SMASH: Co-designing Software Compression and Hardware-Accelerated Indexing for Efficient Sparse Matrix Operations

Konstantinos Kanellopoulos  Nandita Vijaykumar  Christina Giannoula  Roknoddin Azizi  Skanda Koppula  Nika Mansouri Ghiasi  Taha Shahroodi  Juan Gomez Luna  Onur Mutlu

MICRO-52, 2019

Presented by Yannick Schmid
Executive Summary

**Motivation:** Sparse matrix operations enable a wide variety of workloads.
Executive Summary

**Motivation:** Sparse matrix operations enable a wide variety of workloads.

**Problem:** Compressed formats require additional memory accesses to read the indices of non-zeroes.
Executive Summary

**Motivation:** Sparse matrix operations enable a wide variety of workloads.

**Problem:** Compressed formats require additional memory accesses to read the indices of non-zeroes.

**Goal:** New format that reduces memory accesses from indexing and enables H/W support.
Executive Summary

**Motivation:** Sparse matrix operations enable a wide variety of workloads.

**Problem:** Compressed formats require additional memory accesses to read the indices of non-zeroes.

**Goal:** New format that reduces memory accesses from indexing and enables H/W support.

**Key Idea:** Compress indices as hierarchy of bitmaps. Light-weight H/W unit enables efficient decompression.
Executive Summary

**Motivation:** Sparse matrix operations enable a wide variety of workloads.

**Problem:** Compressed formats require additional memory accesses to read the indices of non-zeroes.

**Goal:** New format that reduces memory accesses from indexing and enables H/W support.

**Key Idea:** Compress indices as hierarchy of bitmaps. Light-weight H/W unit enables efficient decompression.

**Evaluation:** New format benefits from H/W support and beats all baselines.
Many Problems are only feasible for Sparse Matrices

Social Network

Video Platform

Users (billions)

Friends Matrix

View Matrix

Users (billions)

Videos (millions)
Compressed Sparse Row/Column

A single row/column can be compressed as follows:

1. Store only the non-zero values in an array
2. Second array stores the position of each non-zero within the row/column

To store an entire matrix:

1. Compress each row/column
2. Concatenate the resulting arrays (values & indices)
3. Point to the beginning/end of each row/column

Matrix can be processed in compressed format.
Compressed Sparse Row/Column
Compressed Sparse Row/Column
Inner Product of Compressed Row & Column

Compressed Row
(Left Matrix)

Compressed Col.
(Right Matrix)
Inner Product of Compressed Row & Column

Compressed Row
(Left Matrix)

Compressed Col.
(Right Matrix)
### Inner Product of Compressed Row & Column

**Compressed Row**  
(Left Matrix)  

| 4 | 10 | 42 | 50 | 64 | 81 | 90 |

**Compressed Col.**  
(Right Matrix)  

| 8 | 10 | 12 | 81 | 91 |
Inner Product of Compressed Row & Column

Compressed Row
(Left Matrix)

4 10 42 50 64 81 90

Compressed Col.
(Right Matrix)

8 10 12 81 91
Inner Product of Compressed Row & Column

Compressed Row
(Left Matrix)

Compressed Col.
(Right Matrix)
Inner Product of Compressed Row & Column

Compressed Row
(Left Matrix)

Indexing accounts for most memory accesses!

Compressed Col.
(Right Matrix)
Evaluating the Indexing Overhead

Indexing represents a significant fraction of the computational costs when working with sparse matrices:

- Additional memory accesses
- Pointer arithmetic

![Graph comparing CSR and CSR with zero-cost indexing for SpMatAdd, SpMV, and SpMM operations.](image-url)
Outline

- Background
- SMASH
- Evaluation
- Strengths & Weaknesses
- Discussion
SMASH: Sparse Matrix Software/Hardware

New Matrix Format

- Represent indices as bitmap
- Compress with a tree-based approach
- Extends typical sparse matrix compression schemes by providing separate compression just for indices

H/W Support for new Format

- Decompresses the compressed indices
- Handles memory accesses for indexing
- Controlled with added instructions
- Does not do any arithmetic on matrix elements
SMASH: Sparse Matrix Software/Hardware

New Matrix Format

Hierarchy of Bitmaps

H/W Support for new Format

Indexing

BMU

Arithmetic
Hierarchy of Bitmaps
Hierarchy of Bitmaps
Hierarchy of Bitmaps
Hierarchy of Bitmaps: Compression ratios
Hierarchy of Bitmaps: Compression ratios

- Compression ratios determine the block sizes
- During indexing, step through each level block by block
- Compression ratios must be chosen to fit the distribution of non-zeros
- The last compression ratio determines the block size of the matrix elements. A larger block size for the matrix elements benefits from clustering of non-zeros: Optimal compression is achieved when many non-zeros follow each other.
- The last compression ratio is used to recover the indices of the elements from the indices of the block
Hierarchy of Bitmaps: Storage Format

Storage Format

bitmap-1: 1 0 1
bitmap-0: 0 1 1 1
NZA
Hierarchy of Bitmaps: Indexing

Index of Block 0

$4^3 \times 0$
Branch 0

$4^2 \times 1$
Branch 1

$4^1 \times 1$
Branch 1

$4^0 \times 0$
Branch 0
Hierarchy of Bitmaps: Indexing

- 4^3 x 0
- 4^2 x 1
- 4^1 x 1
- 4^0 x 3

Index of Block 1

Branch 0

Branch 1

Branch 3

29
Hierarchy of Bitmaps: Indexing

- $4^0 \times 1$
  - Branch 0
  - Branch 2
  - Branch 1

Index of Block 2
H/W support: Bitmap Management Unit (BMU)

<table>
<thead>
<tr>
<th>BMU</th>
<th>Hardware Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitmap-3</td>
<td>matrix-dims</td>
</tr>
<tr>
<td>bitmap-2</td>
<td>bmap-parameters</td>
</tr>
<tr>
<td>bitmap-1</td>
<td>row-index</td>
</tr>
<tr>
<td>bitmap-0</td>
<td>col-index</td>
</tr>
</tbody>
</table>

Store current block of each level
BMU ISA: Configuration

matinfo row,col,grp

BMU

bitmap-3
bitmap-2
bitmap-1
bitmap-0

ISA

Hardware
Logic

matrix-dims
bmap-parameters
row-index
col-index

Configure
BMU ISA: Configuration

bmapinfo comp,lvl,grp

BMU

- bitmap-3
- bitmap-2
- bitmap-1
- bitmap-0

Hardware Logic

- matrix-dims
- bmap-parameters

- row-index
- col-index

Configure
BMU ISA: Load initial Bitmap blocks

rdmap [mem],buf,grp

BMU

bitmap-3
bitmap-2
bitmap-1
bitmap-0

ISA

Hardware Logic

matrix-dims
bmap-parameters
row-index
col-index

Access Memory
BMU ISA: Load initial Bitmap blocks

bitmap-0
bitmap-1
bitmap-2
bitmap-3
BMU ISA: Find next Non-Zero

Access Memory

BMU
- bitmap-3
- bitmap-2
- bitmap-1
- bitmap-0

ISA
- matrix-dims
- bmap-parameters
- row-index
- col-index

Hardware Logic

pbmap grp
BMU ISA: Read out Indices

rdind rd1, rd2, grp

BMU

| bitmap-3 |
| bitmap-2 |
| bitmap-1 |
| bitmap-0 |

ISA

Hardware Logic

- matrix-dims
- bmap-parameters
- row-index
- col-index

Read out
H/W support for multiple matrices
Evaluation: Software-only SMASH

Methodology:
- Evaluated on a real system
- 15 different sparse matrices

Takeaway:
Reading the hierarchy of bitmaps in software is expensive.
Evaluation: SMASH with H/W support

Methodology:
- Evaluated using ZSim [3] simulator
- 15 different sparse matrices
- Baseline: TACO [2]
- Similar results for Graph algorithms

Takeaway: SMASH benefits from H/W support and beats all baselines.
Estimation of H/W area overhead

H/W cost of example configuration: <0.1% of modern server CPU

Takeaway: SMASH incurs negligible H/W area overhead.
Strengths

New Compression Format
1. Specifically targets indexing by compressing indices to reduce memory accesses.
2. Compressed indices can still be read efficiently with H/W support.

Workload Analysis
1. Intuitive reasoning shows that indexing can account for significant amount of computation.
2. Experimental study validates this for a set of relevant operations.

Hardware Support
1. Light-weight design.
2. Flexible interface with new instructions.
Weaknesses

1. No **bandwidth** sensitivity analysis

2. Different matrix formats are used to suit different applications. Tying the format to H/W removes this **flexibility**.

3. Paper conflates **blocking** with its concept of **compression ratios**.
Weakness: Blocking & Compression Ratios

This can be:
1. scalar value
2. sub-matrix (block)

Blocking amortizes indexing cost over multiple elements

Blocking strategy:
1. Block size
2. Block shape

Important for recovering indices within blocks
Weakness: Blocking & Compression Ratios

In general: Blocks are fixed to shape 1xn
Possible improvement: Support for rectangular blocks
Weakness: Blocking & Compression Ratios

Ablation Study

When comparing indexing schemes, they should use the same blocking strategy! Otherwise it’s not clear what caused the performance gains: the indexing or the blocking?

“ For the matrices with the highest sparsity [...], TACO-BCSR is inefficient because it encodes data in blocks [...]. SMASH avoids such overhead by leveraging the configurability of compression ratios in our hierarchical bitmap encoding [...].
Takeaways

- Sparse matrices can be stored and processed in a compressed format.
- Indexing represents a significant fraction of sparse matrix computation.
- Indices are stored in a separate data structure that can be compressed too.
- Further compressing the indices can reduce data movement but adds decompression overhead when reading the indices.
- H/W support can enable efficient decompression of compressed indices.
Topic I: H/W Implementation

ISA extension for CPU

ISA extension for GPU

Separate device
Topic I: H/W Implementation

- ISA extension for CPU
  - Proposal from the Paper
  - CPU is well-suited to irregular execution patterns of sparse matrix operations
  - Hard to change ISA

- ISA extension for GPU

- Separate device
Most numerical workloads are offloaded to GPU these days.
Reading the Hierarchy of Bitmaps not easy to parallelize.
Proprietary standards easier to change.
Has to be adapted to GPU programming model.
Topic I: H/W Implementation

- ISA extension for CPU
- ISA extension for GPU
- Separate device

- Fewest hurdles to implementation.
- Better suited for devices that perform entire matrix operations.
- Low latency tolerance for indexing
Topic II: Sparse Matrix operations

Which sparse matrix operations:
- Have you used? Do you know?
- Are most susceptible to indexing overheads?
- Don’t fit into the paradigm on the left?
Topic II: Sparse Matrix operations

Examples of sparse matrix use cases.

**Recommender Systems**

\[
\begin{bmatrix}
? & ? & 1 & 3 & ? & ?
3 & 1 & ? & ? & ? & ?
0 & ? & ? & 1 & 4 & 2
\end{bmatrix}
\]

*Ratings matrix:* Many entries are unknown

Sparse format used to keep track of which entries are known.
Projection to known entries during Opt.

**Image convolutions**

\[
\text{vec}(\begin{bmatrix}
\end{bmatrix}) = \text{vec}(\text{conv}(\begin{bmatrix}
\end{bmatrix}))
\]

**Graph analytics**

\[
\begin{bmatrix}
0 & 1 & 1 & 1 & 0
1 & 0 & 0 & 0 & 0
1 & 0 & 0 & 1 & 1
1 & 0 & 0 & 0 & 1
0 & 0 & 1 & 1 & 0
\end{bmatrix}
\]
**Input**
Sorted array of non-negative integers

**Goal:** Compress these!
Topic III: Compressing Indices

**Input**
Sorted array of non-negative integers

**Intermediate** (not instantiated)
Bitmap

**Output**
Hierarchy of bitmaps
Topic III: Compressing Indices

**Input**
Sorted array of non-negative integers

**Intermediate**
Deltas

**Output**
Prefix Codes of Deltas
Topic III: Compressing Indices

**Advantages**
- Less data movement for indexing
- Compression scheme can fit distribution of non-zeros (e.g. hierarchy of bitmaps, universal codes)

**Disadvantages**
- Can’t be read in random order / parallel
- Decompression overhead
- Compression overhead
Thank you to my mentors!


Backup slides
Many real-world graphs are sparse. They also have sparse adjacency matrices. For large but sparse graphs it is necessary to avoid storing 0 entries. Some graph analytics tasks can be expressed as matrix operations on the adjacency matrix.
Graph analytics: # of common neighbors

\[
\begin{pmatrix}
0 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 \\
1 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 & 0
\end{pmatrix} \quad \begin{pmatrix}
0 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 \\
1 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 & 0
\end{pmatrix} = \begin{pmatrix}
3 & 0 & 0 & 0 & 2 \\
0 & 1 & 1 & 1 & 0 \\
0 & 1 & 2 & 2 & 0 \\
0 & 1 & 2 & 2 & 0 \\
2 & 0 & 0 & 0 & 2
\end{pmatrix}
\]
Sparse Matrix Formats

Coordinate Format

**COO**
(row-major)

```
val = [1, 2, 3, 4]
row_idx = [0, 1, 1, 3]
col_idx = [3, 0, 2, 0]
```

Compressed Sparse Row

**CSR**

```
val = [1, 2, 3, 4]
row_ptr = [0, 1, 3, 3, 4]
col_idx = [3, 0, 2, 0]
```

2x2-Block Compressed Sparse Row

**BCSR**

```
val = [0,0,2,0,|0,1,3,0,|0,0,4,0]
row_ptr = [0, 2, 3]
col_idx = [0, 1, 0]
```
**Dense Matrix Multiplication**

```c
for (int i=0; i<n; i++){
    for (int j=0; j<n; j++){
        for (int k=0; k<n; k++){
            C[i*n+j] += A[i*n+k]*B[j*n+k];
        }
    }
}
```

**Sparse Matrix Multiplication**

```c
for (int i=0; i<n; i++){
    for (int j=0; j<n; j++){
        float res = 0;
        int idxA = A.row_ptr[i];
        int idxB = B.col_ptr[j];
        int row_end = A.row_ptr[i+1];
        int col_end = B.col_ptr[j+1];
        while (idxA!=row_end && idxB!=col_end){
            int kA = A.col_idx[idxA];
            int kB = B.row_idx[idxB];
            if (kA<kB) idxA++;
            else if (kB<kA) idxB++;
            else res += A.val[idxA++] * B.val[idxB++];
        }
        if (res != 0) C.insert(res, i, j);
    }
}
```
Indexing Overhead of Sparse Formats

1. Iterate to next non-zero elements
2. Discover their matrix indices
3. Compare the indices

```c
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
    {
        float res = 0;
        int idxA = A.row_ptr[i];
        int idxB = B.col_ptr[j];
        int row_end = A.row_ptr[i+1];
        int col_end = B.col_ptr[j+1];
        while (idxA!=row_end && idxB!=col_end)
        {
            int kA = A.col_idx[idxA];
            int kB = B.row_idx[idxB];
            if (kA<kB) idxA++;
            else if (kB<kA) idxB++;
            else res += A.val[idxA++] * B.val[idxB++];
        }
        if (res != 0) C.insert(res, i, j);
    }
```
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>matinfo row, col, grp</td>
<td>Set matrix dimensions</td>
</tr>
<tr>
<td>bmapinfo comp, lvl, grp</td>
<td>Set compression ratio</td>
</tr>
<tr>
<td>rdmap [mem], buf, grp</td>
<td>Load bitmap into SRAM</td>
</tr>
<tr>
<td>pbmap grp</td>
<td>Find next non-zero block</td>
</tr>
<tr>
<td>rdind rd1, rd2, grp</td>
<td>Read out the matrix indices</td>
</tr>
</tbody>
</table>