GPU Sparse Matrix Multiplication with CUDA

Sean Rose

April 29, 2013

1 Introduction

Matrix multiplication is a commonly-used mathematical operation that has many practical applications. It is used to solve a number of problems in a wide variety of fields including science, engineering, and computer science. Given two matrices, A and B, and a resultant matrix C. Matrix multiplication is defined as follows.

$$A = \begin{pmatrix} A_{00} & A_{01} & ... & A_{0m} \\ A_{10} & A_{11} & ... & A_{1m} \\ ... & ... & ... & ... \\ A_{n0} & A_{n1} & ... & A_{nm} \end{pmatrix}$$

$$B = \begin{pmatrix} B_{00} & B_{01} & ... & B_{0p} \\ B_{10} & B_{11} & ... & B_{1p} \\ ... & ... & ... & ... \\ B_{m0} & B_{m1} & ... & B_{mp} \end{pmatrix}$$

$$C = \begin{pmatrix} C_{00} & C_{01} & ... & C_{0p} \\ C_{10} & C_{11} & ... & C_{1p} \\ ... & ... & ... & ... \\ C_{n0} & C_{n1} & ... & C_{np} \end{pmatrix}$$

Where $C_{ij} = \sum_{k=0}^{m} A_{ik} B_{kj}$.

The concept of density is used to describe the number of nonzero elements in a matrix relative to the total number of elements. For an $N \times M$ matrix with Z nonzero elements, the density is defined as $Z/(N * M)$. A sparse matrix is one which has a low density. Sparse matrices can be stored in special formats to eliminate the need for the zero elements to be stored. The storage format and potentially large matrix size presents a challenge when designing an efficient sparse matrix multiplication algorithm.

The rest of this paper is organized as follows. Section 2 describes sparse matrix multiplication and its storage formats. Section 3 describes challenges in implementing an efficient GPU sparse matrix multiplication algorithm. Section 4 describes the solution implementation. Section 5 describes the results of the performance study. Section 6 provides conclusions drawn from the study.

Section 7 provides the paper's references. Finally, section 8 provides additional figures from the performance study.

2 Sparse Matrix Multiplication

Sparse matrix multiplication algorithms aim to leverage the efficient storage formats of sparse matrices for speed. They can calculate the resultant matrix much more quickly than dense sparse matrix multiplication algorithms which attempt process every element in A and B.

2.1 Sparse Matrix Formats

There are three related sparse matrix formats which are commonly used to store sparse matrices. They compress the matrix so the zero valued elements do not need to be stored. The three formats are called COO, CSR, and CSC. These formats use three arrays to store only the nonzero elements of the matrix. One array stores row information and will be referred to as ir. A second array stores column information and it will be denoted by jc. The third array in each case stores the nonzero values in the matrix and it will be referred to as val.

For the purpose of illustrating these storage formats, consider the following matrix with $N = 4$ rows, $M = 4$ columns, and $Z = 6$ nonzero elements.

$$\begin{pmatrix} 0.0 & 0.1 & 0.0 & 0.0 \\ 1.0 & 0.0 & 0.0 & 1.4 \\ 0.0 & 0.0 & 0.0 & 0.0 \\ 4.0 & 4.1 & 0.0 & 4.4 \end{pmatrix}$$

Coordinate (COO) Format

The ir array is of length Z and stores the row for each nonzero element. The jc array is of length Z and stores the column for each nonzero element. The val array is also of length Z and stores the values for each nonzero element. For the above matrix, the elements of the arrays are as follows.

$$ir = \begin{pmatrix} 0 & 1 & 1 & 3 & 3 & 3 \end{pmatrix}$$

$$jc = \begin{pmatrix} 1 & 0 & 3 & 0 & 1 & 3 \end{pmatrix}$$

$$val = \begin{pmatrix} 0.1 & 1.0 & 1.4 & 4.0 & 4.1 & 4.4 \end{pmatrix}$$

Compressed Sparse Row (CSR) Format

CSR format further compresses COO format by limiting the size of ir to $N + 1$. The i^{th} element of the ir array represents the index into jc and val of the first nonzero element of row i. The $(i+1)^{th}$ element of ir is an index into jc and val that is one past the last element in row i. Therefore, $(N+1)^{th}$ element of ir always has the value Z. The jc and val arrays are the same length and purpose as jc and val in COO format. For the above matrix, the elements of the arrays are as follows.

$$ir = \begin{pmatrix} 0 & 1 & 3 & 3 & 6 \end{pmatrix}$$

$$jc = \begin{pmatrix} 1 & 0 & 3 & 0 & 1 & 3 \end{pmatrix}$$

$$val = \begin{pmatrix} 0.1 & 1.0 & 1.4 & 4.0 & 4.1 & 4.4 \end{pmatrix}$$

Compressed Sparse Column (CSC) Format

CSC is similar to CSR format except it uses jc to store indexes into ir and val in the same way that CSR uses ir. The ir and val arrays are the same length and purpose as ir and val in COO format. For the above matrix, the elements of the arrays are as follows.

$$ir = \begin{pmatrix} 1 & 3 & 0 & 3 & 1 & 3 \end{pmatrix}$$

$$jc = \begin{pmatrix} 0 & 2 & 4 & 4 & 6 \end{pmatrix}$$

$$val = \begin{pmatrix} 1.0 & 4.0 & 0.1 & 4.1 & 1.4 & 4.4 \end{pmatrix}$$

2.2 Sequential Algorithm

A simple sequential algorithm using these formats is not much different than a typical sequential algorithm for dense matrix multiplication. The tests for this paper were performed using the SMMP package by Bank and Douglas[4]. As a simple example, consider a matrix B stored in CSC format and matrix A and C stored in CSR format. The algorithm for $A \times B = C$ is as follows.

```
C.ir.append(0);

// For each row in A
for(int rowA = 0; rowA < A.rows; rowA++){
    // Initialize the C.ir end index for the row
    C.ir.append(C.ir[rowA]);

    // For each column in B
    for(int colB = 0; colB < B.cols; colB++){
        float val = 0;
        // For each nonzero in the A row
        for(int i = C.ir[rowA]; i < C.ir[rowA + 1]; i++){
            // For each nonzero in the B column
            for(int j = B.jc[colB]; j < B.jc[colB + 1]; j++){
                // If the A row and B column are the same, add the
                // calculation to the running value.
                if(A.jc[i] == B.ir[j]){
                    val += A.val[i] * B.val[j];
                    break;
                }
            }
        }
```

```
        // Add nonzero calculated values to the C.val and C.jc arrays
        if(val != 0){
            C.jc.append(colB);
            C.val.append(val);
            C.ir[rowA + 1]++;
        }
    }
}
```

3 Challenges of GPU Sparse Matrix Multiplication

3.1 CUDA Concepts

Developed by NVIDIA, CUDA is an extension to C++ that allows for programming the graphics processing unit (GPU) of a system. There are a few important constructs used in this sparse matrix multiplication program that should be described first. Code executing on the GPU is run in a single-instruction multiple-data (SIMD) fashion. Execution is organized into a hierarchy of threads into blocks into a grid. Threads that can be run simultaneously and can synchronize with one another are used to form blocks. Within blocks, threads can be organized in 1, 2, or 3 dimensions. Threads in different blocks generally don't communicate with one another and execute in an exclusive fashion. The blocks can be grouped in 1, 2, or 3 dimensions to form the grid.

Threads can send data to one another through two different types of memory, global and shared. Global memory lies off of the multiprocessor chip and can be accessed by any of the threads running on the device. Global memory has space to store a large amount of data, 1.5 GB for our device, but is very slow to access because of its location. The second memory type, shared memory, can only be accessed by threads within the same block. It lies on the multiprocessor chip and therefore, when compared to global memory, it is faster. However, shared memory is much smaller, just 48 KB for compute capability 2.0 devices.

The GPU program is executed using functions that engage the GPU when called. These functions are called kernels. Launching a kernel is similar to calling any other C++ function except that launch parameters are provided before the function arguments in between <<< and >>>. A function call to launch a kernel looks as follows.

```
GetNNZ<<<numBlocks, numThreads, smSize>>>(A, B, C);
```

The arguments found within the parentheses act as you would expect function arguments to behave. The first launch parameter in between <<< and >>> denotes the number of blocks to be executed on the GPU. This example excludes the second and third dimension block sizes so they default to 1. The

second launch parameter gives the number of threads to execute within each block. Similar to the block parameter, the second and third dimension values default to 1.

The third launch parameter determines the amount of shared memory to allocate for each block. Use of the third launch parameter is referred to as dynamic shared memory allocation. To access the shared memory that is allocated, the following declaration must be added inside the kernel function.

```
extern __shared__ int sharedMem[];
```

Allocation of global memory is done with a call to $cudaMalloc$ which acts very similar to the standard $malloc$ function. Data must be copied between the device and CPU host using the CUDA equivalent of $memcpy$, $cudaMemcpy$.

3.2 GPU Efficiency Concerns

GPU programming has a number of concerns beyond the typical parallelization problems. In CUDA's SIMD model, one instruction is executed simultaneously on up to 32 threads, called a warp. Of particular concern is minimizing the use of control flow branches that evaluate differently for a subset of the threads in a warp. Warp divergence is used to describe a situation when the threads of a warp take different paths after a branch. Warps that diverge will no longer execute in lockstep and will instead execute serially. When this happens, it can significantly impact performance.

An additional concern in achieving efficiency in CUDA programs is the type of memory being accessed. Shared memory accesses are much faster than global memory accesses at 20-40 cycles compared to 400-600 cycles but the amount of shared memory space available is limited. For efficiency, shared memory should be used whenever possible. To reduce delays due to memory access, threads in a warp that write to different memory addresses can have their data transferred simultaneously. The only way to safely write to the same memory location is with atomic operations. However, It is desirable to avoid atomic operations whenever possible because the writes will be forced to be handled serially.

3.3 Initial Study

The code by peer Deshpande [3] was initially studied to provide a code baseline. The individual procedures were timed to identify the best targets for optimization. One key insight gained from observing the run times of Deshpande's code was the unnecessary amount of time spent converting between formats. A goal of this project was to eliminate that step and use CSR format only in order to improve performance.

The use of a single compression format is a conceptual departure from standard matrix multiplication algorithms which typically process a row of matrix A in conjunction with a column of matrix B. The solution would have to perform the calculations in a different order to achieve efficiency. Algorithms of this type

have been published before. A video talk and slides by Demouth [2] was used as inspiration in implementing the solution.

Looking back at the formula for each element in C, $C_{ij} = \sum_{k=0}^{m} A_{ik}B_{kj}$, it is important to notice that the column of each A value has the same index as the row of each B value with which it gets multiplied. This gives us the critical insight that is need to create a solution that uses the same storage format for all of the matrices. As the A values are iterated across each row, their column is used as an index into the rows of B that contain values by which the A value will be multiplied. This technique is illustrated by the following sequential algorithm.

```
int vals[C.cols];

C.ir.append(0);
// For each row in A
for(int rowA = 0; rowA < A.rows; rowA++){
    // Clear out the vals table
    for(int i = 0; i < C.cols; i++){
        vals[i] = 0;
    }

    // For each nonzero value in A
    for(int i = C.ir[rowA]; i < C.ir[rowA + 1]; i++){
        // For each nonzero in the corresponding B row
        for(int j = B.ir[i]; j < B.ir[i + 1]; j++){
            // Add the calculation to the value in the vals table
            vals[B.jc[j]] += A.val[i] * B.val[j];
        }
    }

    // Initialize the end index of the row in C.ir
    C.ir.append(C.ir[rowA]);

    // Move all nonzero values in vals to C.jc and C.val
    for(int i = 0; i < C.cols; i++){
        if(vals[i] != 0){
            C.jc.append(i);
            C.val.append(val);
            C.ir[rowA + 1]++;
        }
    }
}
```

4 Solution Implementation

The solution takes place over three custom kernel launches. The first kernel, *GetNNZ*, calculates the number of nonzero values in each row of the resultant matrix C. *GetVals* is the second kernel that calculates the values for each of the nonzero elements in C. The final kernel is *SortCols* which sorts the columns in each row of C in ascending order. This final step is not necessary, as strictly ordered column values is not a requirement for the storage CSR format, but it provides for easier comparison with the results of other sparse matrix multiplication implementations.

4.1 GetNNZ

```
__global__ void GetNNZ(sparse_matrix A, sparse_matrix B, sparse_matrix C,
        int* workingSet)
{
    const int laneId = threadIdx.x;
    const int warpId = blockIdx.x;
    int* nonzeros;
    int rowAStart, rowAEnd, rowBStart, rowBEnd;
    int nnz;
    int colC;

    extern __shared__ int nzCount[];

    nonzeros = &workingSet[warpId * B.cols];

    // Iterate through each assigned row in A.
    for(int rowA = warpId; rowA < A.rows; rowA += gridDim.x){

        rowAStart = A.ir[rowA];
        rowAEnd = A.ir[rowA + 1];

        // There are no non-zeros in this row so continue
        if(rowAStart == rowAEnd) {
            if (laneId == 0) C.ir[rowA] = 0;
            __syncthreads();
            continue;
        }
```

The kernel uses as many blocks as the system will allow such that there are not more blocks than rows in A. 128 threads are assigned to each block. Each block in the *GetNNZ* kernel is statically assigned one or more rows of A and its corresponding C row.

```
// Reset the nz counts
nzCount[laneId] = 0;

// reset the nonzeros table
for (int i=laneId; i<B.cols; i+= warpSize){
    nonzeros[i] = 0;
}
__syncthreads();

for(int i = rowAStart; i < rowAEnd; ++i){
    rowBStart = B.ir[A.jc[i]];
    rowBEnd = B.ir[A.jc[i]+1];
```

The shared memory for a block of the *GetNNZ* kernel is an array with one spot for each thread in the warp. Each thread in the warp uses its spot to keep track of the number of new nonzero columns found in the current row of C. A *nonzeros* table indexed by column is used to mark columns in the row that have been determined to have a nonzero in them previously. In another inner loop, each thread iterates over each nonzero in the A row in conjunction with one another. The column of the current nonzero A value is used as an index into the row of B with values by which the A value will be multiplied.

```
    for (int j = rowBStart + laneId; j < rowBEnd;
        j += warpSize) {
        colC = B.jc[j];
        nzCount[laneId] += nonzeros[colC] == 0;
        nonzeros[colC] = 1;
    }
    __syncthreads();
}
```

The final inner loop assigns a different nonzero in B to each thread. This is important because it means that each thread will never access the same location in the *nonzeros* array for reading or writing as there will never be two values in the same row of B that are also in the same column. The *nzCount* array is placed in shared memory and each index is used separately by the threads in order to eliminate the need for an atomic increment operation.

```
if(laneId == 0){
    nnz = nzCount[0];
    for(int i = 1; i < warpSize; ++i){
        nnz += nzCount[i];
    }

    C.ir[rowA] = nnz;
```

```
        }
        __syncthreads();
    }
}
```

The final piece of the $GetNNZ$ kernel sums up the nonzero counts calculated by each row and puts the total into the $C.ir$ array. Once all blocks are finished, a call to the thrust library function $maximum$ is used to determine the largest number of nonzero elements in a row of C for use later. Finally, a thrust library call to $exclusive\_scan$ is used to calculate the true $C.ir$ array containing indexes to the nonzero values of each row.

4.2 GetVals

```
__global__ void GetVals(sparse_matrix A, sparse_matrix B,
    sparse_matrix C, int* indexTable)
{
    const int laneId = threadIdx.x;
    const int bloackId = blockIdx.x;

    __shared__ unsigned int back;

    int rowAStart;  // The index into A.jc and A.val
    int rowAEnd;    // The boundary index for A
    float valA;     // The value of the current A nonzero

    int rowBStart;  // The index into B.jc and B.val
    int rowBEnd;    // The boundary index for B
    int colB;       // The current column in B being used

    int rowCStart;  // The index into C.jc and C.val
    int rowCEnd;    // The boundary index for C

    int hash;       // The calculated hash value

    int i, j;       // Loop iterators

    // Set the global hash table to point to the space
    // used by this warp
    int* gColHashTable;
    float* gValHashTable;
    int globalEntries;

    indexTable = &indexTable[C.cols * blockId];

    if(laneId == 0)
```

```
    back = 0;


for(int rowA = blockId; rowA < A.rows; rowA += gridDim.x){
    rowAStart = A.ir[rowA];
    rowAEnd = A.ir[rowA + 1];

    for(i = laneId; i < C.cols; ++i){
        indexTable[i] = -1;
    }
    __syncthreads();
```

The kernel uses as many blocks as the system will allow such that there are not more blocks than rows in A. 32 threads are assigned to each warp to take advantage of the fact that threads in a warp execute in lockstep. Each block in the $GetVals$ kernel is statically assigned one or more rows of A and their corresponding C rows.

```
    // Set the location of the global hash table
    rowCStart = C.ir[rowA];
    rowCEnd = C.ir[rowA + 1];
    globalEntries = rowCEnd - rowCStart;
    gColHashTable = &C.jc[rowCStart];
    gValHashTable = &C.val[rowCStart];

    for(i = rowAStart; i < rowAEnd; ++i){
        valA = A.val[i];

        rowBStart = B.ir[A.jc[i]];
        rowBEnd = B.ir[A.jc[i] + 1];

        int curIdx;
        int* storeInt;
        float* storeFloat;
        float valB;

        for(j = rowBStart + laneId; __any(j < rowBEnd);
            j += warpSize){
```

The same looping method is used in the $GetVals$ kernel as in the $GetNNZ$ kernel. However, the innermost loop is structured to prevent warp divergence from occurring. This is critical for the correctness of the algorithm. The instances of extra complexity in the following code has the purpose of keeping the warp from becoming divergent.

```
                colB = j < rowBEnd ? B.jc[j] : -1;

                curIdx = colB == -1 ? -1 : indexTable[colB];
                hash = colB != -1 && curIdx == -1 ?
                    atomicInc(&back, globalEntries - 1) : curIdx;
                storeInt = hash == -1 ? &hash : &indexTable[colB];
                *storeInt = hash;
```

This part of the kernel looks the column of B up in the index table. If the column has not been given an index into the hash table, it is given the next available slot, denoted by *back*. The *back* index uses *atomicInc* to increment while preventing writing and reading conflicts. The *atomicInc* function does three operations. It first reads the value in *back*. Then, it does $back = back >= (globalEntries - 1)?0 : back + 1$. The value of *back* as it was initially read is then returned by *atomicInc*. It does these three operations atomically. Note that because *atomicInc* should be called *globalEntries* times, one for each nonzero column. As a result, *back* gets reset back to 0 automatically by *atomicInc* when the last nonzero in the C row is added.

```
                storeInt = hash == -1 ? &colB : &gColHashTable[hash];
                *storeInt = colB;

                valB = colB == -1 ? 1 : B.val[j];
                storeFloat = hash == -1 ? &valA : &gValHashTable[hash];
                *storeFloat += valB * valA;

            }
        } // For each nonzero in the A row
    } // For each assigned row in A
}
```

The final part of this kernel stores the column and value in $C.jc$ and $C.val$. For threads that are not doing any useful work because we are trying to keep them from diverging, this has no effect.

4.3 SortCols

The *SortCols* kernel implements a radix sort over the separate rows in the $C.jc$ array, moving the values in the $C.val$ array to match the sorted column positions. The algorithm tries to maximize the number of blocks that can be used. It uses a number of threads based on *maxRowNNZ* which is the number of nonzero values in the most dense row of C.

```
__global__ void SortCols(sparse_matrix C, int maxRowNNZ, int* workQueue)
{
    const int laneId = threadIdx.x;
```

```
const int blockId = blockIdx.x;

// Dynamic shared memory
extern __shared__ int sharedMem[];

// The maximum size of the queue
const int queueSize = (maxRowNNZ / 2) + 1;

// The maximum number of passes needed
int maxShift = __log2f(C.cols) / RADIX_BITS;

// The number of passes for the work in the queue
int* workPasses = &workQueue[blockId * queueSize];
// The front of the bucket for the work in the queue
int* workFronts = &workQueue[gridDim.x * queueSize];
workFronts = &workFronts[blockId * queueSize];
// The back of the bucket for the work in the queue
int* workBacks = &workQueue[gridDim.x * queueSize * 2];
workBacks = &workBacks[blockId * queueSize];

int front;                      // The front of the work queue.
__shared__ unsigned int back;   // The back of the work queue.

// Holds the sizes for the buckets being sorted by the threads
int* bucketSizes = &sharedMem[laneId * RADIX_BASE];
// The ending index of the buckets being sorted
int* bucketBounds = &sharedMem[blockDim.x * RADIX_BASE];
bucketBounds = &bucketBounds[laneId * RADIX_BASE];

int pass;           // The pass number of the current bucket
int bucketFront;    // The index of the front of the bucket
int bucketBack;     // The index of the back of the bucket
int bucketIdx;      // The index of an item in the bucket
int shiftCount;     // The number of bits to shift to get the index
int iTmp;           // A temporary variable for swapping
float fTmp;
int swapIdx;        // The index to swap with
int queueIdx;       // An index into the work queue
int prev;           // The previous bucket offset
int subIdx;

for(int rowC = blockId; rowC < C.rows; rowC += gridDim.x){
    // Skip if there are not non-zeros to sort
    if(C.ir[rowC] == C.ir[rowC + 1])
        continue;
```

```
// Clear the work queue
for(int i = laneId + 1; i < queueSize; i += blockDim.x){
    workPasses[i] = -1;
}

workPasses[0] = 0;
workFronts[0] = C.ir[rowC];
workBacks[0] = C.ir[rowC + 1];

front = 0;
back = 1;

__syncthreads();
```

Each block is statically assigned one more more rows in C. To begin processing the row, the first item is placed into the work queue. The work queue is described next.

```
// While there is more work in the queue
while(front != back){
    queueIdx = (front + laneId) % queueSize;
    // Get the work
    pass = workPasses[queueIdx];
    bucketFront = workFronts[queueIdx];
    bucketBack = workBacks[queueIdx];

    // Clear this work
    workPasses[queueIdx] = -1;

    // Move the front forward
    if((back > front && back - front <= blockDim.x)
        || (back < front && (back + queueSize) - front
        <= blockDim.x)){
        front = back;
    }
    else{
        front = (front + blockDim.x) % queueSize;
    }
```

The algorithm uses a work queue to allocate work for the threads. The queue is implemented as circular buffer. Each item in the work queue consists of the number of passes performed up to that point, the starting position of the items in the bucket, and the ending point of the items in the bucket.

```
// There is work to do
if(pass >= 0){

    // Clear the bucket sizes
    for(int i = 0; i < RADIX_BASE; ++i){
        bucketSizes[i] = 0;
    }

    shiftCount = (maxShift - pass) * RADIX_BITS;

    // First, determine the size of the buckets
    for(int i = bucketFront; i < bucketBack; ++i){
        ++bucketSizes[(C.jc[i] >> shiftCount) & RADIX_MASK];
    }
```

After retrieving the work to be done, the first sorting step is to determine the number of elements in each bucket. This is necessary to allow the sort to occur in place. Without this step, the amount of memory needed to perform the sort would be larger by a factor of $maxRowNNZ$.

```
    // Determine the indexes of the buckets and put
    // them into the work queue
    prev = bucketFront;
    for(int i = 0; i < RADIX_BASE; ++i){
        // Determine the bucket end
        bucketIdx = bucketSizes[i] + prev;

        // Place the bucket into the work queue only
        // if it has items to be sorted
        if(bucketSizes[i] > 1){
            queueIdx = atomicInc(&back, queueSize - 1);

            workPasses[queueIdx] = pass + 1;
            workFronts[queueIdx] = prev;
            workBacks[queueIdx] = bucketIdx;
        }

        // Store the bucket end
        bucketSizes[i] = bucketIdx;
        bucketBounds[i] = bucketIdx;
        prev = bucketIdx;
    }
```

After determining the size of each bucket, another loop calculates the in-place index for the end of each bucket. Additionally, the new buckets that have unsorted elements are placed into the work queue to be handled later.

```
                // Place the items into the buckets
                bucketIdx = bucketFront;
                while(bucketIdx != bucketBack) {
                    subIdx = (C.jc[bucketIdx] >> shiftCount) & RADIX_MASK;
                    swapIdx = --bucketSizes[subIdx];;

                    // Done sorting this bucket, move to the next open one
                    if(swapIdx == bucketIdx){
                        do {
                            bucketIdx = bucketBounds[subIdx++];
                        } while(bucketIdx != bucketBack
                            && bucketSizes[subIdx] == bucketIdx);
                    }
                    else{
                        // Swap swapIdx and bucketIdx
                        iTmp = C.jc[swapIdx];
                        C.jc[swapIdx] = C.jc[bucketIdx];
                        C.jc[bucketIdx] = iTmp;
                        fTmp = C.val[swapIdx];
                        C.val[swapIdx] = C.val[bucketIdx];
                        C.val[bucketIdx] = fTmp;
                    }
                }
            }   // If this thread has work
            __syncthreads();

        }   // While there is work to do
    }   // For all rows in C
}
```

The final step is to place each of the items into their respective buckets. The buckets grow from the end to the beginning as the elements are placed into them.

5 Performance Study

As a comparison of progress from the algorithm by Deshpande and this solution, times were compared between the GetNNZ function and Deshpande's equivalent. The time taken to load and convert format for Deshpande's algorithm were also compared to the time it takes just to load the matrix, which is all that needs to be done for the algorithm in this solution.

The time needed to convert the matrix formats was found to be quite significant. The solution's elimination of that step has shown to reduce the run time significantly as illustrated in *Figure* **??**. The techniques used in this algo-

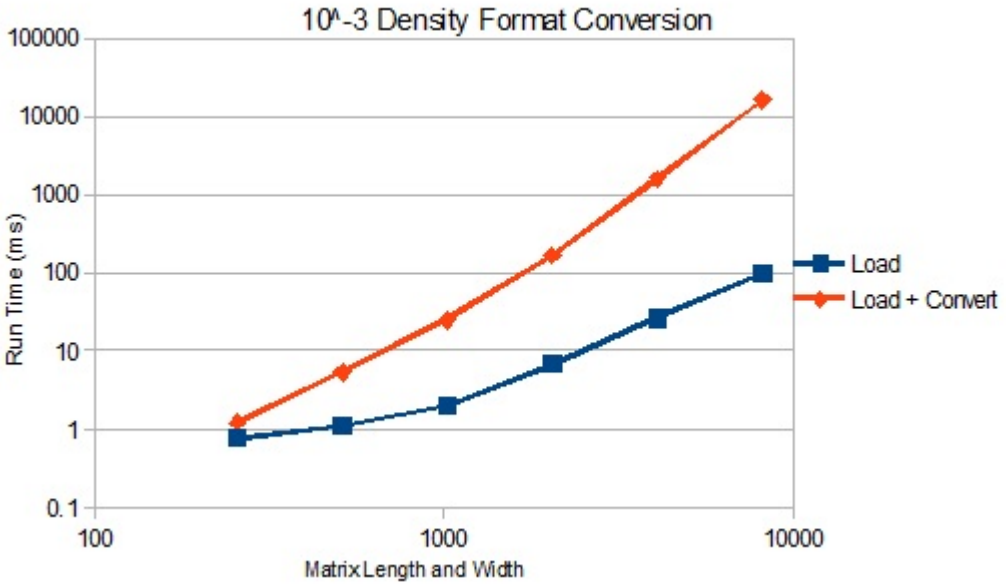

Figure 1: 10^{-3} Density Format Conversion Speed Comparison

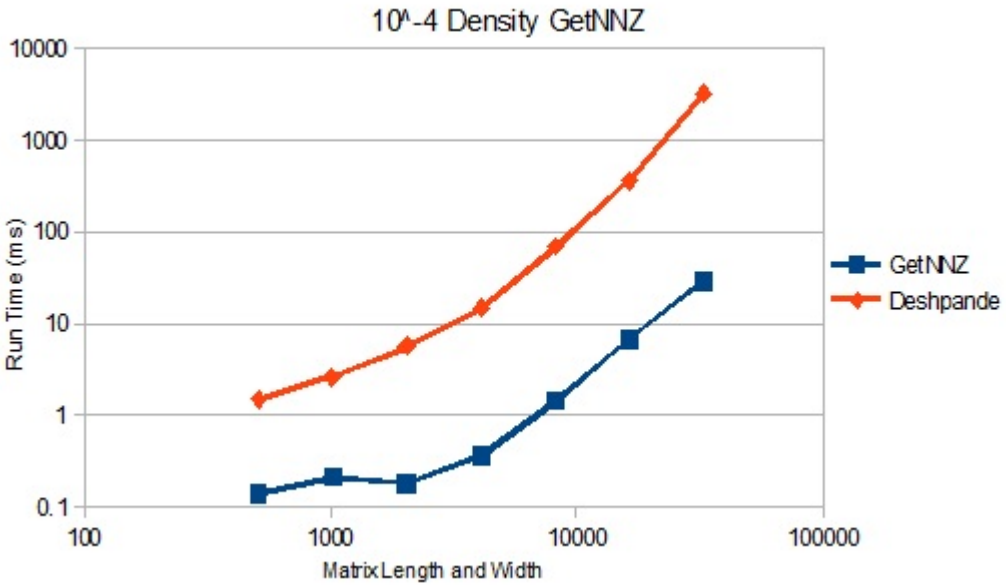

Figure 2: 10^{-4} Density GetNNZ Speed Comparison

rithm have also shown a large improvement over Deshpande's implementation. A significant relative speed up can be seen in *Figure* **??**.

For comparison of the entire calculation algorithm, run times were compared between the solution, Matlab, and the SMMP package by Bank and Douglas[4]. The solution run times in the were obtained by excluding the SortCols part of the algorithm as it is not necessary for the comparison.

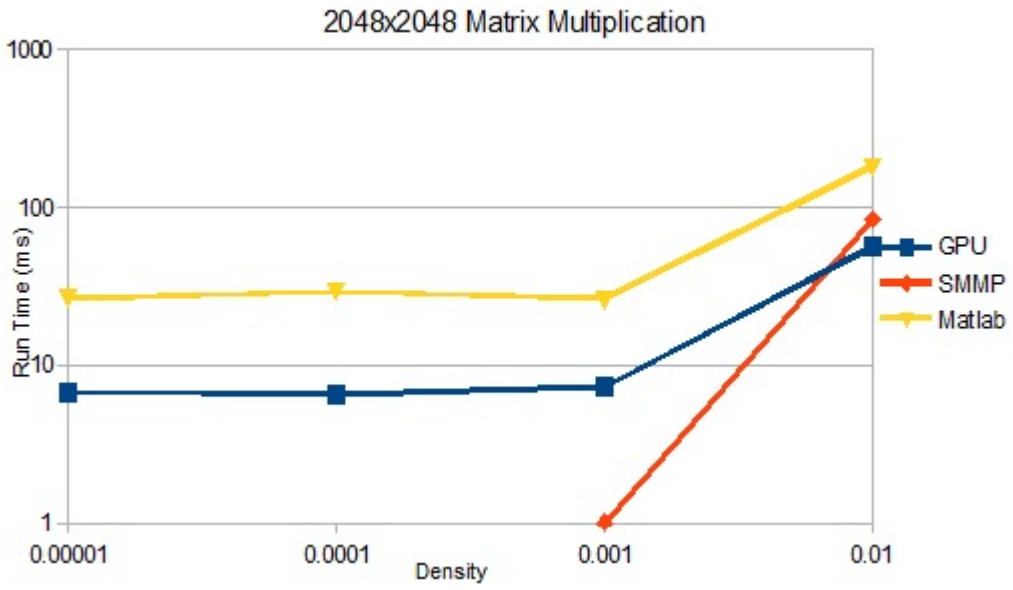

Figure 3: Small MM Speed Comparison

The solution runs relatively quickly for small matrices when compared to Matlab. This is illustrated in *Figure* **??**. Also, for more dense matrices as shown in *Figure* **??**, the solution runs more quickly than Matlab in all cases and more quickly than SMMP for larger matrices.

As the matrices become less dense, the solution starts to fare worse and worse. The run times for the solution cross above the Matlab run times in the middle of the 10^{-4} density tests as seen in *Figure* **??**. Illustrated in *Figure* **??**, the solution fares very poorly while calculating the larger matrices tested during the 10^{-5} density tests.

6 Conclusion

There are several avenues for improvement for the solution. One specific example is of the *GetVals indexTable* and *GetNNZ nonzeros* table. Every element in the table is reset after each C row iteration but this is inefficient for large

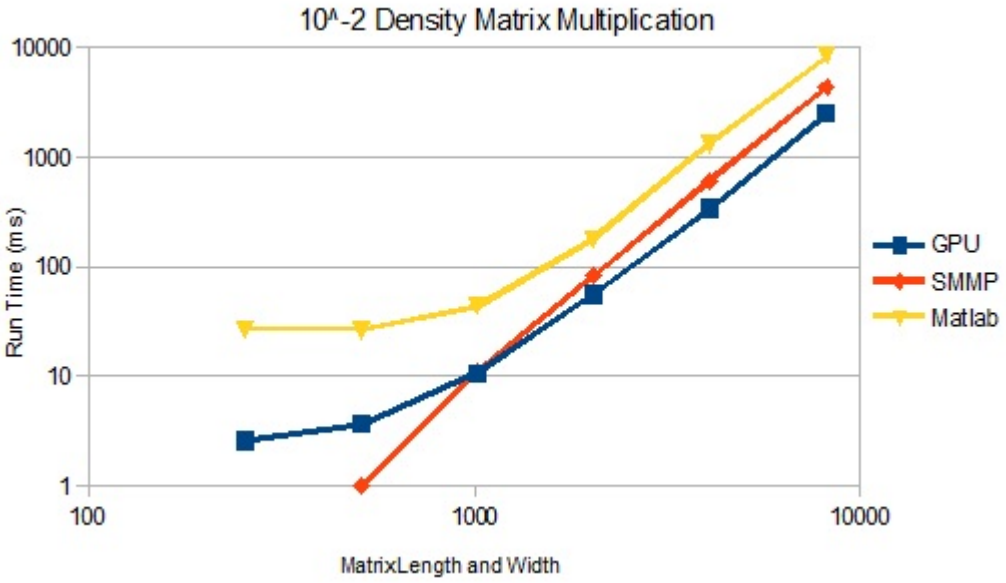

Figure 4: High Density MM Speed Comparison

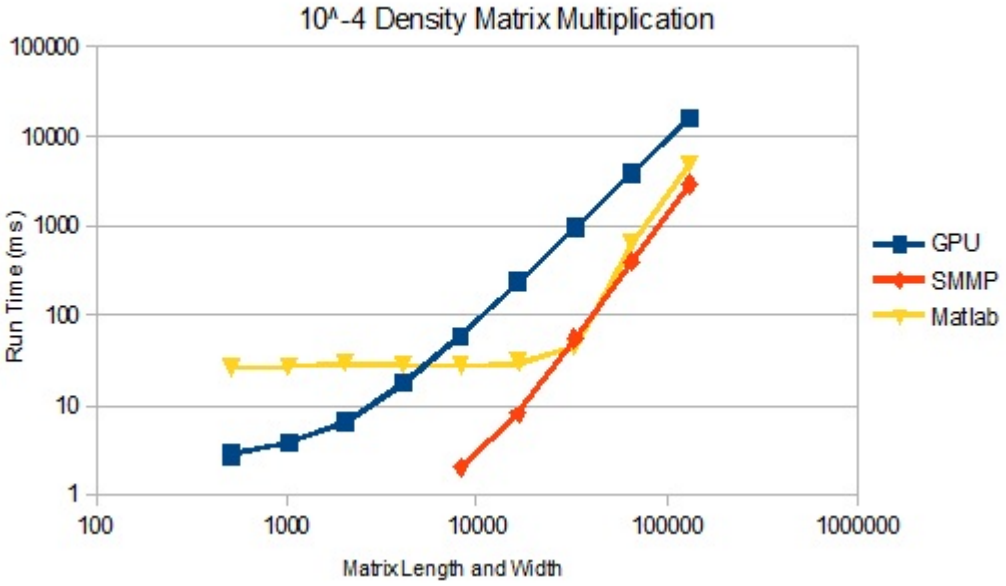

Figure 5: Medium Density MM Speed Comparison

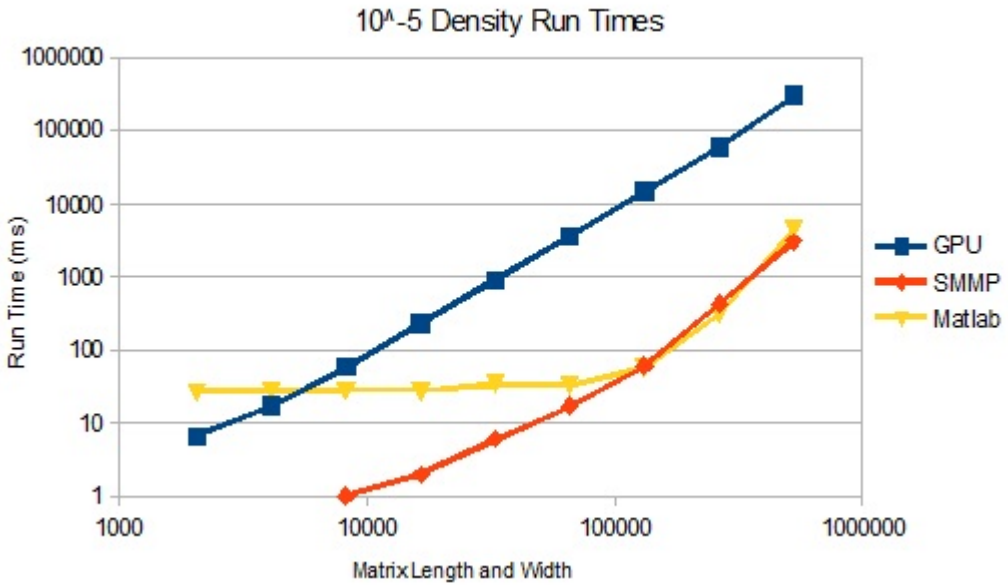

Figure 6: Low Density MM Speed Comparison

and sparse matrices. The *GetVals* function can use the *gColHashTable* to index back into the *indexTable* and reduce the number of iterations needed in the reset loop.

Additionally, there are not many active threads in the solution kernels for very sparse matrices because there is likely to be few nonzero values in each row. One could reduce the number of idle threads by assigning more than one row for the block's threads to handle simultaneously. Also, more tests can be done to identify the ideal balance between the number of blocks and the number of threads for further improvement.

Another area of interest for improving the algorithm is in identifying a better way to handle how the rows get distributed among the blocks. An alternative to assigning a static number of rows per block could be that each block claims the next row or rows that have not been processed, improving performance in non-uniformly sparse matrices.

The solution is an improvement over the baseline code implemented by Deshpande. For smaller and more dense matrices, the solution does fairly well compared to Matlab and SMMP. With the improvements described above, the solution has the potential to be more competitive with Matlab and SMMP for larger matrices as well.

7 References

[1] NVIDIA, CUDA C Programming Guide, April 2013, `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.
[2] Julien Demouth, Optimization of a Sparse Matrix-Matrix Multiplication on the GPU, GPU Technology Conference, 2012.
[3] Abhishek Deshpande, Sparse Matrix Multiplication using CUDA and Mex Interface, 2012.
[4] Randolph E. Bank, Craig C. Douglas, Sparse matrix multiplication package (SMMP). Advances in Computational Mathematics 1993, Volume 1 Issue 1, 127-137.
[5] Aydin Buluc, John Gilbert, "Parallel Sparse Matrix-Matrix Multiplication and Indexing: Implementation and Experiments," SIAM J. Sci. Computing, 34(4), 170-191, 2012.

8 Additional Figures

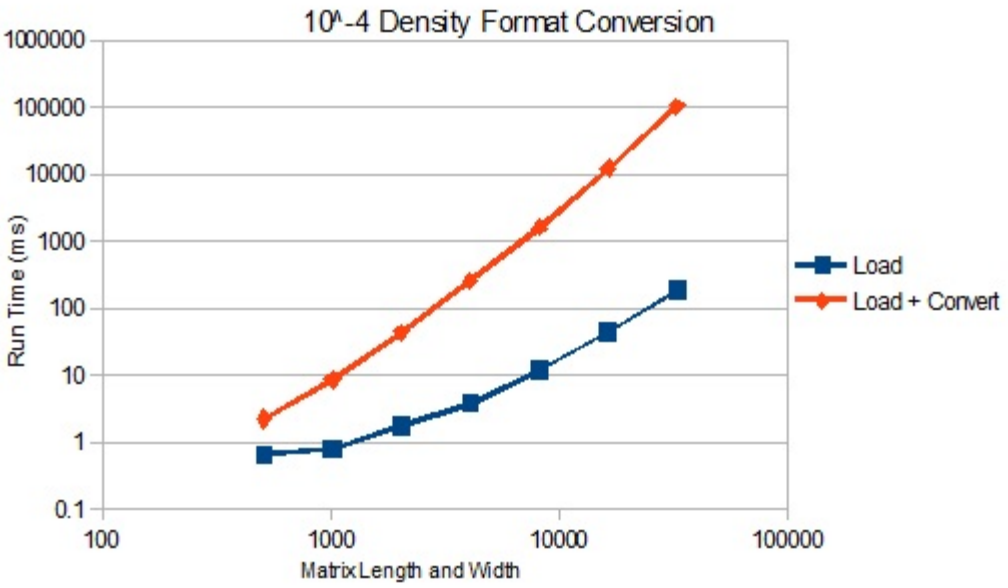

Figure 7: 10^{-4} Density Format Conversion Speed Comparison

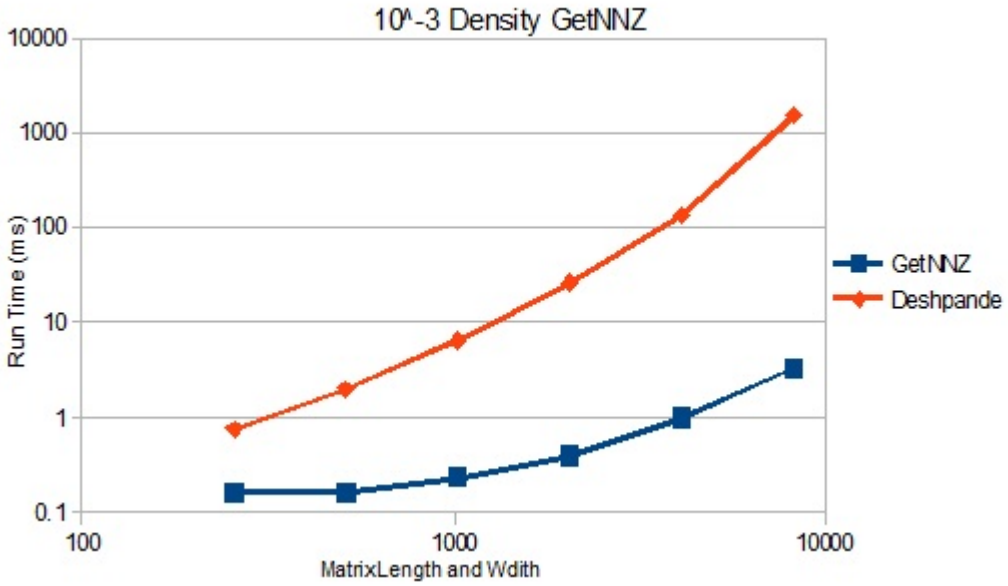

Figure 8: 10^{-3} Density GetNNZ Speed Comparison

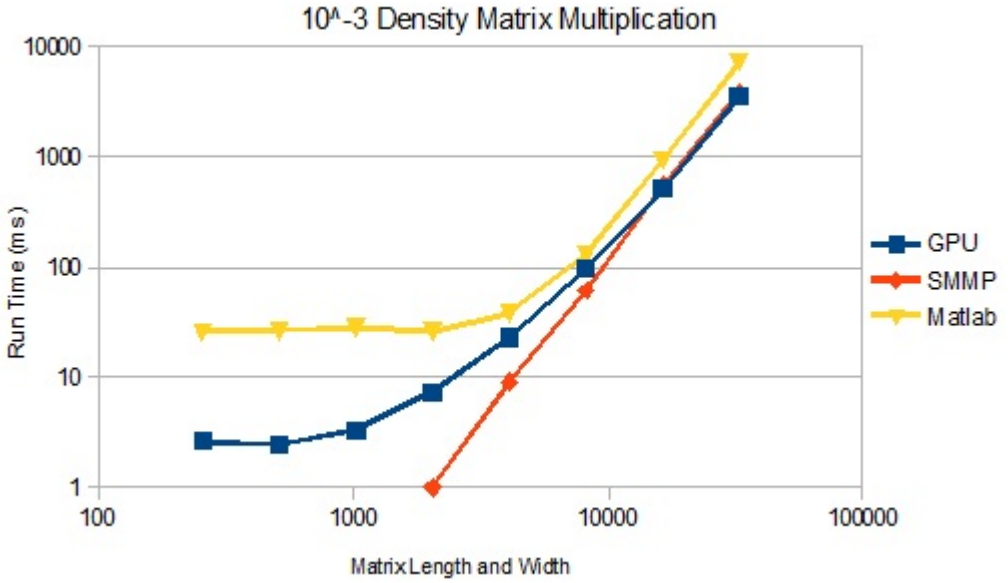

Figure 9: 10^{-3} Density Matrix Multiplication

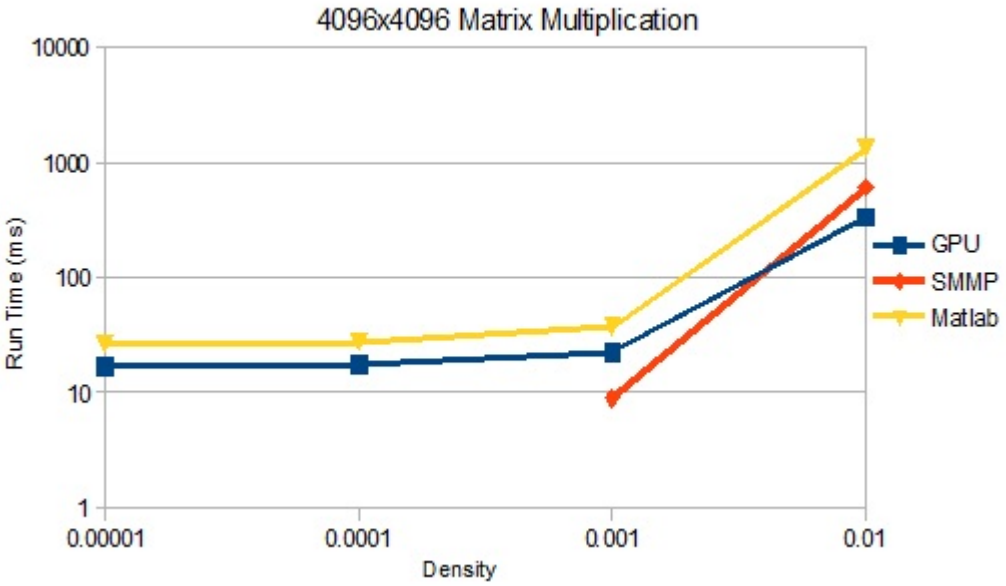

Figure 10: 4096×4096 Matrix Multiplication

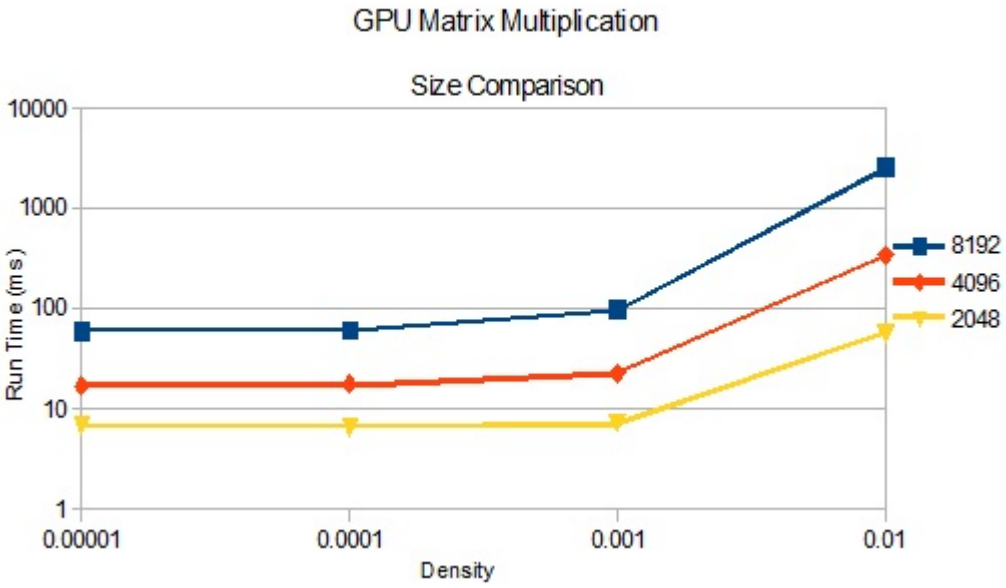

Figure 11: GPU MM Size Comparison