an, an, an)\} T9: \{(an, an, an, an)\} T10: \{(an, an, an, an)\} T11: \{(an, an, an, an)\}
SpMM in Magicube

(a) SpMM

(b) SpMM in Magicube at thread-block level
Load rows of matrix B to shared memory for int8

BS_n = 64
BS_k = 16

Coalesced into a 64-byte transaction
Load blocks of matrix B to shared memory for int8

![Diagram showing load blocks of matrix B to shared memory for int8]
Local transpose on registers for int8

- `Shared memory`:
  - `bank0`
    - `T0`
  - `bank16`
    - `T0`  
  - `bank0`  
    - `T0`  
  - `bank16`
    - `T0`  

- `load`:
  - `8-bit integer`
  - `Registories`
    - `0 4 8 12`
    - `1 5 9 13`
    - `2 6 10 14`
    - `3 7 11 15`

- `transpose in char`:
  - `input of mma0`
  - `input of mma1`
  - `input of mma2`
  - `input of mma3`

- `one bank`:
  - `8-bit integer`
  - `BSn = 64`
    - `T0`  
      - `14 Bank 0 ~ 7 T24 T28`
      - `14 Bank 16 ~ 23 T24 T28`
      - `Bank 8 ~ 15`
      - `Bank 24 ~ 31`

- `Padding: Bank 0 ~ 7`:
  - `T1`  
    - `15 Bank 8 ~ 15 T25 T29`
    - `Bank 16 ~ 23`
    - `Bank 0 ~ 7`

- `BSk = 16`
  - `T2`  
    - `16 Bank 0 ~ 7 T26 T30`
    - `Bank 8 ~ 15`

- `Padding: Bank 8 ~ 15`:
  - `T3`  
    - `17 Bank 24 ~ 31 T27 T31`
    - `Bank 0 ~ 7`

- `Warp0`  
- `Warp1`
Local transpose on registers for int8

Data layout of m8n8k16 for int8 mma
**MMAs in SpMM with int8**

The warp-level view of MMAs in SpMM with int8

- **Shared memory**
  - bank0
  - bank16
  - bank0
  - bank16

- **Load**
  - 8-bit integer

- **Transpose in char**
  - Bank 0
  - Bank 16

- **A block**
  - Shared among warps (on SM)

- **B block**
  - After transpose on **registers** by warp0
    - `4*n = 32`
  - After transpose on **registers** by warp1
    - `4*n = 32`

- **The warp-level view of MMAs in SpMM with int8**
Efficient local transpose for int4 with indices shuffling

Column indices of the sparse matrix A

1. Block-wise indices shuffling

Load data of the dense matrix B (via SM) to registers

4. Transpose in char

5. Split

6. Mask and shift

7. Bitwise OR

8-bit

low 4-bit

green 4-bit

low 4-bit

green 4-bit

0 2 4 6
1 3 5 7
0 2 4 6
1 3 5 7

0 2 4 6
1 3 5 7
0 2 4 6
1 3 5 7

int32
0 2 4 6
1 3 5 7
0 2 4 6
1 3 5 7

Bitwise OR

int32
0 2 4 6
1 3 5 7
0 2 4 6
1 3 5 7

192 bitwise operations

32 bitwise operations
Prefetch data blocks of matrix B of SpMM

Algorithm 1 Prefetch the data block of dense matrix B

\[ \text{steps} = \frac{\text{nnz}}{BS_k}; \]

\begin{align*}
\text{Load}_A\text{\_values\_and\_indices\_to\_shared}(0); \\
\_\text{syncthreads}(); \\
\text{Prefetch}_B\text{\_values\_to\_registers}(0);
\end{align*}

\begin{algorithmic}
\For {i=1; i < \text{steps}; i++}
\State \text{Store}_B\text{\_values\_on\_regs\_to\_shared}(i-1);
\State \text{Load}_A\text{\_values\_and\_indices\_to\_shared}(i);
\State \_\text{syncthreads}();
\State \text{Prefetch}_B\text{\_values\_to\_registers}(i);
\State \text{MMA\_compute\_tiles}(i-1);
\State \_\text{syncthreads}();
\EndFor
\end{algorithmic}

\begin{align*}
\text{Store}_B\text{\_values\_on\_regs\_to\_shared}(i-1); \\
\_\text{syncthreads}(); \\
\text{MMA\_compute\_tiles}(i-1);
\end{align*}

\begin{itemize}
\item \text{Cold start}
\item \text{Load data and indices to SM}
\item Overlap prefetch with MMA
\item The tail of pipeline
\end{itemize}
SDDMM in Magicube

(a) SDDMM

(b) SDDMM in Magicube at thread-block level
MMAs in SDDMM

The thread-block level view of SDDMM

The warp-level view of MMAs in SDDMM
Mixed precision

- **a** is an 8-bit **unsigned** integer, **b** is unsigned 4-bit
  
  \[ a = 11101101 \text{ (237 in decimal)} \]

  - Split
    - \( a_{7-4} \) 1110
    - \( a_{3-0} \) 1101
  
  - Recover
    - \( a = 2^4 \cdot a_{7-4} + a_{3-0} \)
    - \( a \cdot b = 2^4 \cdot a_{7-4} \cdot b + a_{3-0} \cdot b \)

- **a** is an 8-bit **signed** integer, **b** is signed 4-bit
  
  \[ a = 11101101 \text{ (-19 in decimal)} \]

  - Split
    - \( a_{7-4} \) 1110
    - \( a_{3-0} \) 1101
  
  - Recover
    - \( a = 2^4 \cdot a_{7-4} + a_{3-0} \)
    - \( a \cdot b = 2^4 \cdot a_{7-4} \cdot b + a_{3-0} \cdot b \)

(a) Emulation of **A** (8-bit) * **B** (4-bit) using 4-bit **mma**

(b) Stacked into a single **mma** vector

\[ C = 2^0 \cdot C_0 + 2^4 \cdot C_1 \]
Evaluation

- NVIDIA A100-SXM4-40GB GPU
  - total 108 SMs
  - each SM has 192KB configurable L1 cache and shared memory, and 256KB registers
  - supported datatypes on Tensor Core: int8, int4, int1, fp16, bf16, tf32, fp64

- Compare the performance of Magicube with sparse libraries (vectorSparse, cuSPARSE) and dense libraries (cuBLAS, cuDNN)

- Micro-benchmarks: 1,536 sparse matrices from Deep Learning Matrix Collection (DLMC) with sparsity 50%~98%, dilating each scalar with 1-D blocks (length V = 2, 4, 8)

- Case study: end-to-end sparse Transformer inference

This image is from: R. Krashinsky, et al. https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/ May, 2020
Ablation study for SpMM in Magicube

Ablation study for optimizations of SpMM
SpMM with mixed precision in Magicube

$L_x \cdot R_y$ means $x$-bit A matrix multiplied by $y$-bit B matrix
Benchmarking SpMM and SDDMM

SpMM on A100 GPU using 1,536 sparse matrices

- $V = 2$
- $N = 128$
- Median speedup: 2.43x

- $V = 4$
- $N = 128$
- Median speedup: 2.38x

- $V = 8$
- $N = 128$
- Median speedup: 1.34x

SDDMM on A100 GPU using 1,536 sparse matrices

- $V = 2$
- $K = 128$
- Median speedup: 1.18x

- $V = 4$
- $K = 128$
- Median speedup: 1.32x

- $V = 8$
- $K = 128$
- Median speedup: 1.56x
End-to-end sparse Transformer inference

Attention score

Quantized self-attention with sparse attention mask

Latency of end-to-end inference of sparse Transformer

Attention score formula:

\[ \text{Attention}(Q, K, V) = \text{softmax} \left( \frac{Q K^T \odot M}{\sqrt{d_k}} \right) V \]
End-to-end sparse Transformer inference

| dense                  | sparsity=0.9     |
|-----------------------|------------------|
| PyTorch (cuDNN, fp32) | vectorSparse (fp16) |
| 57.36%                | 57.14%           |
| PyTorch (cuDNN, fp16) | Magicube (16b-8b) |
| 57.50%                | 57.32%           |
|                       | Magicube (8b-8b) |
|                       | 57.11%           |
|                       | Magicube (8b-4b) |
|                       | 56.79%           |

Test accuracy of text classification using sparse Transformer with num_heads=4 and seq_len=4,096
# Conclusion

## 1. Challenges

1. How to achieve practical speedup in a large range of sparsity ratio, e.g., 10% ~ 98%?
2. How to efficiently support sparse workloads with mixed precision (two input matrices with different precision), e.g., 8-bit weights and 4-bit activations?

### Challenges

| Supported Tensor | FP32, FP16, INT8, INT16 | FP32, FP16, INT8, INT16 | FP32, FP16, INT8, INT16 | FP32, FP16, INT8, INT16 |
|------------------|--------------------------|--------------------------|--------------------------|--------------------------|
| Low precision    | FP32, INT8               | FP32, INT8               | FP32, INT8               | FP32, INT8               |

## 2. SR-BCRS format

**SR-BCRS sparse matrix format**

- **Columns**
  - Row pointers: \([0, 4, 7, 13]\)
  - Column indices: \([0, 1, 2, 5, 8, 11, 7, 11, 13, 14]\)
- **Values**: \([-0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]\)

**SR-BCRS is more friendly to Tensor Cores**

## 3. SpMM in Magicube

### SpMM in Magicube

- **A** (dense)
- **B** (dense)
- **C** (sparse)

**Mixed precision**

- 31 bits on 8-bit unsigned integer, 8 bits on 4-bit unsigned integer.
- \(a = 123456\) in decimal
- \(b = 789\) in decimal
- \(a \times b = 123456789\)

## 4. SDDMM in Magicube

### SDDMM in Magicube

- **A** (dense)
- **B** (dense)
- **C** (sparse)

**Mixed precision**

- 31 bits on 8-bit unsigned integer, 8 bits on 4-bit signed integer.
- \(a = 123456\) in decimal
- \(b = 789\) in decimal
- \(a \times b = 123456789\)

## 5. Mixed precision

**End-to-end sparse Transformer inference**

- Quantized self-attention with sparse attention mask
- Latency of end-to-end inference of sparse Transformer

## 6. Evaluation

- https://zenodo.org/record/6924338
- https://github.com/Shigangli/Magicube