Sparse Matrix-Matrix Multiplication on the GPU

Julien Demouth, NVIDIA
Introduction: Problem

- Two sparse matrices $A$ and $B$, compute:
  $$C = AB$$
- Sparse matrix: Many zeroes
- Only non-zero elements are stored in memory
foreach rowA : A.rows (in parallel):
    hashtbl = {}

    foreach (colA, valA) : rowA.nonZeroes:
        rowB = B.rows[colA]

        foreach (colB, valB) : rowB.nonZeroes:
            hashtbl[colB] += valA * valB

    store hashtbl

Two main steps:
- Find the # of non-zeroes per row of C
- Compute the value of non-zeroes of C
Introduction: Implementation

- CPU implementation

- GPU implementation

Thread’s hash table:
- Host memory (and cache)

Warp’s hash table:
- Shared memory
- Global memory
IMPLEMENTATION
Sparse Matrix Representation: CSR

- Do not store 0s
- Store column indices and values. Compress row indices
GPU Algorithm

- **Count** the number of non-zeroes in each row of C (1 kernel)
  - Conceptually, a warp computes the result for one row
    - When # of rows > # of warps, a warp works on more rows
  - Store the numbers in C.rows

- Exclusive scan on C.rows to compute offsets (1 Thrust call)

- **Compute** column indices and values (1 kernel)
  - Conceptually, a warp computes the pairs for one row
    - When # of rows > # of warps, a warp works on more rows
  - Store the column indices and values in C.cols and C.vals
Pseudo-code: Load \((colA, valA)\) Pairs

- One warp per row of \(A\) (coalesced loads):

  \[
  \text{warpId} = \text{threadIdx.x} / \text{WARP\_SIZE};
  \text{laneId} = \text{threadIdx.x} \% \text{WARP\_SIZE};
  \]

  \[
  \text{aColIt} = \text{A.\text{rows}[\text{warpId}]} + \text{laneId};
  \text{aColEnd} = \text{A.\text{rows}[\text{warpId} + 1]};
  \]

  \[
  \text{colA} = \text{aColIt} < \text{aColEnd} \text{? A.\text{cols}[aColIt]} : -1;
  \text{valA} = \text{aColIt} < \text{aColEnd} \text{? A.\text{vals}[aColIt]} : 0.0;
  \]

- Each thread keeps its pair \((colA, valA)\) in registers

  \[
  // \text{asm}( \text{"mov.u32 0, \%laneid;" : "=r\"( \text{laneId} )\"});
  \]
Pseudo-code: Vote to Load Rows of B

Each thread, in turn, will ask the warp to load its row of B:

```c
__shared__ volatile int sColA[nWarps];
__shared__ volatile double sValA[nWarps];

for( k = 0, end = __popc( __ballot( aColIt < aColEnd ) ) ; k < end ; ++k ) {
    if( laneId == k ) { sColA[warpId] = colA; sValA[warpId] = valA; }
    bColIt = B.rows[sColA[warpId]] ; // sColA is volatile and warp's threads
    bColEnd = B.rows[sColA[warpId] + 1]; // are implicitly synchronized
    for( bColIt += laneId ; __any( bColIt < bColEnd ) ; bColIt += 32 ) { Ω }
}
```

Number of valid (colA, valA) pairs
The kth thread pushes its (colA, valA) pair
As long as there are unread (colB, valB) pairs in the row colA of B

Note: Warp synchronization is a HW concept. Use it with care.
**Pseudo-code: Load (colB, valB) Pairs**

- Inside the \(\textcircled{1}\) loop, each thread loads its pair (colB, valB)
  
  ```
  colB = bColIt < bColEnd ? B.cols[bColIt] : -1;
  valB = bColIt < bColEnd ? B.vals[bColIt] : 0.0;
  ```

- And **inserts** its pair in the hash table
  
  ```
  hashtbl.insert( colB, sValA[warpId] * valB );
  ```

- Insertions are performed in **parallel**

- Important note: All threads have different colB values
Hash Table

- Stored in shared memory and global memory

```c
__shared__ volatile unsigned sKeys[nWarps][sMemSize]; // In our impl, sMemSize == 256
__shared__ volatile double sVals[nWarps][sMemSize];
```

- Insertion uses four different hash functions in order:

```c
foreach hashFunction:
    if( __all( inserted ) )
        break;
    tryToInsertInSharedMemory( colB, value );
```

```c
foreach hashFunction:
    if( __all( inserted ) )
        break;
    tryToInsertInGlobalMemory( colB, value );
```

- If it fails, we re-run the kernel using more global memory
Hash Table Insertion

- Each thread computes its hash value
- If the slot contains the same key, update the value
  \[s\text{Vals}[\text{warpId}][\text{hash}] += \text{value};\]
- If the slot is empty, try to insert:
  \[s\text{Keys}[\text{warpId}][\text{hash}] = \text{colB};\]
  \[\text{if}( s\text{Keys}[\text{warpId}][\text{hash}] == \text{colB} ) \quad // ?\]
  \[s\text{Vals}[\text{warpId}][\text{hash}] = \text{value};\]
- If the slot is full, retry with next hash function (next iteration)

Write conflict! Only one winner: \([\text{colB}=5 \quad \text{colB}=8]\)
Hash Table Size

- Hash tables in global memory contain $2^k$ slots
  - We usually start with 2048 slots
- We use inlined PTX to compute $\text{hash} \mod \text{kGlobalSize}$

```c
// Given $2^k$, it finds k.
int nBits;
asm( "bfind.u32 %0, %1;" : "=r"( nBits ) : "r"( kGlobalSize ) );

// Compute hash % (2 << nBits) == hash & ((2 << nBits)-1).
unsigned dst;
asm( "bfe.u32 %0, %1, 0, %2;" : "=r"( dst ) : "r"( hash ), "r"( nBits ) );
```

- Honestly, it’s not critical for performance here but it’s fun 😊
Speed Comparison with CUSP

C = AA on a Tesla M2090 - fp64

[Bar chart showing speedup compared to CUSP for various benchmarks, with categories such as Default setup, Tuned GMEM size, and Tuned load width.]
IMPROVEMENTS
Memory Consumption

- Global memory is allocated for each warp (to store hash tables)

\[ C = AA \]

C = AA - fp64 code on Tesla M2090 - A: hood.mtx
Load Balancing

- **Objective:** Performance with lower memory requirements
- **Initial approach:** Static scheduling
  ```
  for( ; __syncthreads_or( aRowId < A_nRows ) ; aRowId += nWarps )
  ```
- **Simple load-balancing:**
  ```
  for( ; __syncthreads_or( aRowId < A_nRows ) ; aRowId += getWork( workQueue ) )
  ```
- **getWork** is implemented using a simple atomic operation:
  ```
  __shared__ unsigned work;
  if( threadIdx.x == 0 )
    work = atomicAdd( workQueue, nWarpsPerBlock );
  __syncthreads();
  return work + warpId;
  ```
Load Balancing: Results

- Lower memory usage

C = AA - fp64 code on Tesla M2090 - A: hood.mtx
Load Several Rows of B

- Modify the algorithm to *load several rows of B*
- Hash tables have to use *atomics*

```c
sKeys[warpId][hash] = colB;
if ( sKeys[warpId][hash] == colB ) // Winner?
    atomicAdd( &sVals[warpId][hash], value );
```

- Fp64 `atomicAdd` is implemented using *Compare-and-Swap*
  - See CUDA programming guide
Load Several Rows of B: Results

atmosmodd

mc2depi

1: with atomics -- 1 w/o a: without atomics
Summary

- We have implemented a CPU-like algorithm on the GPU
  - It gives good results compared to CUSP

- Ideas
  - Use hash tables stored in shared and global memories
  - Map a CPU thread to a warp of GPU threads
  - Use simple load-balancing to reduce memory consumption

- Future work
  - Try with better load-balancing
  - Adapt the algorithm to multiple GPUs, then MPI nodes