Fast Sparse General Matrix-Matrix Multiplication on GPU with Low Memory Usage

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

Introduction and Motivation

Sparse general matrix-matrix multiplication (SpGEMM) is one of the key kernel of preconditioner such as algebraic multigrid (AMG) method or graph algorithms. Even though SpGEMM is matrix-matrix multiplication, the performance of SpGEMM is quite low since the memory access to both input matrix and output matrix is random. The pattern of non-zero elements of output matrix is not known beforehand, which makes it hard to manage the memory usage. There are several GPU implementations of fast SpGEMM computation while consuming large temporal memory. We devise new SpGEMM algorithm which can be applied to various type matrix data and achieves further speedups on GPU by utilizing shared memory and appropriate case analysis.

Proposal : *Fast and Memory Saving SpGEMM Algorithm*

Flow of new algorithm

(1) Count the number of intermediate products of each row
↓
(2) Group rows by the number of intermediate products
↓
(3) Count the number of non-zero elements of each row of output matrix for all groups with CUDA kernels and streams
↓
(4) Set row pointers of output matrix to store in CSR by Thrust scan
↓
(5) Group rows by the number of non-zero elements
↓
(6) Calculate the output matrix for all groups with CUDA kernels and streams
  a. Calculate values and column indices on hash table
  b. Shrink table to hold only non-zero elements
  c. Sort by column index in ascending order

Two Way Thread Assignment and Memory Access

Better load-balancing with perfect coalesced memory access to input matrices

WARP/ROW : One warp for each row of A, one thread for each non-zero of A and B

Counting non-zero elements of C's i-th row

```
tid ← threadIdx % warp
for j ←rpt_A[i], rpt_A[i + 1] stride 32 do
  col ←col_A[j + tid]
  for k ←0, 32 do
    dest ←__shfl(col, k)
    for g ←rpt_B[dest] + tid, rpt_B[dest + 1]
                                  stride 32 do
      // hash operation
    end for
  end for
end for
```

TB/ROW : One thread block for each row of A, one warp for each non-zero of A, one thread for each non-zero of B

Counting non-zero elements of C's i-th row

```
tid ← threadIdx % warp
wid ← threadIdx / warp
wnum ← blockDim / warp
for j ←rpt_A[i] + wid, rpt_A[I + 1] stride wnum do
  dest ← col_A[j]
  for k ←rpt_B[dest] + tid, rpt_B[dest + 1]
                                  stride 32 do
    // hash operation
  end for
end for
```

Hash Table Construction

Counting the number of non-zero and calculating value and column index of C by using hash table

- Table size is set based on group
- Basically hash table is on shared memory (Group which has large number of products or non-zero elements only uses global memory for hash table)
- Linear Probing Hashing
  - Execution can be on only shared memory

Part of Count non-zero of each row of C

```
(Hash table is initialized : table[] ← (-1))
key ← col_B
hash ← (col_B * HASH_SCAL) % table_size
nz ← 0
while true do
  if (table[hash] == key) then break
  else if (table[hash] == -1) then
    old ← atomicCAS(table + hash, -1, key)
    if (old == -1) then
      nz ← nz + 1
      break
    end if
  else
    hash ← (hash + 1) % table_size
end while
```

Thread Block Size and Shared Memory Size Setting

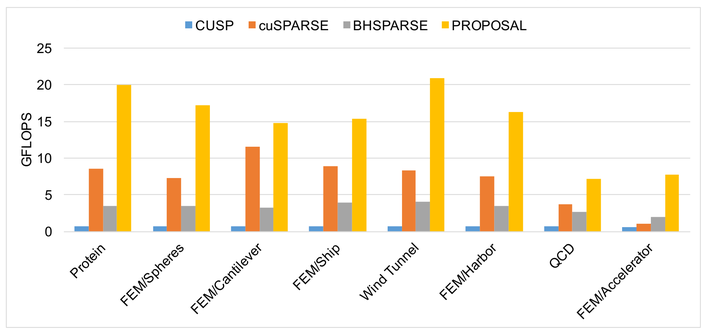
Algorithm forms 7 groups of rows by the number of (2)intermediate products or (5)non-zero elements of each row of output matrix. Thread blocks are fully assigned to SMM on Maxwell GPU (2048 threads / SMM, 96KB shared memory / SMM)

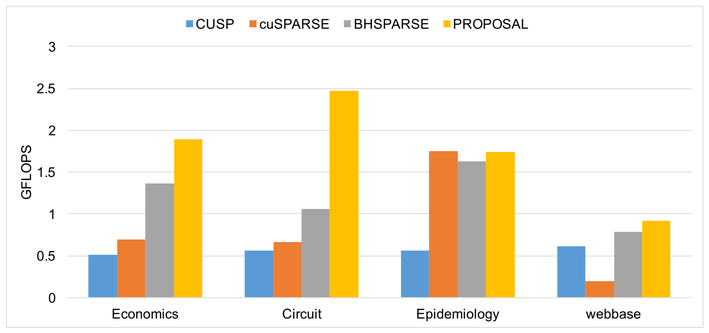
(2) Number	(5) Number	Assignment	TB size	TB / SMM
0 - 256	0 - 128	WARP/ROW	512	4
257 - 512	129 - 256	WARP/ROW	256	4
513 - 1024	257 - 512	TB/ROW	128	16
1025 - 2048	513 - 1024	TB/ROW	256	8
2049 - 4096	1025 - 2048	TB/ROW	512	4
4097 - 8192	2049 - 4096	TB/ROW	1024	2
8192 -	4097 -	TB/ROW	1024	2

Performance Evaluation

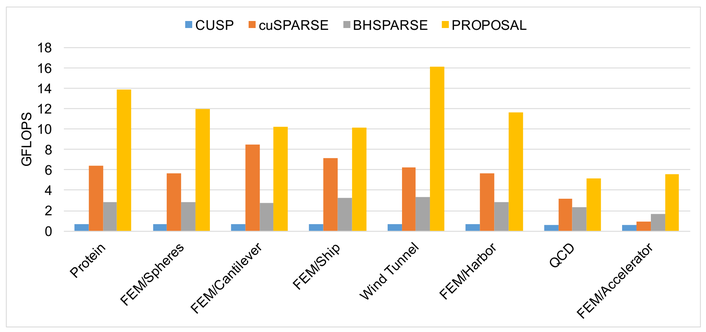
- NVIDIA Quadro M6000 GPU
- Matrix data is taken from the University of Florida Sparse Matrix Collection [1].
- Compared to
  - cuSPARSE (CUDA 8.0 RC)
  - CUSP [2] (ver. 0.5.1)
  - BHSPARSE [3]

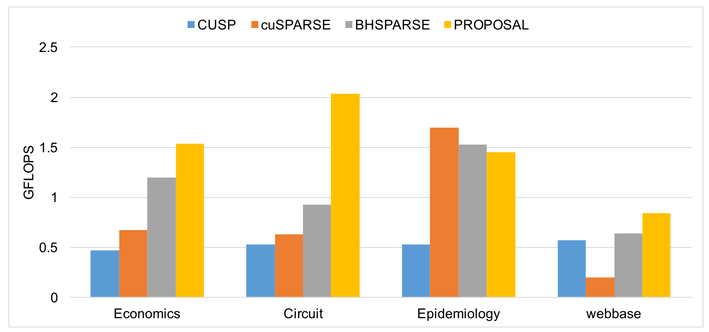
Single Precision : Speed up is up to **x4.0**





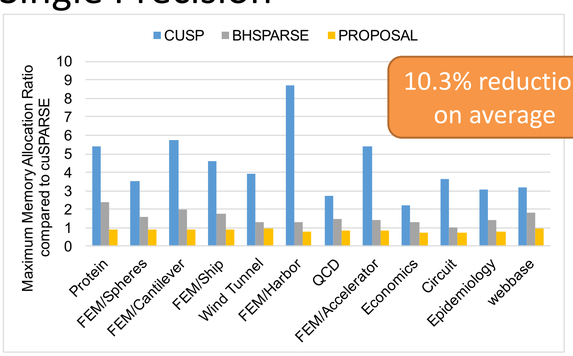
Double Precision : Speedup is up to **x3.3**



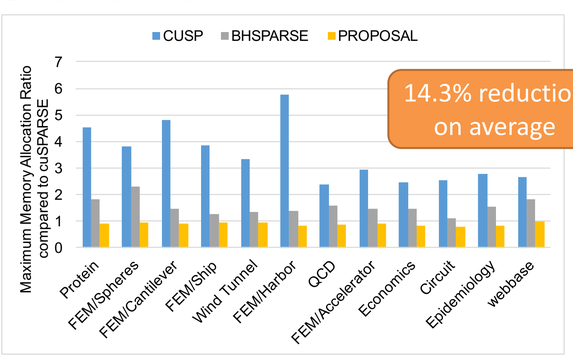


Maximum Memory Usage

Single Precision



Double Precision



Conclusion and Future work

We propose the novel sparse general matrix-matrix multiplication algorithm which is designed for reducing both memory overhead and execution time on GPU. Our algorithm achieves speedups of up to **x4.0** in single precision and **x3.3** in double precision compared to existing fast SpGEMM libraries.
For future work, we will apply our technique to the preconditioner such as AMG method and real-world applications.

Reference

[1] Tim Davis, "The University of Florida Sparse Matrix Collection", "http://www.cise.ufl.edu/research/sparse/matrices"
[2] Steven Dalton et al., "Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations", 2014, "http://cusplibrary.github.io/"
[3] Liu et al., "An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data", IPDPS, 2014,