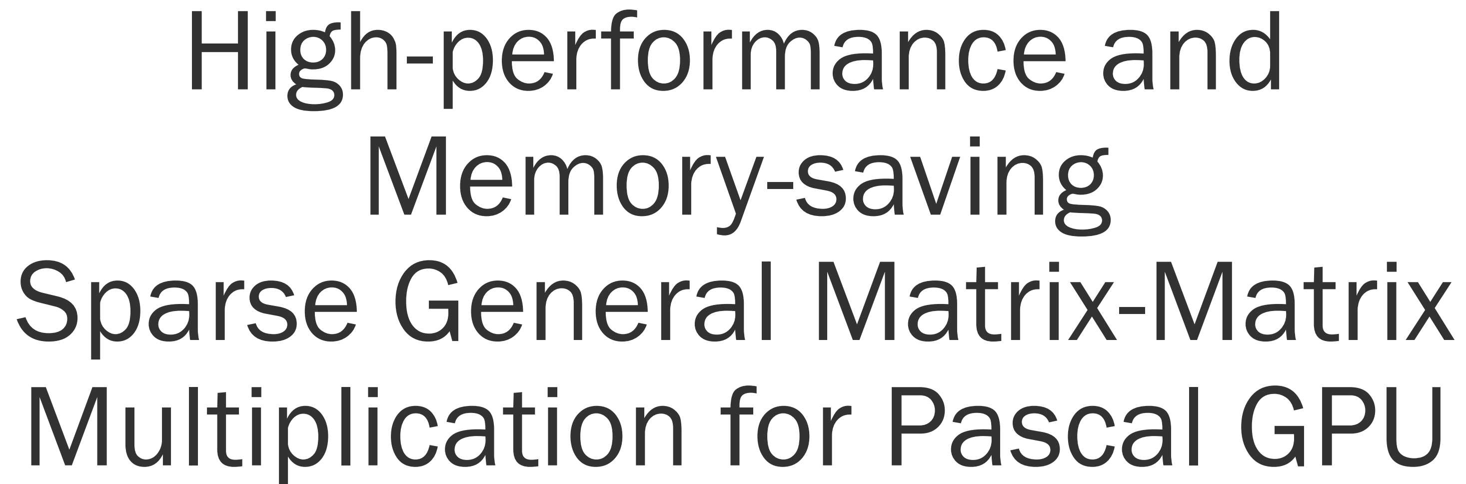
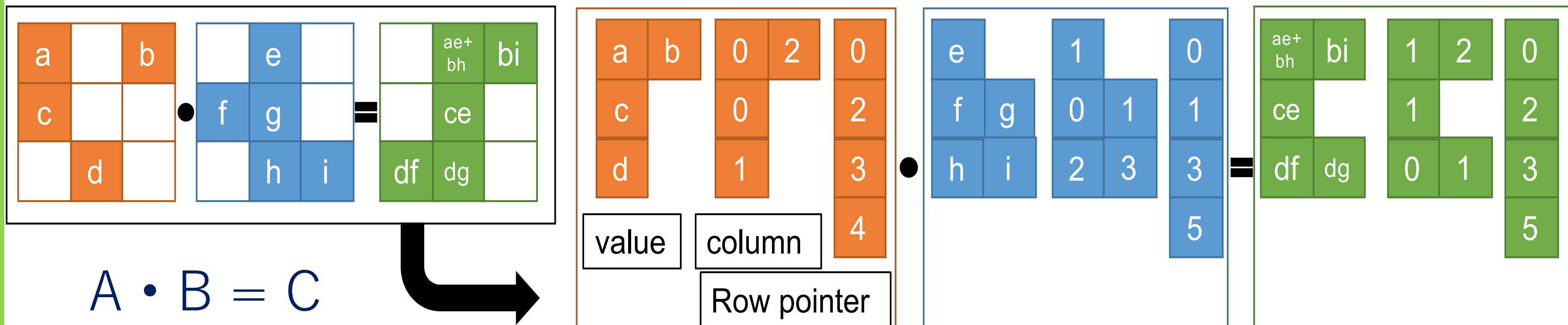


High-performance and Memory-saving Sparse General Matrix-Matrix Multiplication for Pascal GPU

Yusuke Nagasaka, Akira Nukada, Satoshi Matsuoka
Tokyo Institute of Technology

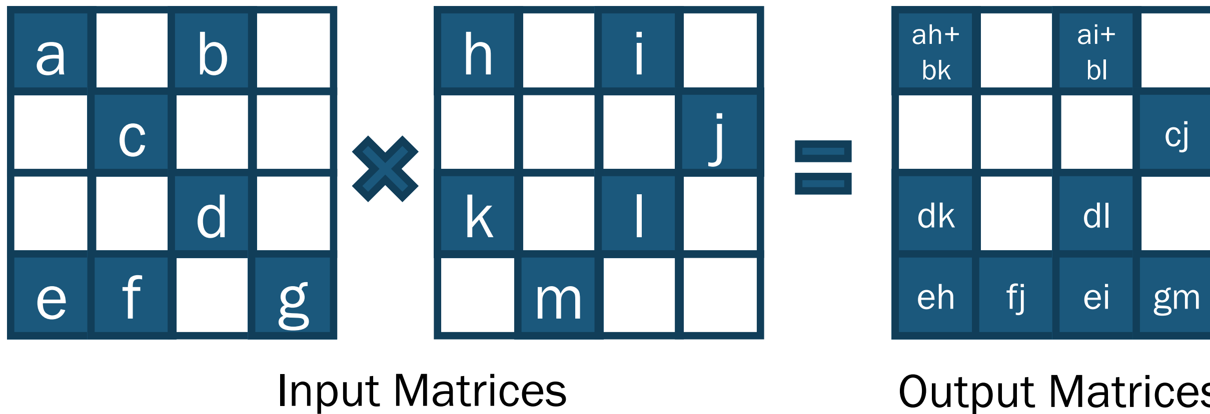
Sparse General Matrix-Matrix Multiplication (SpGEMM)

- Numerical application, graph processing
  - AMG method, graph clustering
- Low performance
  - Non-zero pattern of output matrix is unknown before execution
    - Accumulate intermediate products into one non-zero element
    - Hard to manage memory allocation



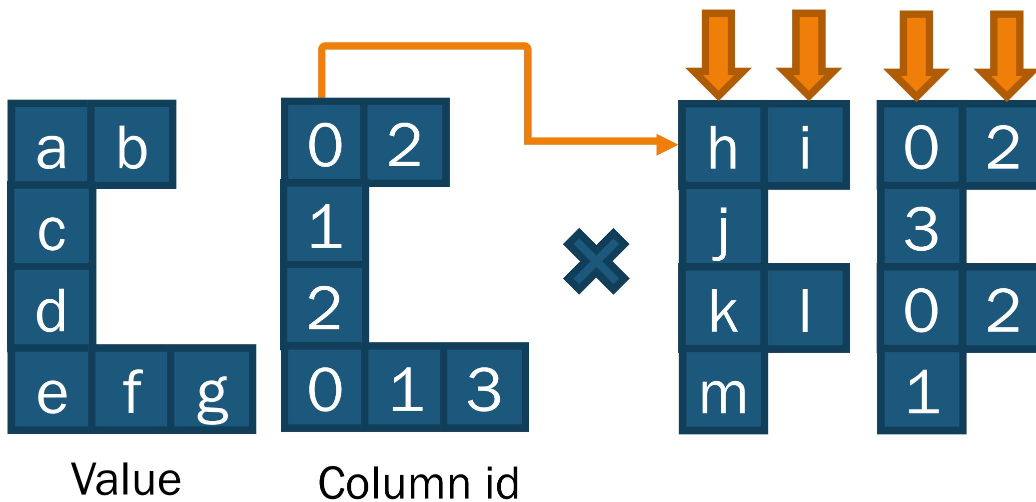
Accumulation of intermediate products

Sparse Accumulator (SPA) [Gilbert, SIAM1992]



Input Matrices

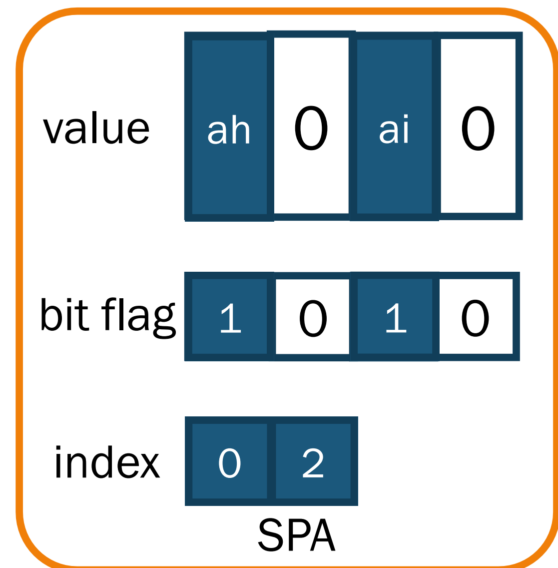
Output Matrices



Value

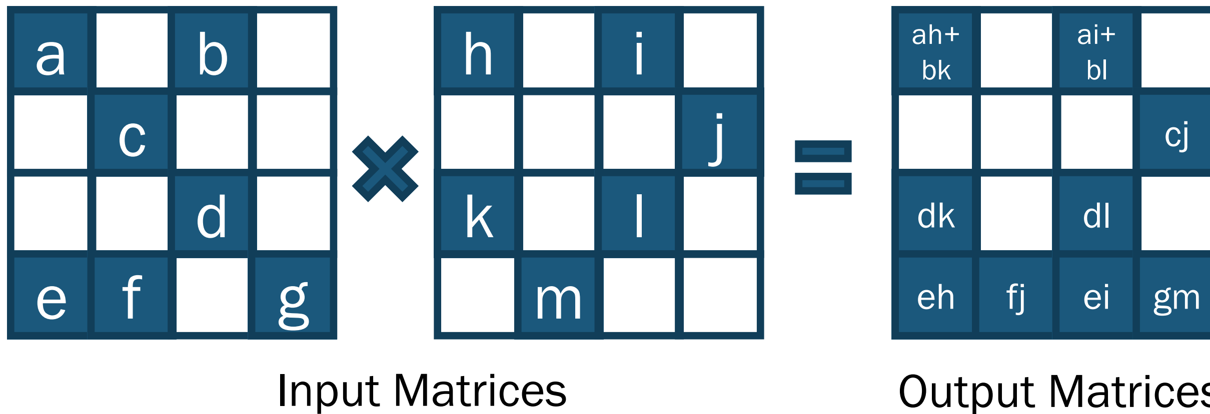
Column id

Input matrices in sparse format



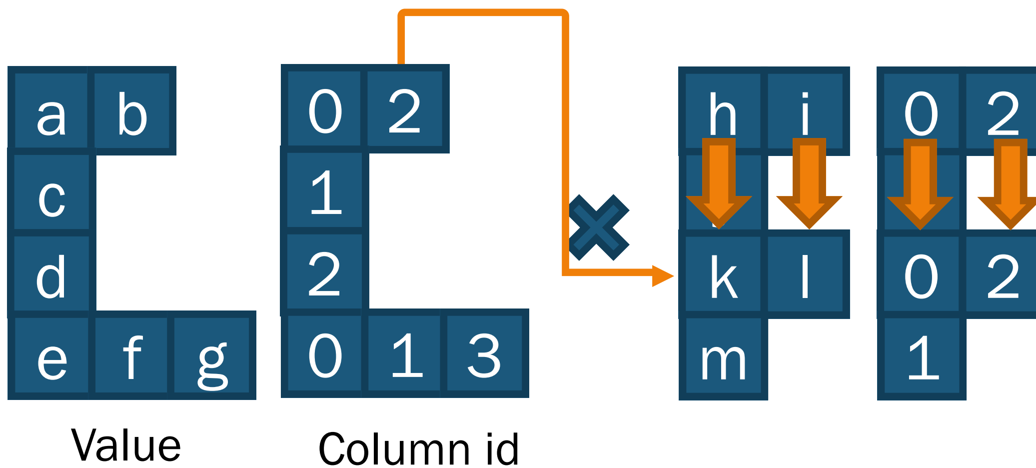
Accumulation of intermediate products

Sparse Accumulator (SPA) [Gilbert, SIAM1992]



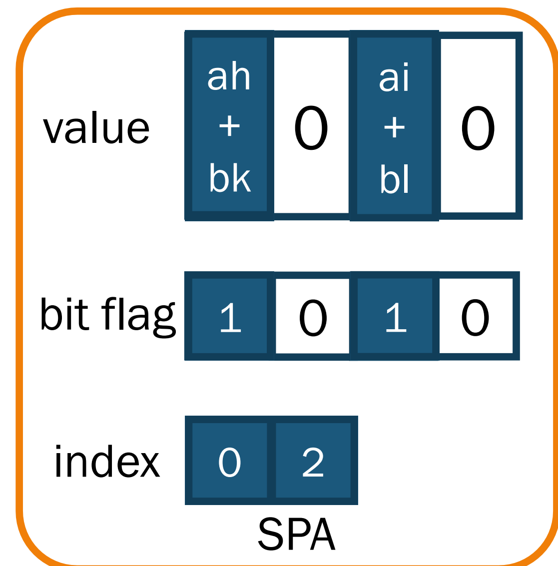
Input Matrices

Output Matrices



Value

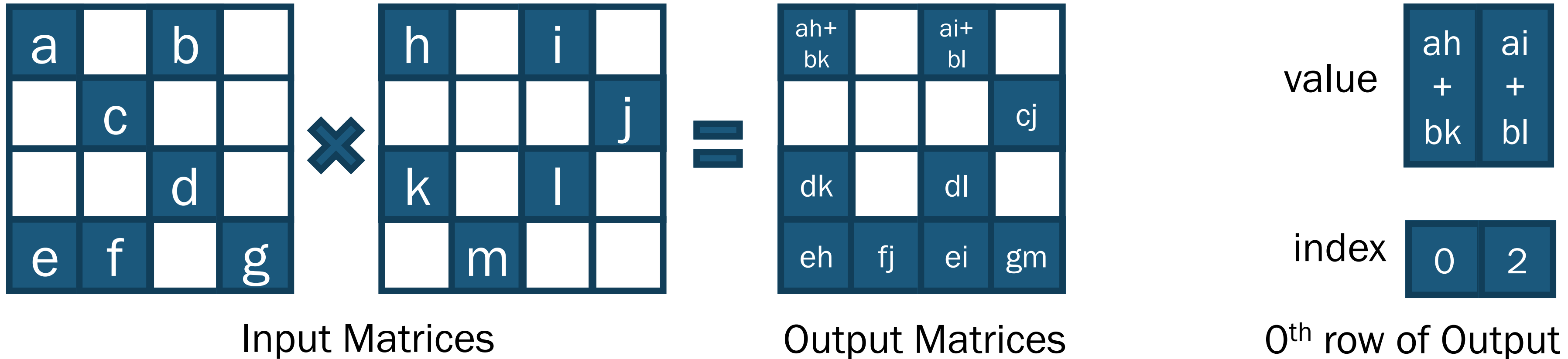
Column id



Input matrices in sparse format

Accumulation of intermediate products

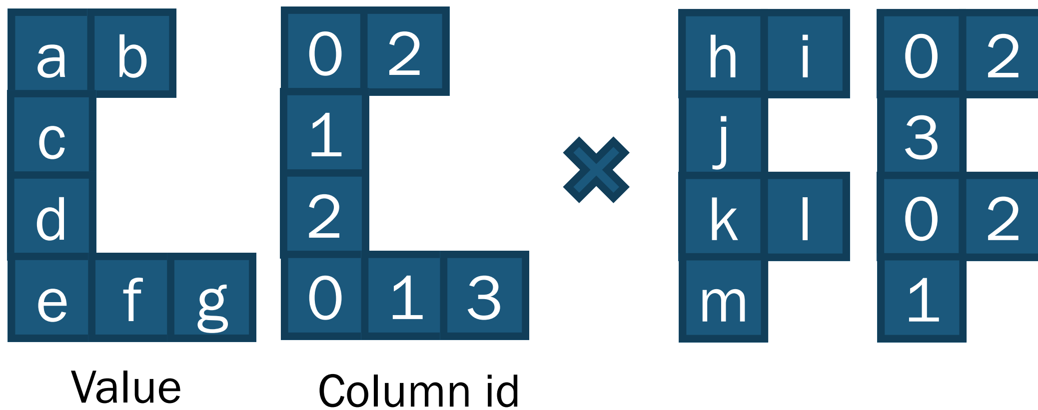
Sparse Accumulator (SPA) [Gilbert, SIAM1992]



Input Matrices

Output Matrices

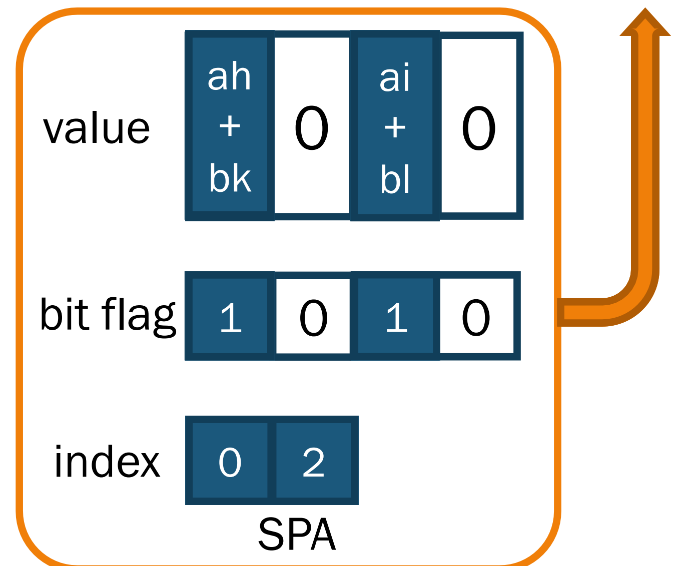
0th row of Output



Value

Column id

Input matrices in sparse format

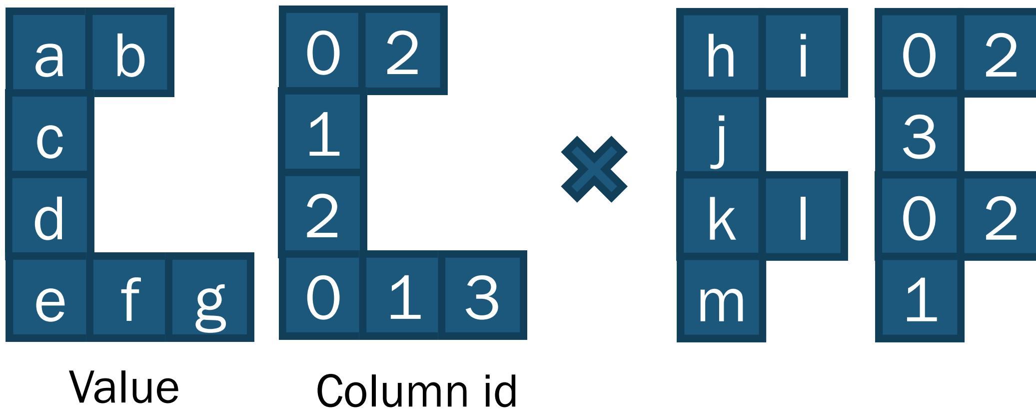


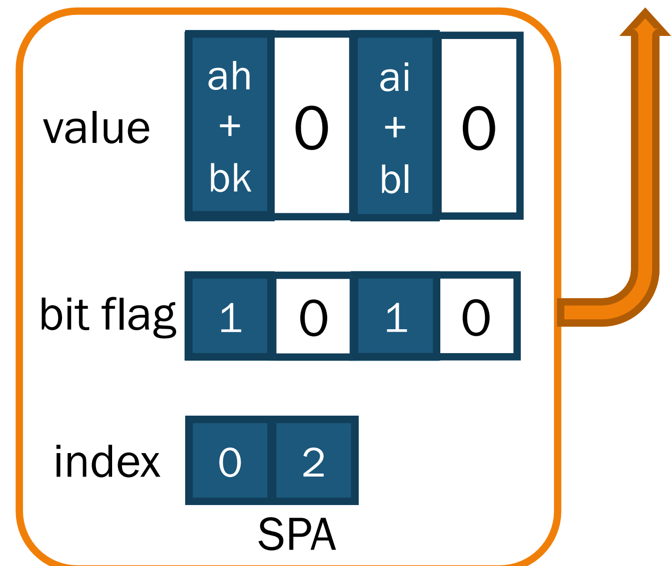
Accumulation of intermediate products

Sparse Accumulator (SPA) [Gilbert, SIAM1992]

☺ Efficient accumulation of intermediate products: Lookup cost is O(1)

☹ Require O(#columns) memory by one thread





Input matrices in sparse format

Memory Allocation of Output Matrix

- Non-zero pattern of output is unknown before execution
  - Cannot allocate exact memory space for output before execution
- Two ways for allocation of output
  - 1-phase
    - Allocate enough large memory space for output
  - 2-phase
    - Count #non-zero of output, then allocate memory for output

	Computation cost	Memory usage	Libraries
1-phase	Low	Large	CUSP, BHSPARSE
2-phase	High	Small	cuSPARSE

SpGEMM on GPU

- Massive parallelism
  - Simple row/column-based parallelization causes load-imbalance
    - Largely different computation cost by row/column
- Difficulty of memory management
  - Small global memory
    - Up to 16GB (P100 GPU)
  - Hierarchical memory
    - Shared memory (fast, but only 64KB/SM on P100)

Contribution

- We propose GPU-optimized fast SpGEMM algorithm with low memory usage
  - Efficiently manage column index of output matrix and accumulate intermediate products by hash table
    - Utilize GPU's shared memory for hash table
  - Make row groups by the number of non-zero elements or intermediate products to improve load balance
  - Evaluate the performance of SpGEMM for the Sparse Matrix Collection from University Florida
    - Up to x4.3 in single precision, x4.4 in double precision
    - Memory usage is reduced by
      - 14.7% in single precision
      - 10.9% in double precision

Related work (1)

- ESC Algorithm [Bell, SIAM2012]
  - **E**xpansion: Generate the list of all intermediate products
  - **S**orting by column and row indices
  - **C**ontraction: Accumulate intermediate products
  - Each part can be executed with high parallelism
    - Whole performance is low since ESC requires large memory access, and also large memory space
- BHSPARSE [Liu, IPDPS2014]
  - For irregular matrices
  - Binning by the number of intermediate products per row
    - Switch the algorithms of accumulation by bin
      - Heap method, bitonic ESC method, mergepath
    - Better load-balance

Related work (2)

- Balanced Hash [Anh, ICS'16]
  - Improve load balance
    - Worklist: pair of indices for computation of intermediate products
      - Worklist is stored on global memory
  - Improve the process of accumulation
    - Use hash table
      - Fixed size of hash table on shared memory
        - Waste shared memory when the number of non-zero is small
      - When hash collision occurs, the products are added to queue
        - Store the calculated elements in the table to memory, refresh table, and then process the products in queue
        - Repeat until queue becomes empty
        - Additional memory usage and memory access to queue

Proposed Algorithm

Key Points

- Two-phase execution
  - (1 - 4): Count #non-zero elements of output matrix
  - (6 - 7): Calculate output matrix
  - Minimize the usage of memory

(1) Count #intermediate products

(2) Divide the rows into groups by #intermediate products

(3) Count #non-zero elements

(4) Set row pointers of output matrix

(5) Memory allocation of output matrix

(6) Divide the rows into groups by #non-zero elements

(7) Compute the output matrix

a. Calculate values and column indices on hash table
b. Shrink the hash table
c. Store to the memory with sorting

Proposed Algorithm

Key Points

- Utilize hash table for accumulator
  - Allocated on fast shared memory
- Divide the rows into groups by #intermediate products or #non-zero elements
  - Improve load balance by appropriate thread assignment
  - Better utilization of shared memory by coordinating hash table size

Proposed Algorithm

Count #intermediate products / Grouping

- Rows are divided into several groups by #intermediate products or non-zero elements
  - Improve the load-balance
  - Utilize shared memory
  - #intermediate products is upper bound of #non-zero elements
    - Counting cost of #intermediate product is relatively small

Algorithm 2 Count the number of intermediate products of i-th row

$n_{prod} \leftarrow 0$
for $j = rpt_A[i]$ to $rpt_A[i+1]$ **do**
  $n_{prod} \leftarrow n_{prod} + (rpt_B[col_A[j]+1] - rpt_B[col_A[j]])$
end for

(1) Count #intermediate products

(2) Divide the rows into groups by #intermediate products

(3) Count #non-zero elements

(4) Set row pointers of output matrix

(5) Memory allocation of output matrix

(6) Divide the rows into groups by #non-zero elements

(7) Compute the output matrix
a. Calculate values and column indices on hash table
b. Shrink the hash table
c. Store to the memory with sorting

Proposed Algorithm

Count #Non-zero Elements / Compute the output

- Two-way thread assignment and memory access to input matrices for load-balance
  - Appropriate thread assignment for both dense row and sparse row
- Column indices of output matrix are managed by hash table
  - Tables are on shared memory
- CUDA kernel for each group
  - In order to execute concurrently, each kernel is assigned to different CUDA stream

(1) Count #intermediate products

(2) Divide the rows into groups by #intermediate products

(3) Count #non-zero elements

(4) Set row pointers of output matrix

(5) Memory allocation of output matrix

(6) Divide the rows into groups by #non-zero elements

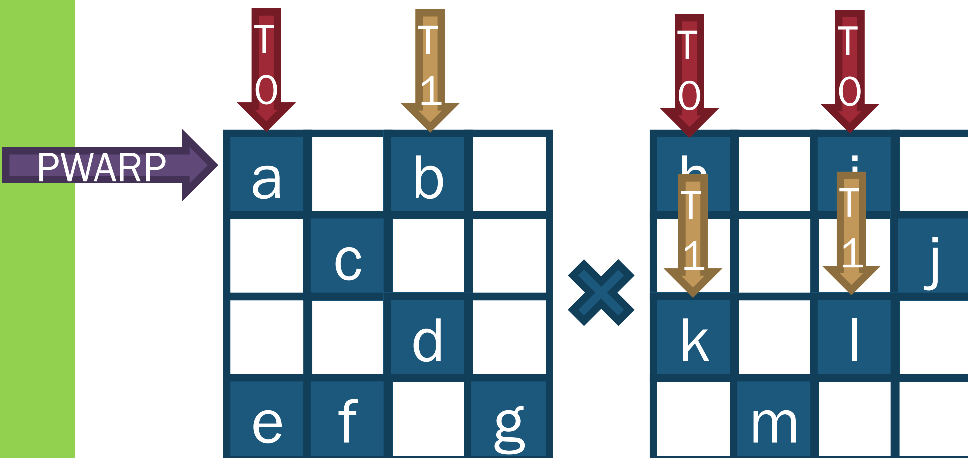
(7) Compute the output matrix

a. Calculate values and column indices on hash table
b. Shrink the hash table
c. Store to the memory with sorting

Proposed Algorithm

Two-ways thread Assignment -1-

- **PWARP/ROW:** Partial warp / row
  - Partial warp means a bundle of 4 threads
  - 1 pwarp for each row of matrix A, and 1 thread for each non-zero element of A and corresponding row of B
  - Selected for the groups with sparser rows



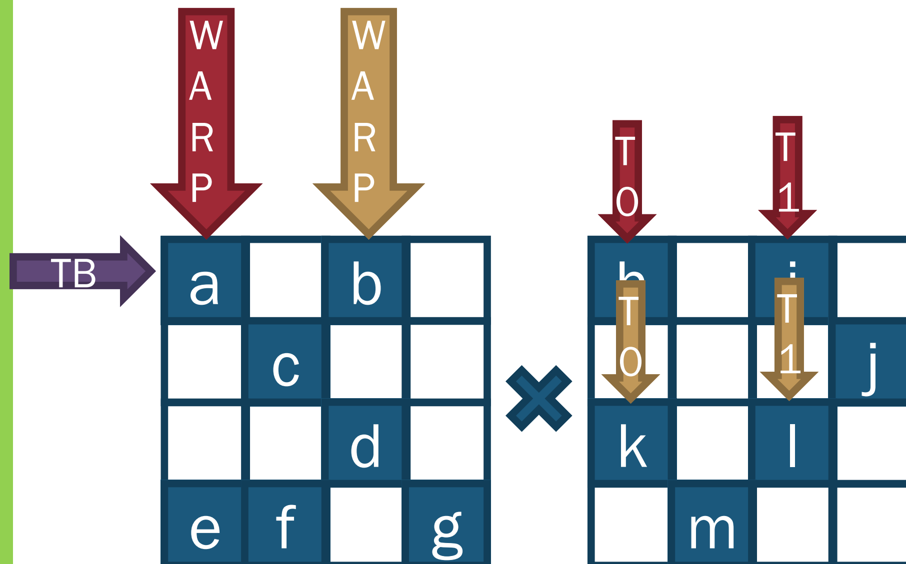
Algorithm 3 Count the number of non-zero elements of i-th row by PWARP/ROW

$tid \leftarrow threadIdx\%4$
for $j \leftarrow rpt_A[i]$ to $rpt_A[i+1]$ **stride** 4 **do**
  $d \leftarrow col_A[j + tid]$
  for $k \leftarrow rpt_B[d]$ to $rpt_B[d+1]$ **stride** 1 **do**
    //hash operation
  end for
end for

Proposed Algorithm

Two-ways thread Assignment -2-

- **TB/ROW:** Thread block / row
  - Assign 1 thread block (TB) for each row of matrix A, 1 warp for each non-zero element of A, and 1thread for each non-zero element of B
  - Selected for the groups with denser rows



Algorithm 4 Count the number of non-zero elements of i-th row by TB/ROW

$tid \leftarrow threadIdx\%warpsize$

$wid \leftarrow threadIdx/warpsize$

$wnum \leftarrow blockDim/warpsize$

for $j \leftarrow rpt_A[i] + wid$ to $rpt_A[i+1]$ **stride** $wnum$ **do**

    $d \leftarrow col_A[j]$

    for $k \leftarrow rpt_B[d] + tid$ to $rpt_B[d+1]$ **stride** 32 **do**

        //hash operation

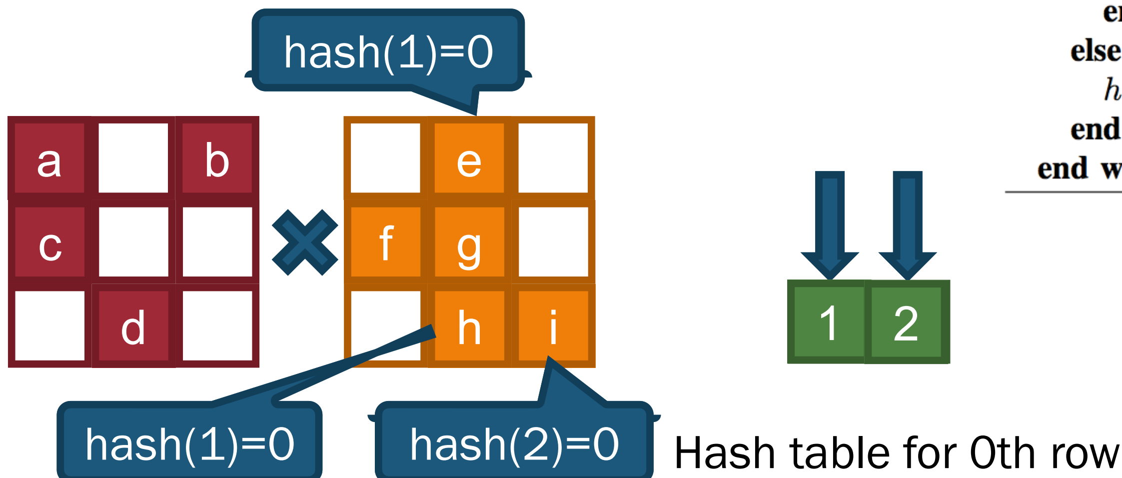
    end for

end for

Proposed Algorithm

Hash Table

- Key is column index of B
  - if empty, add the element
    - compare-and-swap
    - Each thread counts the number of non-zero elements
  - Linear probing
    - When the hash is collided, the algorithm tries next entry



Hash table for 0th row

Algorithm 5 Hash Algorithm

(Hash table is initialized: $table[] \leftarrow -1$)
(nz is initialized: $nz \leftarrow 0$)
(k comes from Algorithm 3, 4)
$key \leftarrow col_B[k]$
$hash \leftarrow (key * HASH_SCAL)\% t_{size}$
while true **do**
  if $table[hash] = key$ **then**
    break
  else if $table[hash] = -1$ **then**
    $old \leftarrow$ atomicCAS$(table + hash, -1, key)$
    if $old = -1$ **then**
      $nz \leftarrow nz + 1$
      break
    end if
  else
    $hash \leftarrow (hash + 1)\% t_{size}$
  end if
end while

Proposed Algorithm
Count #non-zero elements

- Accumulate the number of non-zero counted by each row
  - PWARP/ROW: Utilizing warp shuffle
  - TB/ROW: Accumulate by warp shuffle in warp level, and then accumulate the sum of each warp by using shared memory

(1) Count #intermediate products

(2) Divide the rows into groups by #intermediate products

(3) Count #non-zero elements

(4) Set row pointers of output matrix

(5) Memory allocation of output matrix

(6) Divide the rows into groups by #non-zero elements

(7) Compute the output matrix
a. Calculate values and column indices on hash table
b. Shrink the hash table
c. Store to the memory with sorting

Proposed Algorithm

Compute the output matrix

- Calculate values and column index as well as counting #non-zero
  - Allocate another hash table for value
  - Accumulate the value by atomicAdd
- Shrink table to hold only non-zero
- Output with sorting by column index

(1) Count #intermediate products

(2) Divide the rows into groups by #intermediate products

(3) Count #non-zero elements

(4) Set row pointers of output matrix

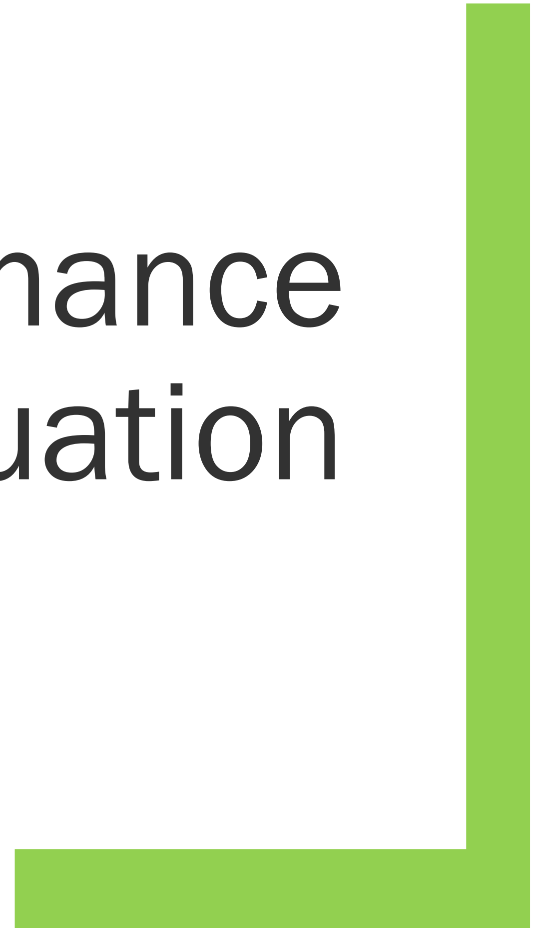
(5) Memory allocation of output matrix

(6) Divide the rows into groups by #non-zero elements

(7) Compute the output matrix

a. Calculate values and column indices on hash table
b. Shrink the hash table
c. Store to the memory with sorting

Performance Evaluation



Experimental Setup

- Pascal GPU Machine
  - CPU : Intel Xeon CPU E5-2650 v3
  - GPU : NVIDIA Tesla P100
    - SM : 56
    - CUDA cores : 3584 (64[/SM])
    - Memory size : 16 [GB]
    - Memory bandwidth : 732 [GB/sec]
    - ECC : Off
    - L2 cache size : 4[MB]
  - CUDA : Version 8.0
  - OS : CentOS release 7.2.1511

Experimental Setup

- Sparse Libraries
  - cuSPARSE
    - CUDA 8.0 version
  - CUSP : ESC algorithm [Dalton, 2014]
    - v0.5.1
  - BHSPARSE [Liu, IPDPS2014]
    - Effective for irregular matrices
- FLOPS Performance
  - Evaluate the performance of A^2
    - #(intermediate products) * 2 / (execution time)

Matrix Data

Florida Sparse Matrix Collection

Name	Row	Non-zero	Nnz /row	Max nnz / row	Intermediate product of A^2	Nnz of A^2
Protein	36,417	4,344,765	119.3	204	555,322,659	19,594,581
FEM /Spheres	83,334	6,010,480	72.1	81	463,845,030	26,539,736
FEM /Cantilever	62,451	4,007,383	64.2	78	269,486,473	17,440,029
FEM /Ship	140,874	7,813,404	55.5	102	450,639,288	24,086,412
Wind Tunnel	217,918	11,634,424	53.4	180	626,054,402	32,772,236
FEM /Harbor	46,835	2,374,001	50.7	145	156,480,259	7,900,917
QCD	49,152	1,916,928	39.0	39	74,760,192	10,911,744
FEM /Accelerator	121,192	2,624,331	21.7	81	79,883,385	18,705,069
Economics	206,500	1,273,389	6.2	44	7,556,897	6,704,899
Circuit	170,998	958,936	5.6	353	8,676,313	5,222,525
Epidemiology	525,825	2,100,225	4.0	4	8,391,680	5,245,952
webbase	1,000,005	3,105,536	3.1	4700	69,524,195	51,111,996
cage15	5,154,859	99,199,551	19.2	47	2,078,631,615	929,023,247
wb-edu	9,845,725	57,156,537	5.8	3841	1,559,579,990	630,077,764
cit-Patents	3,774,768	16,518,948	4.4	770	82,152,992	68,848,721

High-Throughput Matrix Data

Low-Throughput Matrix Data

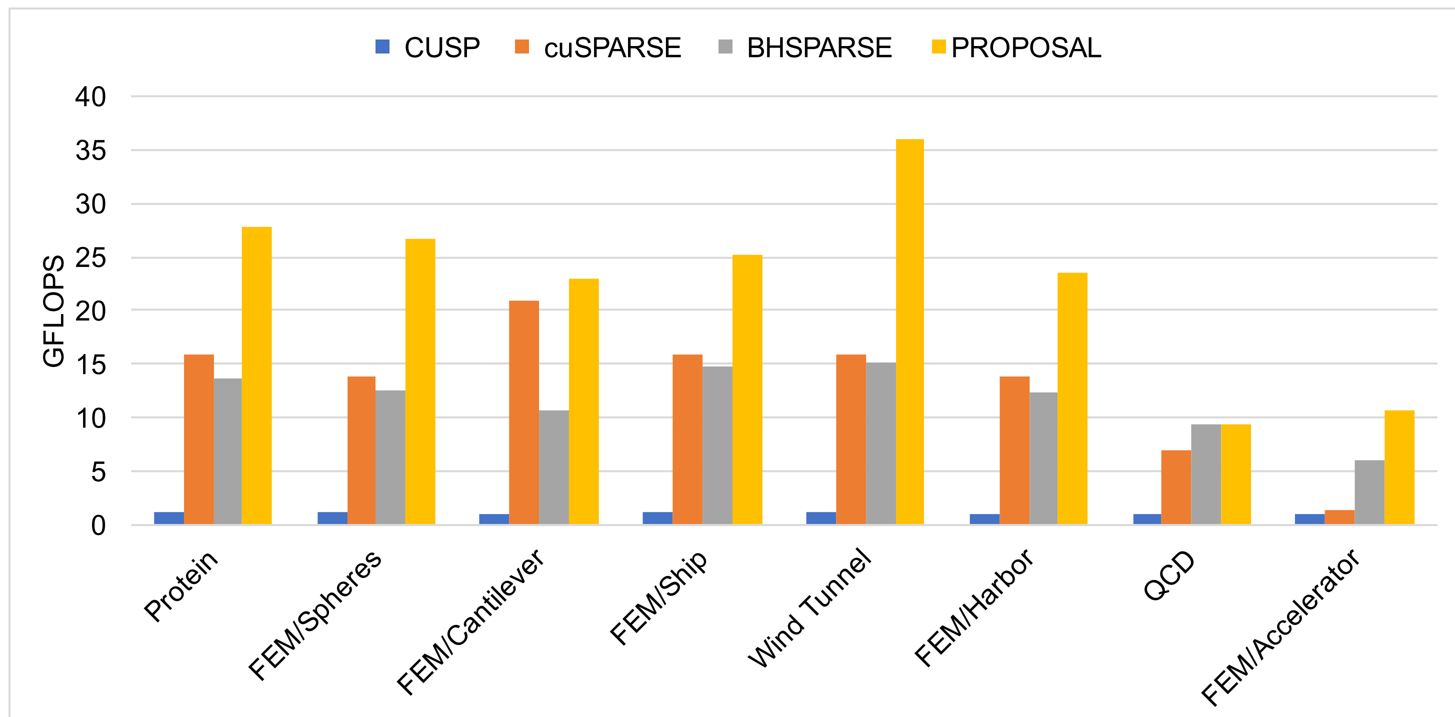
Large-size Graph Data

Parameter Setting for P100 GPU

	(3) #intermediate products	(6) #non-zero elements	Assignment	Thread block size
On global memory	8193 -	4097 -	TB / ROW	1024
On shared memory	4097 - 8192	2049 - 4096	TB / ROW	1024
	2049 - 4096	1025 - 2048	TB / ROW	512
	1025 - 2048	513~1024	TB / ROW	256
	513~1024	257 - 512	TB / ROW	128
	33 - 512	17 - 256	TB / ROW	64
	0 - 32	0 - 16	PWARP / ROW	512

Performance -Single Precision-
High-Throughput Matrix Data

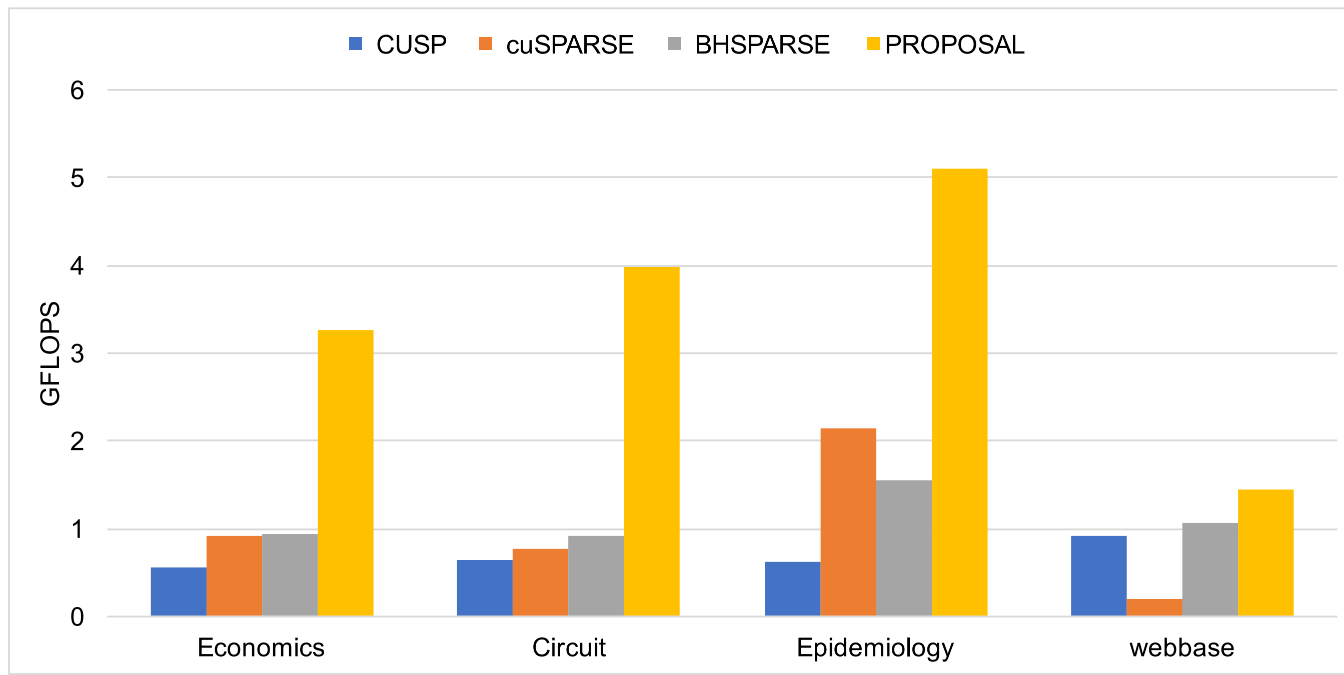
- Proposal > cuSPARSE > BHSPARSE
  - Speedup is up to x2.26



Performance -Single Precision-

Low-Throughput Matrix Data

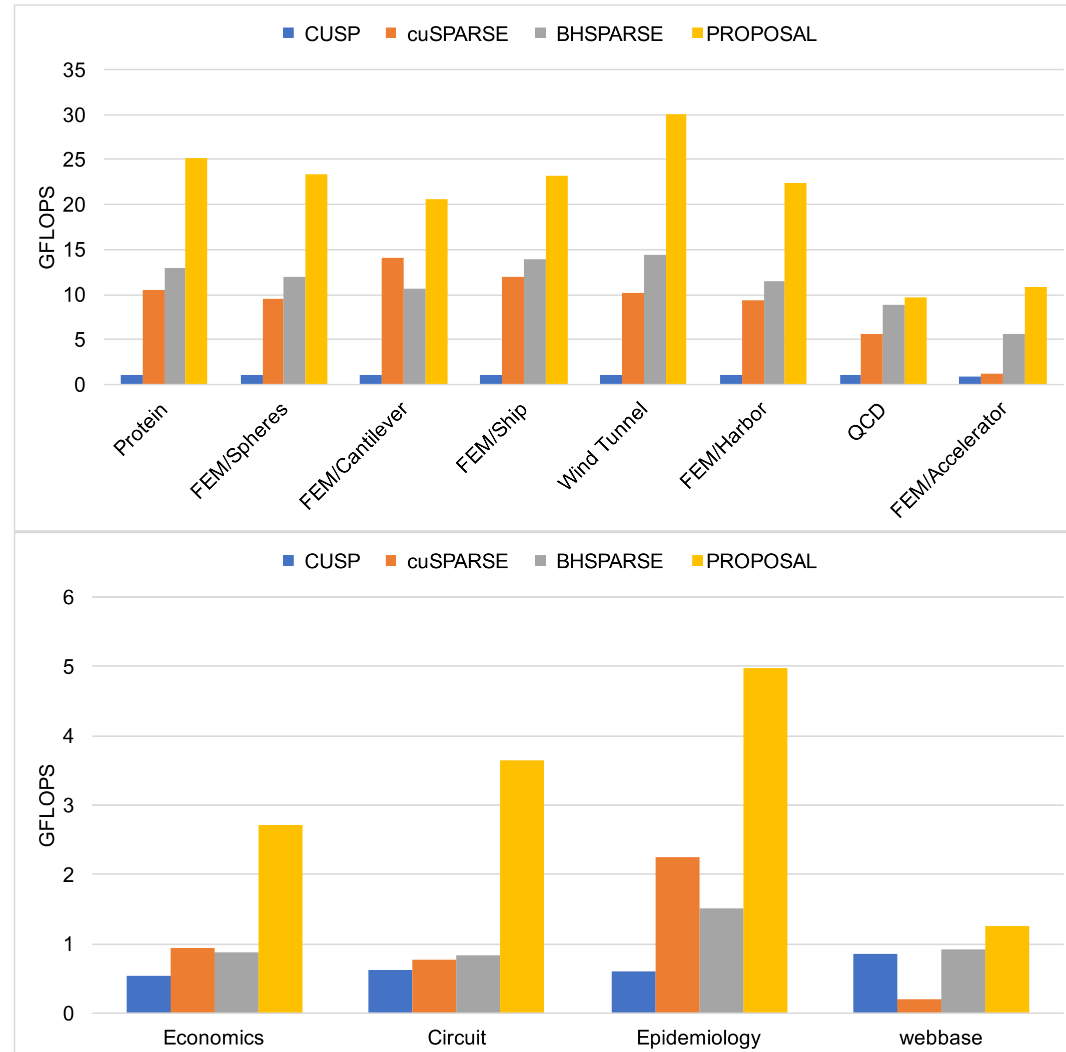
- Proposal > BHSPARSE > cuSPARSE
- Dividing rows into groups improves load-balance for irregular matrices like ‘webbase’
  - Speedup is up to x4.3



Performance -Double Precision-

High-Throughput Matrix Data

- Similar performance trend as single precision
  - Speedup is up to x2.1 for High-Throughput
  - Speedup is up to x4.4 for Low-Throughput



Performance -Double Precision-

Large-size Graph Data

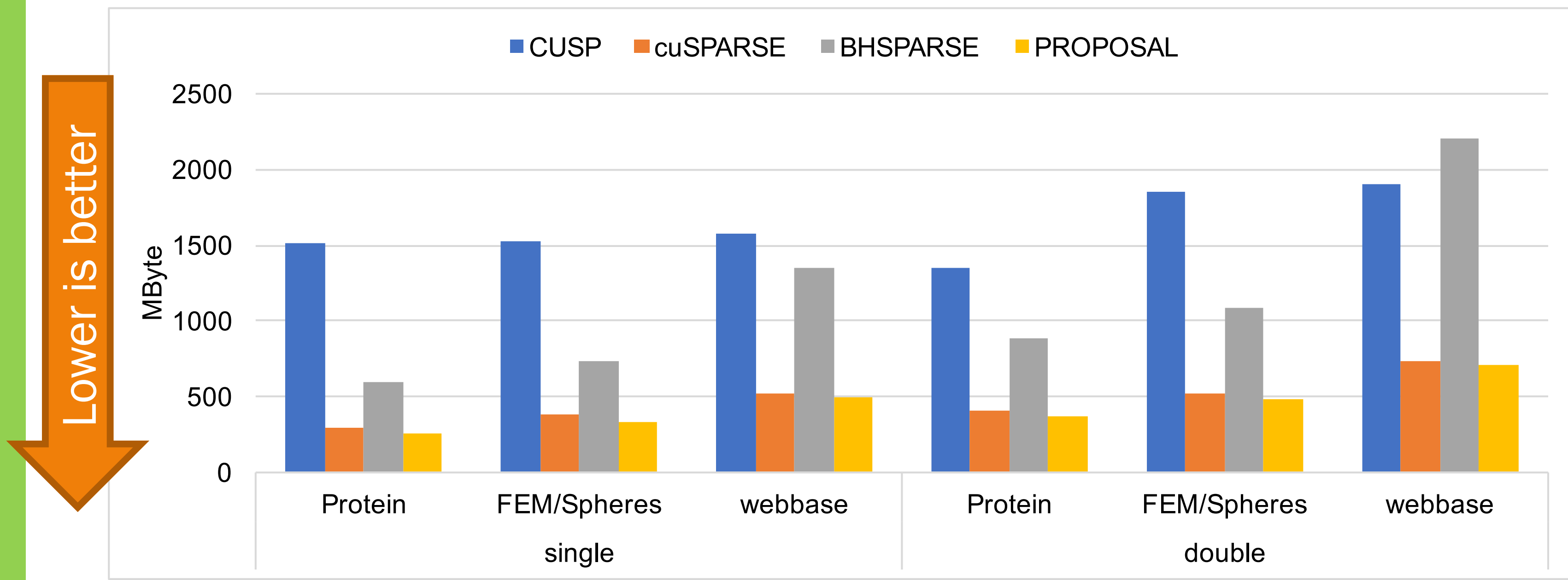
- Our approach shows significant speedups for large size graph data
  - BHSPARSE cannot handle ‘cage15’ and ‘wb-edu’ because of memory shortage

Precision	Matrix	CUSP	cuSPARSE	BHSPARSE	PROPOSAL	**Speedup from cuSPRASE**	**Speedup from BHSPARSE**
Single	cage15	-	0.519	-	5.955	**x11.5**	**-**
	wb-edu	-	2.348	-	5.403	**x2.4**	**-**
	cit-Patents	0.837	0.028	0.880	3.351	**x119.6**	**x3.8**
Double	cage15	-	0.491	-	5.684	**x11.6**	**-**
	wb-edu	-	2.145	-	4.618	**x2.2**	**-**
	cit-Patents	0.780	0.028	0.813	2.980	**x106.8**	**x3.7**

[GFLOPS]

Memory Usage

- Lower memory usage compared to other sparse matrix libraries for all matrix data
  - Compared to cuSPARSE, reduced by 14.7% in single precision and 10.9% in double precision on average
  - For the matrix data webbase, our proposal not only achieves better performance but also reduces memory usage by 67.7%



Conclusion

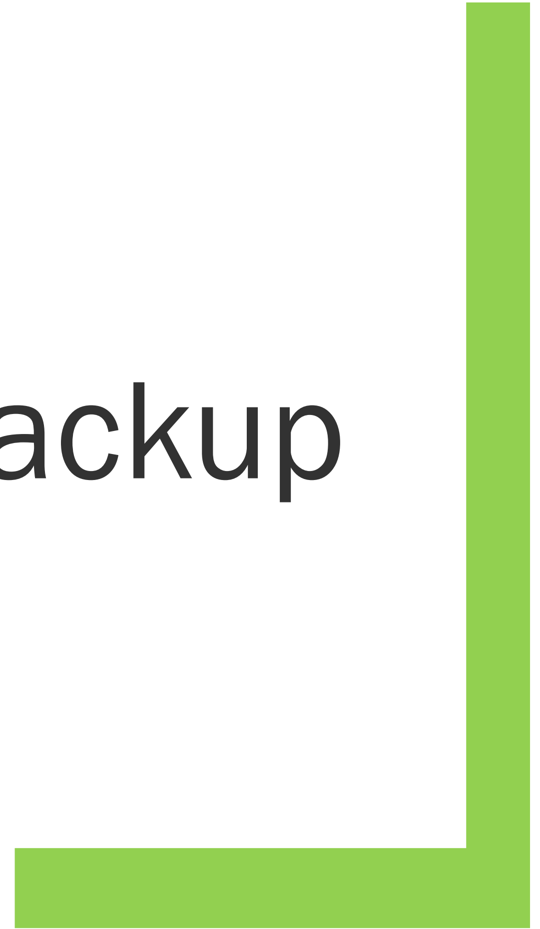
- We propose fast and memory-saving SpGEMM algorithm for GPU
  - Appropriate grouping and utilizing shared memory
  - Performance evaluation with cuSPARSE and BHSPARSE
    - Speedups are up to x4.3 in single precision and x4.4 in double precision
    - Memory usage is reduce by 14.7% in single precision and 10.9% in double precision on average
    - For Low-Throughput matrix, our algorithm achieves higher performance and reduces memory usage by 67.7%
- Future work
  - Evaluate on AMD GPU and Xeon Phi
  - Evaluate our SpGEMM algorithm in real-world application

Acknowledgement

- This work is partially supported by by JST CREST Grant Number JPMJCR1303 and JPMJCR1687

- Source code of proposed SpGEMM algorithm for GPU is released under **nsparse** library
  - https://github.com/EBD-CREST/nsparse

Backup



Performance Breakdown

- Largely reduce calculation time from cuSPARSE
- Grouping phase affects little to total performance
- On sparser matrices, cudaMalloc becomes bottleneck