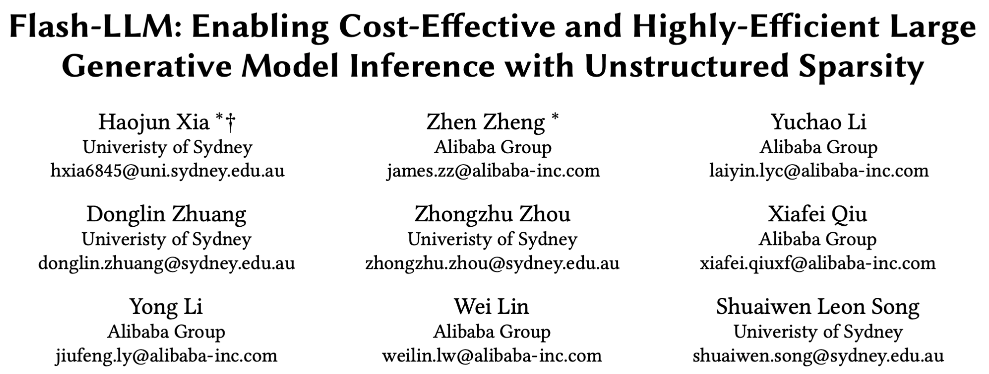
Flash-LLM: Enabling Cost-Effective and Highly-Efficient Large Generative Model Inference with Unstructured Sparsity

- Sep, 2023
- Yue Guan

Overview



Flash-LLM: Enabling Cost-Effective and Highly-Efficient Large Generative Model Inference with Unstructured Sparsity

Haojun Xia *†
Univeristy of Sydney
hxia6845@uni.sydney.edu.au

Zhen Zheng *
Alibaba Group
james.zz@alibaba-inc.com

Yuchao Li
Alibaba Group
laiyin.lyc@alibaba-inc.com

Donglin Zhuang
Univeristy of Sydney
donglin.zhuang@sydney.edu.au

Zhongzhu Zhou
Univeristy of Sydney
zhongzhu.zhou@sydney.edu.au

Xiafei Qiu
Alibaba Group
xiafei.qiuxf@alibaba-inc.com

Yong Li
Alibaba Group
jiufeng.ly@alibaba-inc.com

Wei Lin
Alibaba Group
weilin.lw@alibaba-inc.com

Shuaiwen Leon Song
Univeristy of Sydney
shuaiwen.song@sydney.edu.au

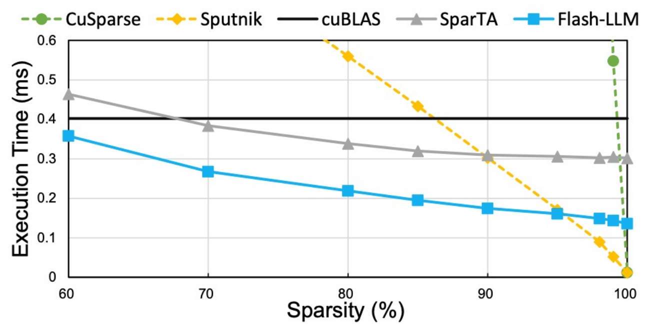
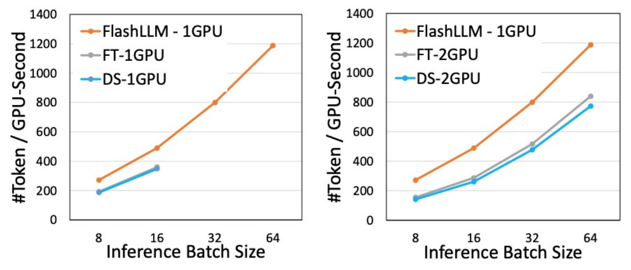


Figure 3: Performance of an unstructured SpMM (M/K/N =hidden_size*4/hidden_size/batch_size=36K/9K/8) under different designs on GPU. SIMT core centric designs are indicated with dash lines while tensor core centric designs are indicated with solid lines (including our solution Flash-LLM).



(a) Single GPU solutions. **(b) Compared to 2-GPU solutions.**

Figure 13: OPT-30B Inference Throughput.

- Sep. 17
- VLDB24
- Implementation: wrapping FasterTransformer, open sourced.

Contents

- Motivation: skinny GEMM is memory bound in LLM inference.
- Insight: load as sparse and compute as dense.
- Contributions:
  - Kernel implementation
    - Tiled GEMM with sparse decoding and TC.
    - Pipelining and double buffer.
    - Shared memory bank conflict with reordering.
  - Sparse format: Tiled-CSL
- Evaluation.
- Summary and discussion

Background

Transformer computation

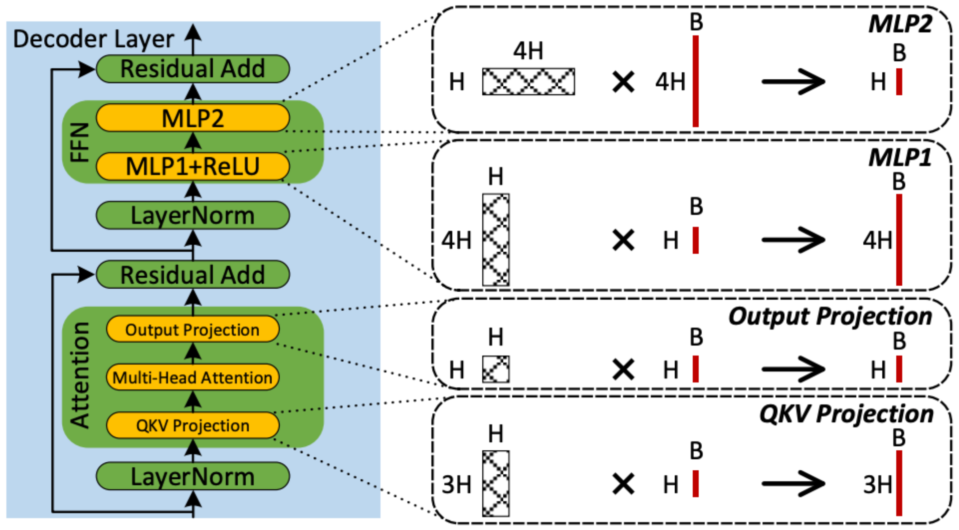
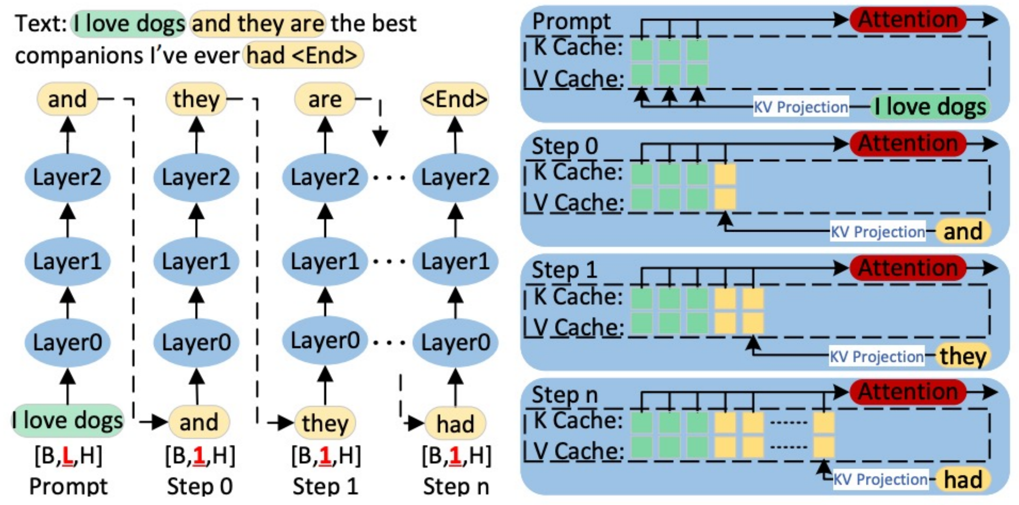


Figure 2: Decoder Layer Architecture. The H here means the hidden dimension aka. model dimension, which equals 12K for GPT-3. The B refers to the inference batch size which is typically small for real-time inference, e.g. 8, 16 or 32.

LLM Inference



(a) Auto-regressive Token Generation (b) KV Cache within Layer-0

Figure 1: (a) Generative model inference; (b) KV-Cache.

Motivation
Skinny GEMM is memory bound in LLM inference.

- The generation phase of LLM inference is bounded on memory access.
  - 70% is GEMM.
    - Batch size in generation phase is low.
    - Usually, 1~16 level.
  - Tensor Core utilization is low.

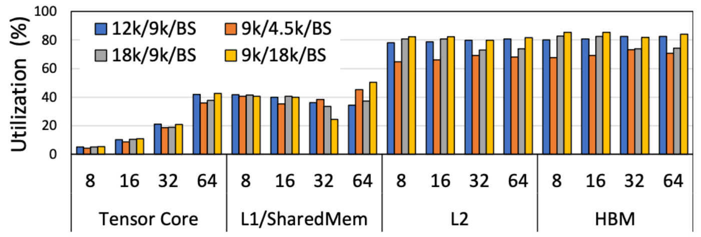


Figure 4: GPU utilization Breakdown. The MatMuls profiled in this figure are the most time-consuming parts during OPT-66B inference (with 2 GPUs) at batch sizes 16, 32, 64, and 128.

- Theoretical analysis with computation intensity
  - Compute: 2MNK FLOPs
  - Memory: 2MK + 2NK
  - Intensity: $CI = \frac{M \times N}{M + N}$

Insight
Load as sparse and compute as dense

- Load as sparse and compute as dense.
  - Unstructured sparsity is accuracy friendly.
  - Hardware (Tensor Core) supports very efficient dense GEMM.
- Computation intensity
  - Compute: 2MNK
  - Memory: 2(1-b)MK + NK (+ Sparse Index)
  - Intensity:

$$CI_{SparseLoad}{}^{4} = \frac{M \times N}{M \times (1-\beta) + N}$$

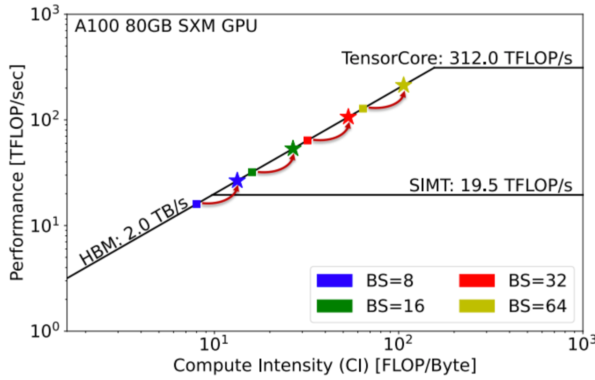
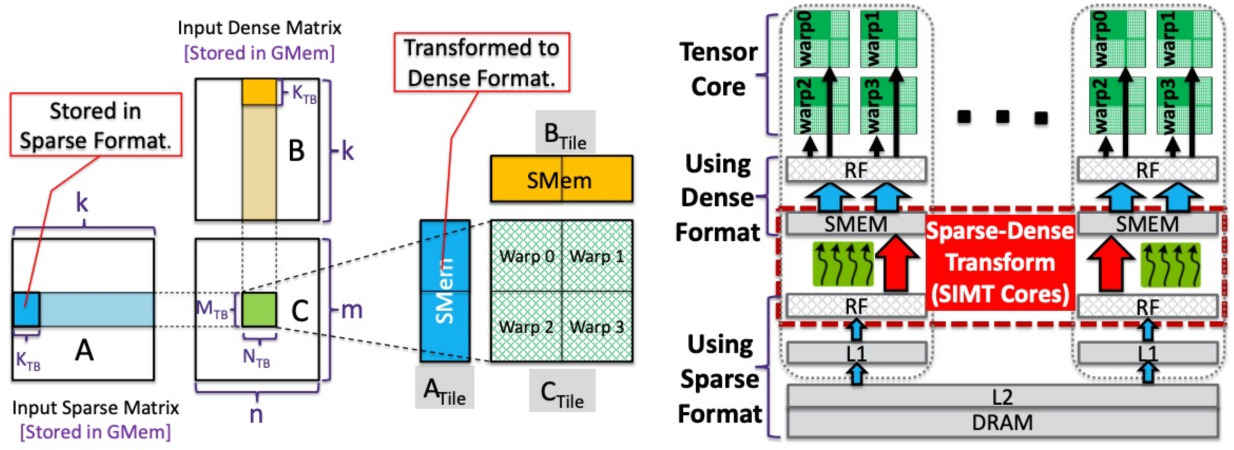


Figure 5: Roofline model for skinny MatMuls. The solid Squares refer to the CI and the performance upper bound for dense solutions (e.g. cuBLAS), while the solid Stars represent the improved CI and the new performance bound with our *Load-as-Sparse Compute-as-Dense*. Note that the vertical axis is displayed on a logarithmic scale.

Contents

- Motivation: skinny GEMM is memory bound in LLM inference.
- Insight: load as sparse and compute as dense.
- Contributions:
  - Kernel implementation
    - Tiled GEMM with sparse decoding and TC.
    - Pipelining and double buffer.
    - Shared memory bank conflict with reordering.
  - Sparse format: Tiled-CSL
- Evaluation.
- Summary and discussion

Tiled GEMM with sparse decoding and TC

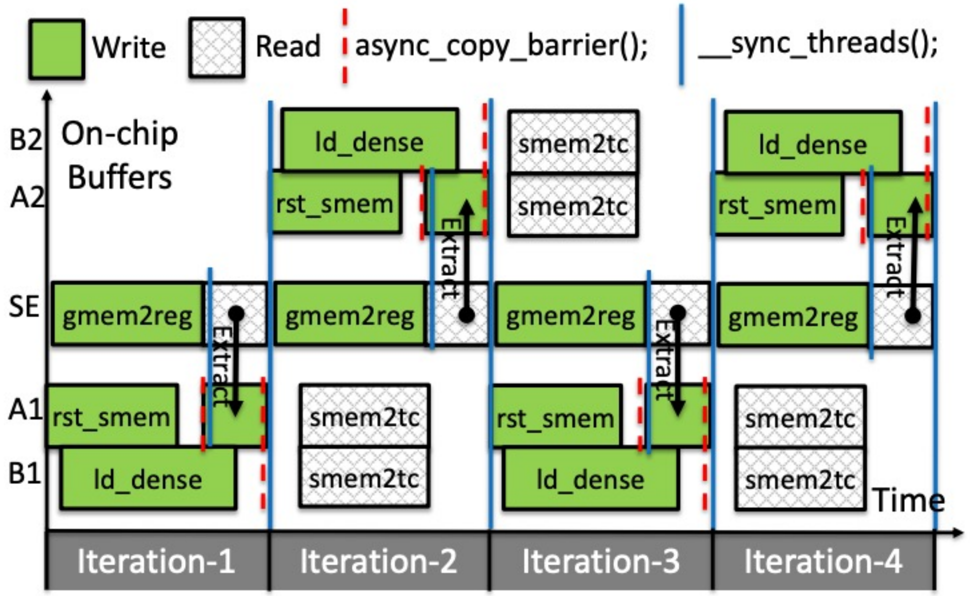


(a) Load-as-Sparse, Compute-as-Dense. (b) Sparse-to-Dense Transformation.

- Load A tile
- *gmem2reg*: loading sparse encoding from global memory to the distributed registers
- *rst_smem*: resetting the target shared memory buffer with zero
- *extract*: writing the sparse encoding from registers to the corresponding locations on shared memory buffer
- Load B tile
- *ld_dense*: loading directly from global memory to the target shared memory buffer
- Compute C tile
- *smem2tc*: loading the shared memory data of A tile and B tile, and executes tensor core computations

Pipelining and double buffer

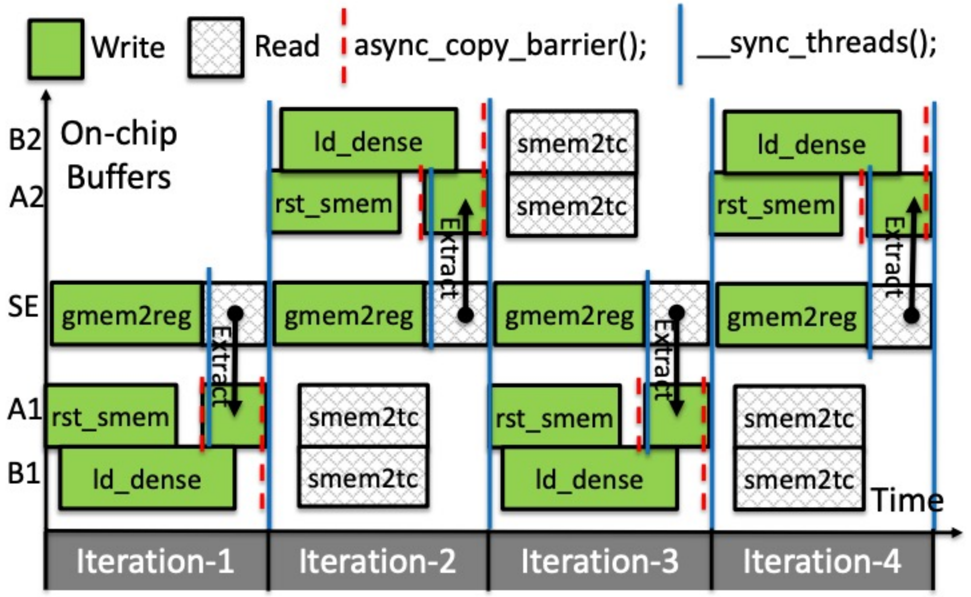
- Piplining
  - load A, extract A, load B, compute C
- Double buffer
  - async load A, async load B
- Async load barrier
  - Reset to 0 before extract
  - Finish load before compute
- Sync thread
  - Load A values before extract
  - Finish all before next iteration



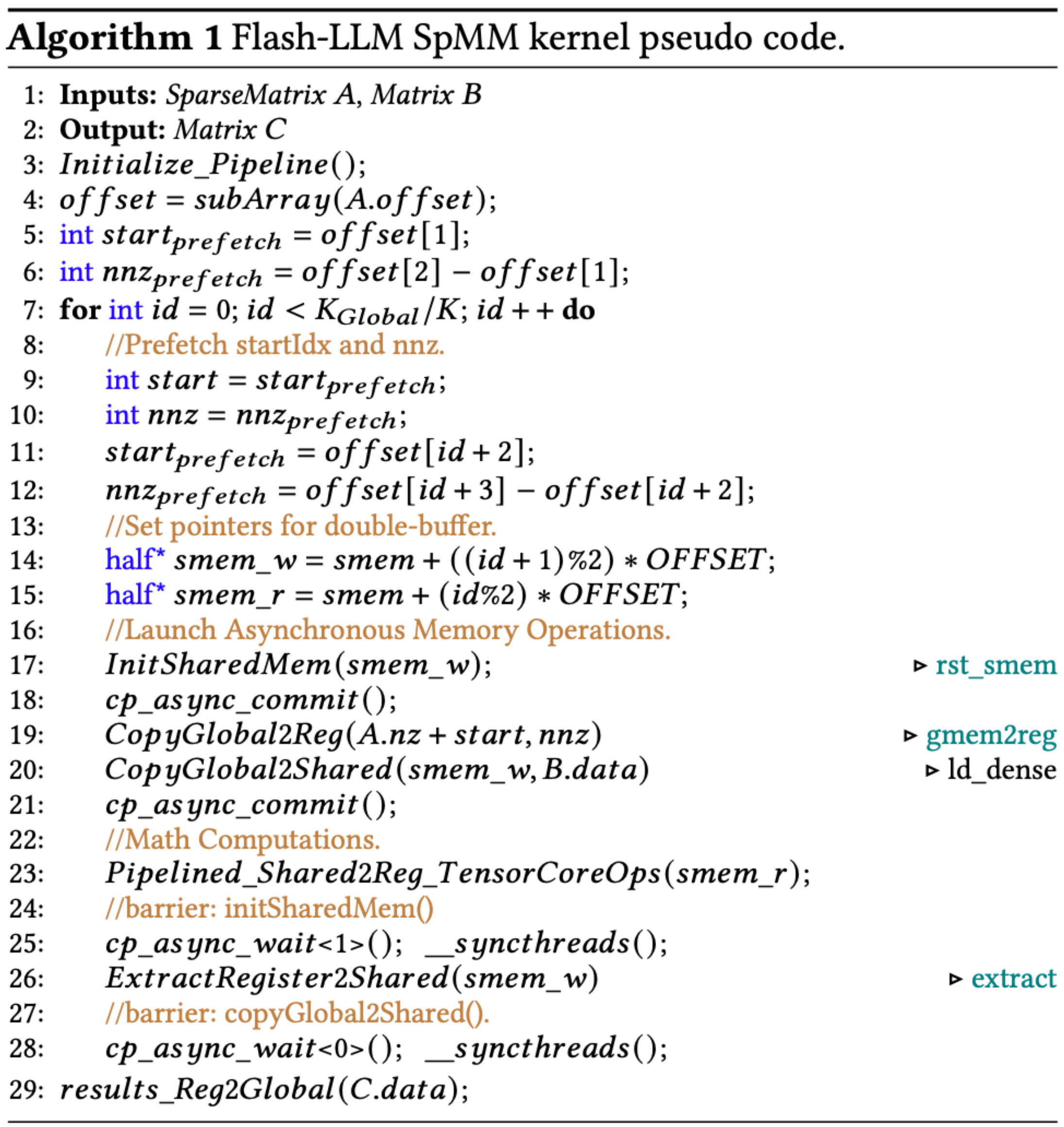
(c) Pipelined memory and tensor core operations.

Kernel Implementation

- Wrapping FasterTransformer



(c) Pipelined memory and tensor core operations.



Algorithm 1 Flash-LLM SpMM kernel pseudo code.

```
Inputs: SparseMatrix A, Matrix B
Output: Matrix C
$Initialize\_Pipeline()$;
$offset = subArray(A.offset)$;
int $start_{prefetch} = offset[1]$;
int $nnz_{prefetch} = offset[2] - offset[1]$;
for int $id = 0$; $id < K_{Global}/K$; $id++$ do
    //Prefetch startIdx and nnz.
    int $start = start_{prefetch}$;
    int $nnz = nnz_{prefetch}$;
    $start_{prefetch} = offset[id + 2]$;
    $nnz_{prefetch} = offset[id + 3] - offset[id + 2]$;
    //Set pointers for double-buffer.
    half* $smem\_w = smem + ((id + 1)\%2) * OFFSET$;
    half* $smem\_r = smem + (id\%2) * OFFSET$;
    //Launch Asynchronous Memory Operations.
    $InitSharedMem(smem\_w)$;                              ▷ rst_smem
    $cp\_async\_commit()$;
    $CopyGlobal2Reg(A.nz + start, nnz)$                   ▷ gmem2reg
    $CopyGlobal2Shared(smem\_w, B.data)$                  ▷ ld_dense
    $cp\_async\_commit()$;
    //Math Computations.
    $Pipelined\_Shared2Reg\_TensorCoreOps(smem\_r)$;
    //barrier: initSharedMem()
    $cp\_async\_wait<1>()$; $\_\_syncthreads()$;
    $ExtractRegister2Shared(smem\_w)$                     ▷ extract
    //barrier: copyGlobal2Shared().
    $cp\_async\_wait<0>()$; $\_\_syncthreads()$;
$results\_Reg2Global(C.data)$;
```

Sparse format: Tiled-CSL

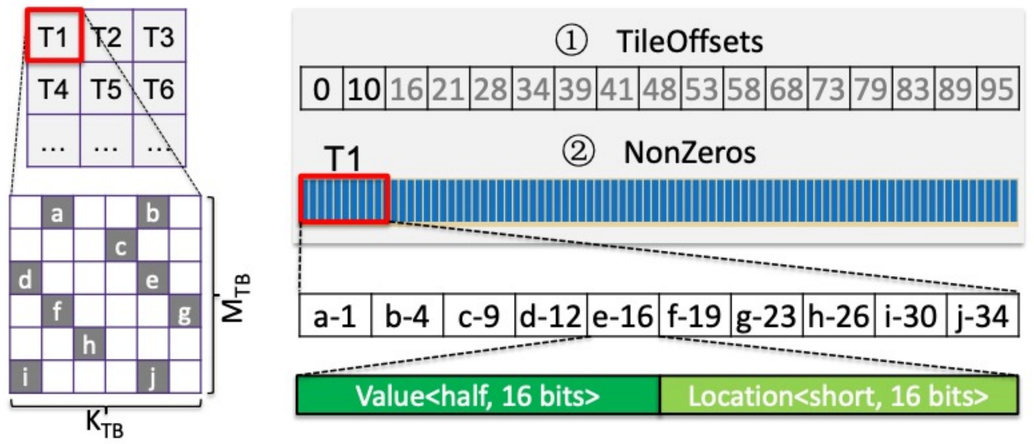
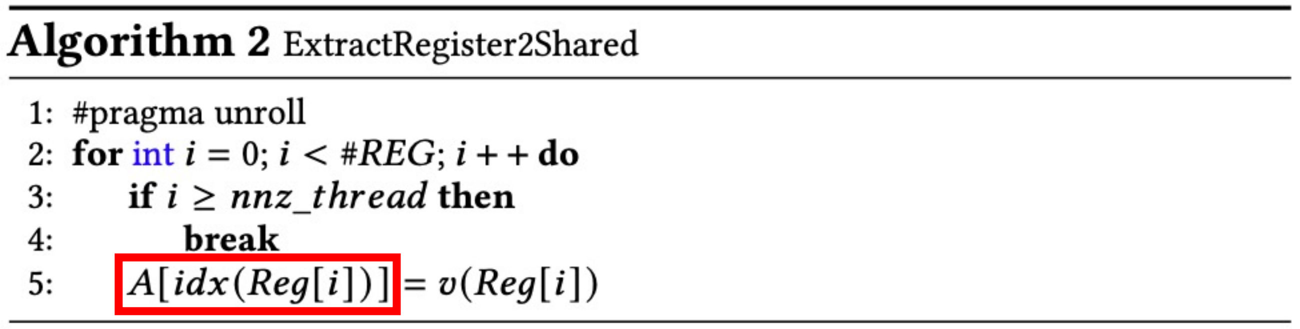


Figure 7: *Tiled-CSL* Format for sparse matrices.

- SIMT decoding with CUDA core.



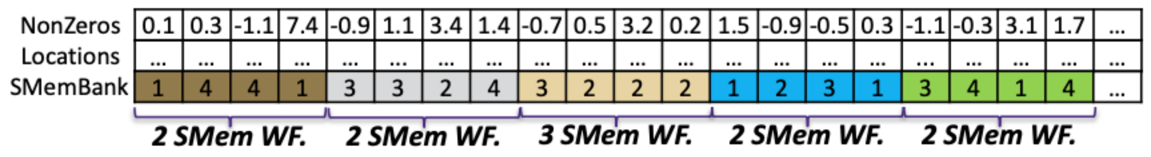
Algorithm 2 ExtractRegister2Shared

```
1: #pragma unroll
2: for int $i = 0; i < \#REG; i++$ do
3:     if $i \geq nnz\_thread$ then
4:         break
5:     $\boxed{A[idx(Reg[i])]} = v(Reg[i])$
```

Inefficient random access: reg->shared, bank conflict

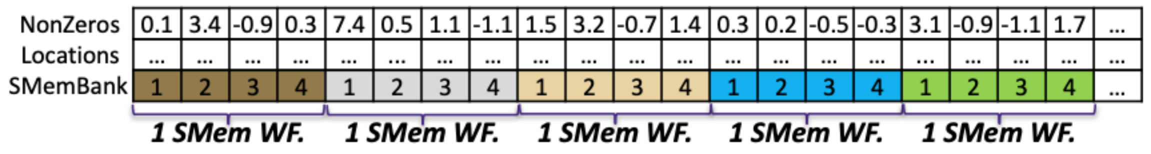
Shared memory bank conflict with reordering

- Reorder sparse A in Tile-CSL format in advance.



NonZeros	0.1	0.3	-1.1	7.4	-0.9	1.1	3.4	1.4	-0.7	0.5	3.2	0.2	1.5	-0.9	-0.5	0.3	-1.1	-0.3	3.1	1.7	...
Locations	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
SMemBank	1	4	4	1	3	3	2	4	3	2	2	2	1	2	3	1	3	4	1	4	...

(b) Bank conflicts during *ExtractRegister2Shared*. (WARP size and SMem banks are reduced from 32 to 4 for simplicity.)



NonZeros	0.1	3.4	-0.9	0.3	7.4	0.5	1.1	-1.1	1.5	3.2	-0.7	1.4	0.3	0.2	-0.5	-0.3	3.1	-0.9	-1.1	1.7	...
Locations	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
SMemBank	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	...

(c) No conflict after Sparse Data Reorder.

Figure 8: Ahead of time sparse data reordering.

Contents

- Motivation: skinny GEMM is memory bound in LLM inference.
- Insight: load as sparse and compute as dense.
- Contributions:
  - Kernel implementation
    - Tiled GEMM with sparse decoding and TC.
    - Pipelining and double buffer.
    - Shared memory bank conflict with reordering.
  - Sparse format: Tiled-CSL
- Evaluation.
- Summary and discussion

Evaluation
Main results

- cuSparse, Sputnik: SpMM (unstructured sparsity) library
- cuBLAS: dense GEMM library
- SparTA: converting SpMM to dense + unstructured

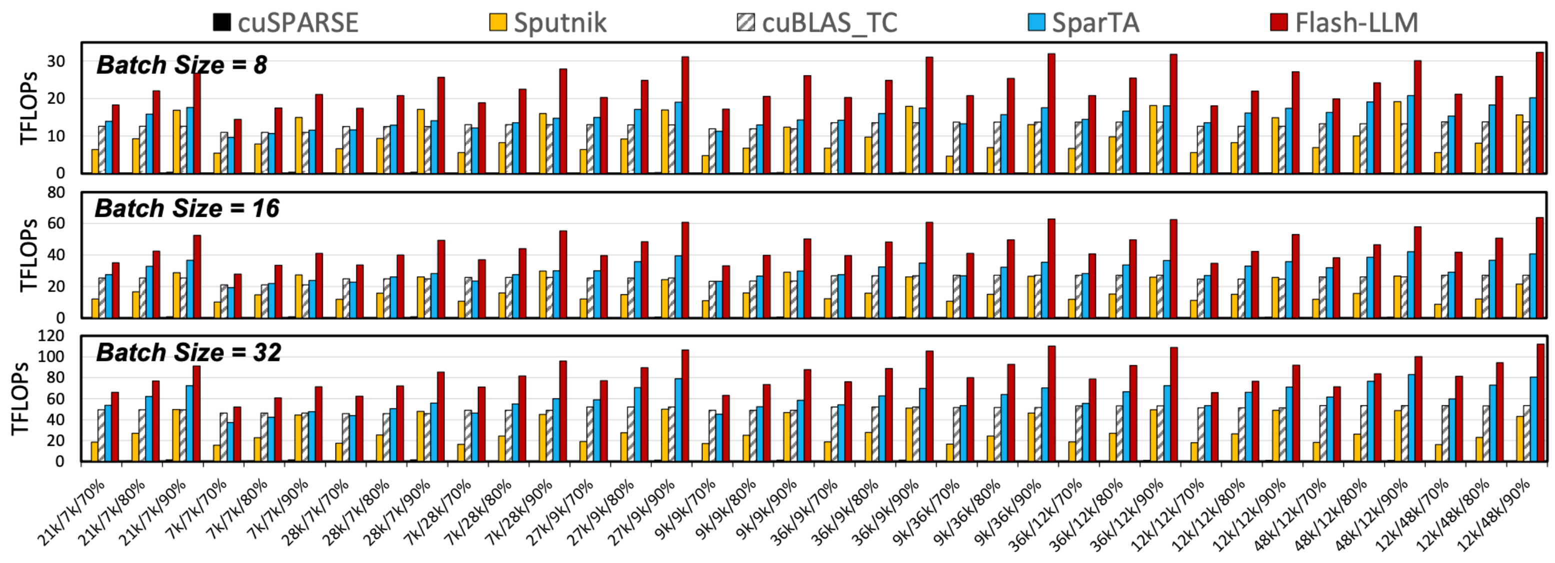
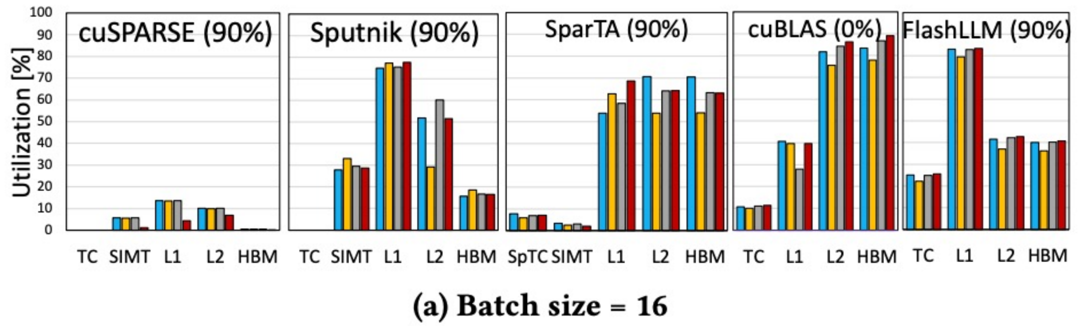
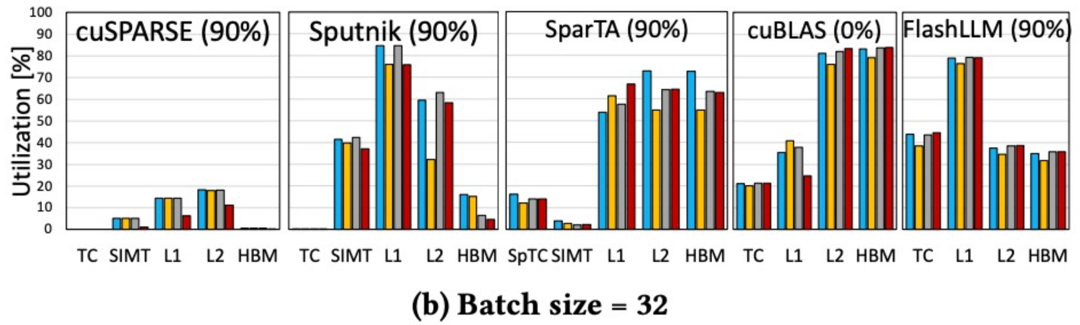


Figure 9: Kernel Benchmarking (M/K/Sparsity; weight matrix: M × K).

Evaluation
Hardware utilization



(a) Batch size = 16



(b) Batch size = 32

Figure 10: Kernel utilization breakdown with four MatMul shapes (indicated with different colors) from OPT-66B.

Evaluation Sensitivity

Latency breakdown

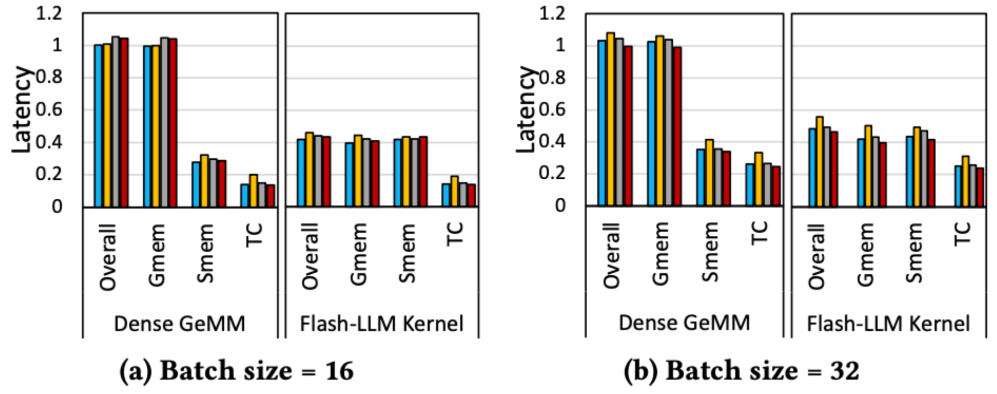


Figure 11: Latency breakdown of Dense and Flash-LLM Kernels (normalized to cuBLAS[39] kernel latency).

Batch size (skiny)

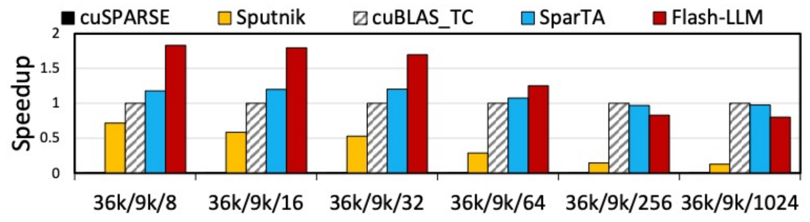
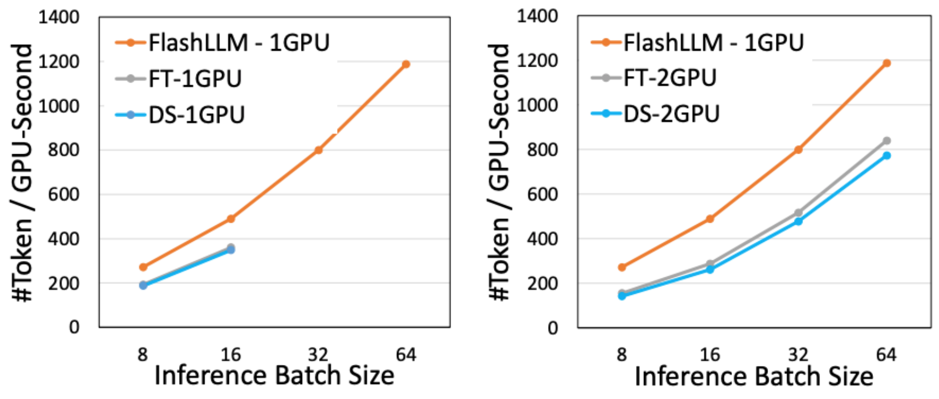


Figure 12: Kernel speedups over cuBLAS [39] GeMM kernel with different shapes (M/K/N, sparsity=80%).

Evaluation
End to end

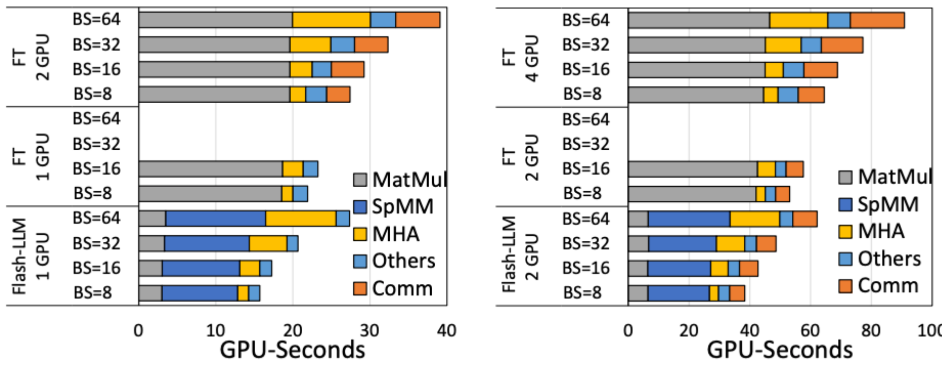
E2E throughput



(a) Single GPU solutions. (b) Compared to 2-GPU solutions.

Figure 13: OPT-30B Inference Throughput.

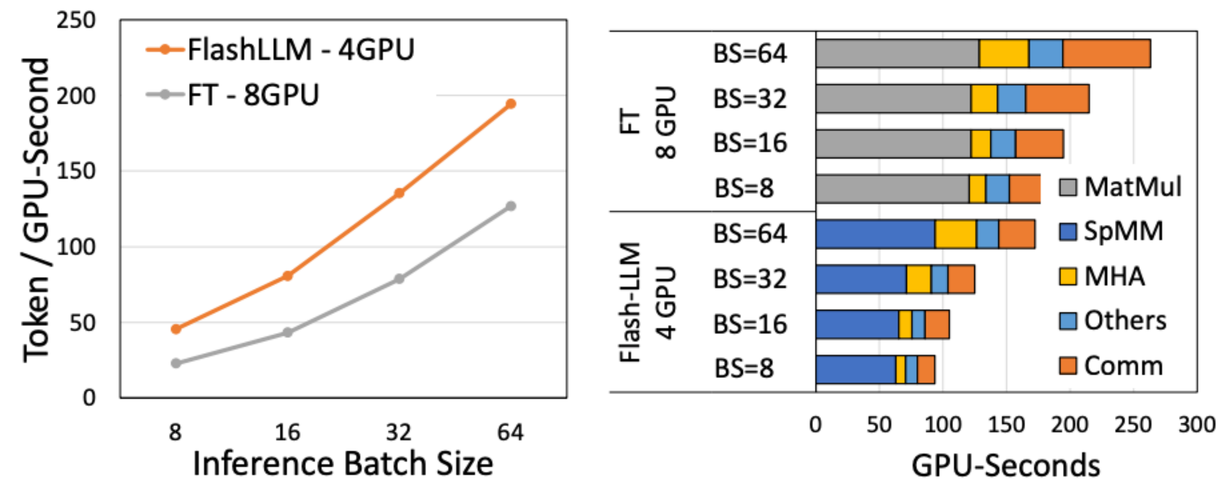
E2E breakdown



(a) OPT-30B. (b) OPT-66B.

Figure 14: Inference Time Breakdown. (MHA: multi-head attention, Comm: cross-GPU communications)

Evaluation
One node multiple devices



(a) Token Generation Throughput. (b) Breakdown.

Figure 16: OPT-175B Inference

Contents

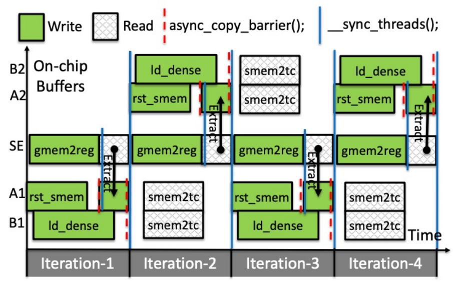
- Motivation: skinny GEMM is memory bound in LLM inference.
- Insight: load as sparse and compute as dense.
- Contributions:
  - Kernel implementation
    - Tiled GEMM with sparse decoding and TC.
    - Pipelining and double buffer.
    - Shared memory bank conflict with reordering.
  - Sparse format: Tiled-CSL
- Evaluation.
- Summary and discussion

Discussion

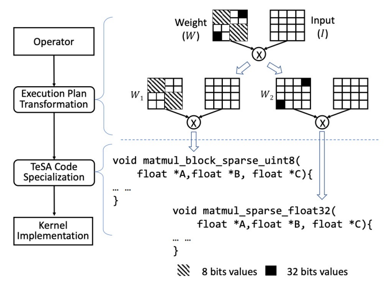
- High level computation intensity is relevant to M size.
- But computation of each tile is always the same.
- Why skinny MM computation intensity is low?

Discussion
Unstructured sparsity on structured hardware

- Flash-LLM
  - Structured sparsity is too lossy.
  - Still want to use structured hardware (TC/STC).
  - Only use sparsity to optimize memory access.
- SparTA
  - Structured sparsity is too lossy.
  - Still want to use structured hardware (TC/STC).
  - Compile unstructured to structured.
- AdaPrune
  - Structured sparsity is too lossy.
  - Still want to use structured hardware (TC/STC).
  - Transform unstructured to structured.

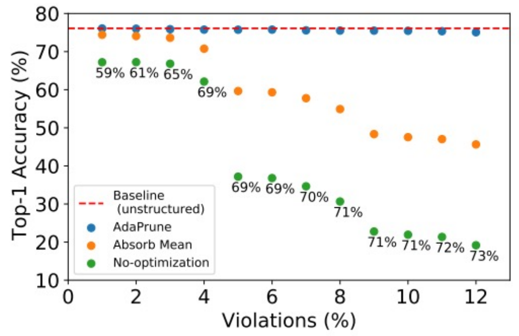


(c) Pipelined memory and tensor core operations.



```
void matmul_block_sparse_uint8(
    float *A,float *B, float *C){
… …
}
        void matmul_sparse_float32(
            float *A,float *B, float *C){
        … …
        }
```

Figure 5: Two-pass compilation to generate an efficient kernel for an operator (MatMul).



Discussion
Unstructured sparsity on structured hardware

- TW/Magicube/…
  - Structured sparsity is too lossy.
  - Still want to use structured hardware (TC/STC).
  - Design semi-structured patterns.
- Others
  - Make structured sparsity less lossy.
  - Make unstructured SpMM faster.
    - Compiler: Scheduled-TACO, Sparse Abstract Machine, …
    - Accelerator: Eyeriss-v2, Fractal-TC, …

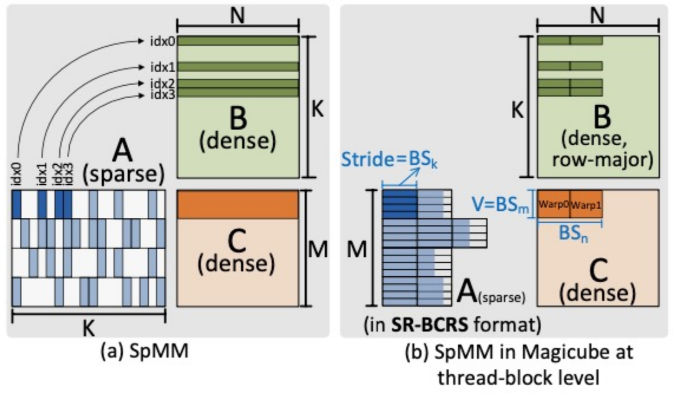


Fig. 3. SpMM and its thread-block view in Magicube.

Discussion
Why dense TC

- Sparse TC is inefficient.
  - High encoding overhead cannot be hide by computation?
  - If so, will discuss in paper.
- Sparse TC is hard to implement.
  - No sufficient PTX level supported.

- Maybe we can try sparse TC.

Thank you!!!

- Motivation: skinny GEMM is memory bound in LLM inference.
- Insight: load as sparse and compute as dense.
- Contributions:
    - Kernel implementation
        - Tiled GEMM with sparse decoding and TC.
        - Pipelining and double buffer.
        - Shared memory bank conflict with reordering.
    - Sparse format: Tiled-CSL
- Evaluation.
- Summary and discussion