High-Performance Sparse Matrix-Matrix Products on Intel KNL and Multicore Architectures

Yusuke Nagasaka†, Satoshi Matsuoka§†
Ariful Azad‡, Aydı̈n Buluç‡

† Tokyo Institute of Technology
§ Riken Center for Computational Science
‡ Lawrence Berkeley National Laboratory
Sparse General Matrix-Matrix Multiplication (SpGEMM)

- Key kernel in graph processing and numerical applications
  - Markov clustering, Betweenness centrality, triangle counting, ...
  - Preconditioner for linear solver
    - AMG (Algebraic Multigrid) method
    - Time-consuming part

<table>
<thead>
<tr>
<th></th>
<th>Input Matrices</th>
<th>Output Matrices</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td>ah+</td>
</tr>
<tr>
<td>b</td>
<td></td>
<td>ai+b</td>
</tr>
<tr>
<td>c</td>
<td></td>
<td>bk</td>
</tr>
<tr>
<td>d</td>
<td></td>
<td>dl</td>
</tr>
<tr>
<td>e</td>
<td></td>
<td>dk</td>
</tr>
<tr>
<td>f</td>
<td></td>
<td>cj</td>
</tr>
<tr>
<td>g</td>
<td></td>
<td>eh</td>
</tr>
<tr>
<td>h</td>
<td></td>
<td>fj</td>
</tr>
<tr>
<td>i</td>
<td></td>
<td>ei</td>
</tr>
<tr>
<td>j</td>
<td></td>
<td>gm</td>
</tr>
<tr>
<td>k</td>
<td></td>
<td></td>
</tr>
<tr>
<td>l</td>
<td></td>
<td></td>
</tr>
<tr>
<td>m</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Output Matrices**
Accumulation of intermediate products
Sparse Accumulator (SPA) [Gilbert, SIAM1992]

Input Matrices

Output Matrices

Input matrices in sparse format
Accumulation of intermediate products
Sparse Accumulator (SPA) [Gilbert, SIAM1992]

Input Matrices

Output Matrices

Value
Column id

Input matrices in sparse format
Accumulation of intermediate products
Sparse Accumulator (SPA) [Gilbert, SIAM1992]

Input Matrices

Output Matrices

Input matrices in sparse format
Accumulation of intermediate products
Sparse Accumulator (SPA) [Gilbert, SIAM1992]

😊 Efficient accumulation of intermediate products: Lookup cost is $O(1)$
😊 Requires $O(\#\text{columns})$ memory by one thread

<table>
<thead>
<tr>
<th>Value</th>
<th>Column id</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>1</td>
</tr>
<tr>
<td>f</td>
<td>2</td>
</tr>
<tr>
<td>g</td>
<td>0</td>
</tr>
<tr>
<td>h</td>
<td>1</td>
</tr>
<tr>
<td>i</td>
<td>2</td>
</tr>
<tr>
<td>j</td>
<td>0</td>
</tr>
<tr>
<td>k</td>
<td>0</td>
</tr>
<tr>
<td>l</td>
<td>2</td>
</tr>
<tr>
<td>m</td>
<td>0</td>
</tr>
</tbody>
</table>

Input matrices in sparse format

<table>
<thead>
<tr>
<th>Value</th>
<th>Column id</th>
</tr>
</thead>
<tbody>
<tr>
<td>ah</td>
<td>0</td>
</tr>
<tr>
<td>bk</td>
<td>0</td>
</tr>
<tr>
<td>ai</td>
<td>0</td>
</tr>
<tr>
<td>bl</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>Bit flag</th>
<th>Value</th>
<th>SPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Existing approaches for SpGEMM

- Several sequential and parallel SpGEMM algorithms
  - Also packaged in software/libraries

<table>
<thead>
<tr>
<th>Algorithm (Library)</th>
<th>Accumulator</th>
<th>Sortedness (Input/Output)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MKL</td>
<td>-</td>
<td>Any/Select</td>
</tr>
<tr>
<td>MKL-inspector</td>
<td>-</td>
<td>Any/Unsorted</td>
</tr>
<tr>
<td>KokkosKernels</td>
<td>HashMap</td>
<td>Any/Unsorted</td>
</tr>
<tr>
<td>Heap</td>
<td>Heap</td>
<td>Sorted/Sorte</td>
</tr>
<tr>
<td>Hash</td>
<td>Hash Table</td>
<td>Any/Select</td>
</tr>
</tbody>
</table>
Existing approaches for SpGEMM

- Several sequential and parallel SpGEMM algorithms
  - Also packaged in software/libraries

Questions?

(a) What is the best algorithm/implementation for a problem at hand?

(b) What is the best algorithm/implementation for the architecture to be used in solving the problem?
Contribution

- We characterize, optimize and evaluate existing SpGEMM algorithms for real-world applications on modern Multi-core and Many-core architectures
  - Characterizing the performance of SpGEMM on shared-memory platforms
    - Intel Haswell and Intel KNL architectures
    - **Identify bottlenecks and mitigate them**
  - Evaluation including several use cases
    - $A^2$, Square x Tall-skinny, L*U for triangle counting
  - Showing the **impact of keeping unsorted output**
  - **A recipe for selecting the best-performing algorithm for a specific application scenario**
Benchmark for SpGEMM

Thread scheduling cost

- Evaluates the scheduling cost on Haswell and KNL architectures
  - OpenMP: static, dynamic and guided

- Scheduling cost hurts the SpGEMM performance
Benchmark for SpGEMM
Memory allocation/deallocation cost

- Identifies that allocation/deallocation of large memory space is expensive
- Parallel memory allocation scheme
  - Each thread independently allocates/deallocates memory and accesses only its own memory space
  - **For SpGEMMM, we can reduce deallocation cost**

```
Parallel memory allocation
1   eachN ← N/nthreads
2   ALLOCATE(a, nthreads)
3   for tid ← to nthreads in parallel
4       do ALLOCATE(a[tid], eachN)
5       do for i ← to eachN
6           do a[tid][i] ← i
7       do DEALLOCATE(a[tid], eachN)
8   DEALLOCATE(a[t짐])
```
Benchmark for SpGEMM
Impact of MCDRAM

- MCDRAM provides high memory bandwidth
  - Obviously improves stream benchmark
  - Performance of stanza-like memory access is unclear
    - Small blocks of consecutive elements
    - Access to rows of B in SpGEMM

Hard to get the benefits of MCDRAM on very sparse matrices in SpGEMM
**Architecture Specific Optimization**

**Thread scheduling**

- **Good load-balance with static scheduling**
  - Assigning work to threads by FLOP
  - Work assignment can be efficiently executed in parallel
    - Counting required FLOP of each row
    - PrefixSum to get total FLOP of SpGEMM
    - Assigning rows to thread (Eg. shows the case of 3 threads)
      - Average FLOP = 11/3
Architecture Specific Optimization
Accumulator for Symbolic and Numeric Phases

■ Optimizing algorithms for Intel architectures

■ Heap [Azad, 2016]
  - Priority queue indexed by column indices
  - Requires logarithmic time to extract elements
  - **Space efficient**: $O(\text{nnz}(a_{i*}))$
    - Better cache utilization

■ Hash [Nagasaka, 2016]
  - Uses hash table for accumulator, based on GPU work
    - **Low memory usage and high performance**
  - Each thread once allocates the hash table and reuses it
  - **Extended to HashVector to exploit wide vector register**
Architecture Specific Optimization

**HashVector**

- **Utilizing 256 and 512-bit wide vector register of Intel architectures for hash probing**
  - **Reduces the number of probing caused by hash collision**
  - Requires a few more instructions for each check
- **Degrades the performance when the collisions in Hash are rare**

![Diagram](image)

(a) Hash
1) Check the entry
2) If hash is collided, check next entry
3) If the entry is empty, add the element

(b) HashVector
1) Check multiple entries with vector register
2) If the element is not found and the row has empty entry, add the element

- element to be added
- non-empty entry
- empty entry
Performance Evaluation
Matrix Data

■ Synthetic matrix
  - R-MAT, the recursive matrix generator
  - Two different non-zero patterns of synthetic matrices
    ■ ER: Erdős–Rényi random graphs
    ■ G500: Graphs with power-law degree distributions
      - Used for Graph500 benchmark
  - Scale \( n \) matrix: \( 2^n \)-by-\( 2^n \)
  - *Edge factor*: the average number of non-zero elements per row of the matrix

■ SuiteSparse Matrix Collection
  - 26 sparse matrices used in several past work
Evaluation Environment

- **Cori system @NERSC**
  - **Haswell Cluster**
    - Intel Xeon Processor E5-2698 v3
    - 128GB DDR4 memory
  - **KNL Cluster**
    - Intel Xeon Phi Processor 7250
      - 68 cores
      - 32KB/core L1 cache, 1MB/tile L2 cache
      - 16GB MCDRAM
      - Quadrant, cache
    - 96GB DDR4 memory
  - **OS:** SuSE Linux Enterprise Server 12 SP3
  - **Intel C++ Compiler (icpc) ver18.0.0**
    - -g -03 -qopenmp
Benefit of Performance Optimization
Scheduling and memory allocation

- **Good load balance** with static scheduling
- For larger matrices, parallel memory allocation scheme keeps high performance

\[ A^2 \text{ of G500 matrices with edge factor}=16 \]
Benefit of Performance Optimization
Use of MCDRAM

- Benefit of MCDRAM especially on denser matrices

![Graph showing the benefit of MCDRAM with different edge factors.](image)
Performance Evaluation

A^2: Scaling with density (KNL, ER)

- Scale = 16

- Different performance trends
  - Performance of MKL degrades with increasing density

![Graph showing performance evaluation results with different algorithms and edge factors.]
Performance Evaluation
A^2: Scaling with density (KNL, ER)

- Performance gain with keeping output unsorted
Performance Evaluation

A^2: Scaling with density (KNL, G500)

- Denser inputs do not simply bring performance gain
  - Different from ER matrices
Performance Evaluation
A^2: Scaling with density (Haswell)

- HashVector achieves much higher performance
Performance Evaluation
A^2: Scaling with input size (KNL, ER)

- Edge factor = 16
- Hash and HashVector show good performance in any input size

![Graph showing performance evaluation](image-url)
Performance Evaluation

A^2: Scaling with input size (KNL, ER)

- **Performance gain with keeping output unsorted**
- **MKL for small scale ↔ HashVector for large scale**

![Graph showing performance evaluation](image)
Performance Evaluation
A^2: Scaling with input size (KNL, G500)

- Hash is best performer
Performance Evaluation
A^2: Scaling with input size (Haswell)

- **More clear performance trend of KNL**
  - MKL for smaller scales
  - Hash and HashVector for larger scales
Performance Evaluation

A^2: Scalability (KNL)

- **Good scalability of Hash and HashVec** even after 64 threads
Performance Evaluation

A^2: Sensitivity of compression ratio (KNL)

- Evaluation on SuiteSparse matrices
- Compression ratio (CR): #flop/#non-zero of output
- Heap: stable performance
- **MKL and Hash: Better performance with higher CR**
Performance Evaluation
A^2: Sensitivity of compression ratio (KNL)

- Hash for low CR ⇔ MKL family for high CR
- KokkosKernel underperforms other kernels
Performance Evaluation
A^2: Profile of Relative Performance

- **Sorted**: Hash is best performer for 70% matrices
  - Runtime of Hash is always within 1.6x of the best

- **Unsorted**: Hash, HashVector and MKL-inspector perform equally
  - Each of them performs the best for about 30%
Performance Evaluation
Square x Tall-skinny matrix (KNL)

- Multiple BFS, Betweenness Centrality
- **Hash or HashVec is the best performer**

![Graph showing performance evaluation results for different algorithms and scales.](image-url)
Performance Evaluation
Triangle Counting on SuiteSparse matrices (KNL)

- Reorders and transforms a matrix to L and U
  - L is lower triangle and U is upper triangle
- Similar performance trend to that of $A^2$
  - Hash and HashVector generally overwhelm MKL
Empirical Recipe for SpGEMM on KNL

(a) Real data specified by compression ratio (CR)

<table>
<thead>
<tr>
<th></th>
<th>High CR (&gt;2)</th>
<th>Low CR (&lt;=2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A x A</td>
<td>Sorted</td>
<td>Hash</td>
</tr>
<tr>
<td></td>
<td>Unsorted</td>
<td>MKL-inspector</td>
</tr>
<tr>
<td>L x U</td>
<td>Sorted</td>
<td>Hash</td>
</tr>
</tbody>
</table>

(b) Synthetic data specified by sparsity and non-zero pattern

<table>
<thead>
<tr>
<th></th>
<th>Sparse (Edge factor &lt;=8)</th>
<th>Dense (Edge factor &gt; 8)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Uniform</td>
<td>Skewed</td>
</tr>
<tr>
<td>A x A</td>
<td>Sorted</td>
<td>Heap</td>
</tr>
<tr>
<td></td>
<td>Unsorted</td>
<td>HashVec</td>
</tr>
<tr>
<td>Tall-Skinny</td>
<td>Sorted</td>
<td>HashVec</td>
</tr>
<tr>
<td></td>
<td>Unsorted</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>Hash</td>
</tr>
</tbody>
</table>
Conclusion

- Performance analysis of SpGEMM on Intel KNL and multicore architectures
  - Optimizing implementation for these architectures
    - Identify the bottlenecks
    - Evaluation in various use cases
      - Clarify which SpGEMM algorithm works well
      - Highlighting the benefit of leaving matrices unsorted
      - Empirical recipe for selecting the best-performing algorithm for a specific application scenario

Source code is publicly available at https://bitbucket.org/YusukeNagasaka/mtspgemmlib