Sparse Matrix Vector Multiplication on GPUs: Implementation and analysis of five algorithms

Pooja Hiranandani
CIMS, NYU
ph1130@nyu.edu

***Abstract*— Sparse matrix vector multiplication (SpMV) is of significant importance to computing in a number of scientific and engineering disciplines. However due to the arbitrary sparsity patterns and sizes of sparse matrices, the parellisation of SpMV is still beset with many operational issues including poor memory coalescing, thread divergence and load imbalance. For my project, I implement five different GPU based algorithms and analyze their performance on different types and sizes of data. I try to draw out insights on the strengths and weaknesses of these algorithms and the situations in which they are best used. I look at performance in terms of computational throughput, memory bandwidth utilization and other system generated metrics.**

I. INTRODUCTION

Sparse matrix vector multiplication (SpMV) comprises an important part of scientific applications in a number of fields including linear algebra [1], data mining [2], bioinformatics [3], engineering and graph analytics [4]. It represents the dominant cost in many iterative methods for solving large scale linear systems and eigenvalue problems which arise in these applications [5]. As such, optimization of its workings is of significant importance to the state of scientific computing. Graphics Processing Units (GPUs), with their emphasis on throughput, have been employed successfully in implementing multiplication of dense matrices which are particularly suited to the parellelisation required of applications running on the GPU. On the other hand, matrix-vector multiplication is known to be memory-bound with $O(n^2)$ operations on $O(n^2)$ amount of data [6], with only little data reuse. This implies that matrix-vector multiplication has a high ratio of memory accesses to arithmetic operations, and also has very restricted data locality. Such common characteristics significantly degenerate the performance of matrix-vector multiplications, regardless of dense or sparse matrices. In particular, for sparse matrices, the situation becomes more severe since a sparse matrix merely has a small proportion of non-zero elements and these non-zeros are often distributed in irregular structures [7].

II. BACKGROUND INFORMATION

Sparse matrices are stored in a variety of formats designed to represent the information contained in them in either the most economical way possible or in a way as to facilitate operations such as SpMV on them. A few of the popular formats are described below:

Diagonal format: The diagonal format is formed by two arrays: data, which stores the nonzero values, and offsets, which stores the offset of each diagonal from the main diagonal. By convention, the main diagonal corresponds to offset 0, while $i > 0$ represents the i-th super-diagonal and $i < 0$ the i-th sub-diagonal. The diagonal format excels when the nonzero values are restricted to a small number of matrix diagonals. However this severely restricts the scope of its usage and the format is appropriate only for a small subset of matrices such as the application of stencils to regular grids, a common discretization method. [5]

ELLPACK: For an M-by-N matrix with a maximum of K nonzeros per row, the ELLPACK format stores the nonzero values in a dense M-by-K array data, where rows with fewer than K nonzeros are zero-padded. Similarly, the corresponding column indices are stored in indices, again with zero, or other some sentinel value used for padding [5]. The ELLPACK format is good for matrices where the average number of nonzero values per row do not vary a lot else a lot of space is wasted in padding short rows.

Coordinate Format (COO): The arrays of COO are row, col, and data which store the row indices, column indices, and values, respectively, of the nonzero matrix entries. COO is a general sparse matrix representation since, for an arbitrary sparsity pattern, the required storage is always proportional to the number of non zeros. The main weakness of this format is that it is the least economical of all the formats described because it requires three arrays that are each the size of the number of nonzero elements in the matrix.

Compressed Sparse Row Format (CSR): The CSR format explicitly stores column indices and nonzero values in arrays indices and data. A third array of row pointers, ptr, takes the CSR representation. For an M-by-N matrix, ptr has length M + 1 and stores the offset into the i-th row in ptr[i]. The last entry in ptr, which would otherwise correspond to the (M + 1)-st row, stores NNZ, the number of non zeros, in the matrix. [5] Given that the CSR format can be used to store arbitrarily sparse matrices and is very efficient in terms of storage space, it is the most popular sparse matrix storage format in usage today. Since the focus of my project is CSR based SpMV and understanding its format is crucial to understanding the algorithms presented in this paper, a brief illustration of the format is presented below:

$$\text{Matrix} = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 0 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

$$\text{ptr} = \begin{bmatrix} 0 & 2 & 3 & 6 \end{bmatrix}$$

$$\text{cols} = \begin{bmatrix} 0 & 2 & 2 & 0 & 1 & 2 \end{bmatrix}$$

$$\text{vals} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix}$$

III. LITERATURE REVIEW

SpMV has been a frequent topic of research in the past decade owing to its ubiquity in scientific computing. The earliest variants of GPU-based CSR algorithms for SpMV were proposed by Bell and Garland [8]. They were called CSR Scalar and CSR Vector. CSR Scalar assigns one row to every thread while CSR Vector assigns one row to every warp and makes use of shared memory to store elements of the input array which are then summed up. Lu et al. [9] optimize CSR Scalar by padding CSR arrays and achieve 30% improvement of the memory access performance. Dehnavi et al. [10] propose a prefetch-CSR method that partitions the matrix nonzeros to blocks of the same size and distributes them amongst GPU resources. This method obtains a slightly better behavior than CSR Vector by padding rows with zeros to increase data regularity, using parallel reduction techniques, and prefetching data to hide global memory accesses. Furthermore, Dehnavi et al. enhance the performance of the prefetch-CSR method by replacing it with three subkernels [11], [12]. Greathouse and Daga [12] propose CSR Adaptive which lets small rows be handled by CSR Stream - an algorithm that streams input data into shared memory - and lets CSR Vector handle very large rows. PCSR, proposed by He and Gao [13], divides up the work between two kernels - one that multiplies elements of the input matrix with their corresponding vector elements and the other that reduces them. Liu and Schmidt [7] suggest dynamically distributing rows within a warp. They propose two variants of this dynamic distribution - vector level dynamic row distribution and warp level dynamic row distribution. They make use of shuffle instructions and atomic operations to assist in this dynamic allocation.

In addition to the CSR based GPU algorithms, there have been advances to explore algorithms using other storage formats. Bell and Garland presented some of the first work in this direction when they showed that the column-major ELLPACK format presented much better performance for matrices with short rows [8]. They further described a hybrid ELLPACK + COO format that remedied the difficult problems of using ELLPACK on long rows or in matrices with variable NNZ/row [12]. ELLPACK-R was developed as an improvement upon ELLPACK by adding a row length array to the mix [14]. Ashari et. al. [15] presented a new Blocked Row Column (BRC) storage format with a two-dimensional blocking mechanism that reduces thread divergence by reordering and blocking rows of the input matrix with nearly equal number of non-zero elements onto the same execution units (i.e., warps). BRC improves load balance by partitioning rows into blocks with a constant number of non-zeros such that different warps perform the same amount of work. Examples of other formats include BCSR [16], BELLPACK [16], sliced ELL [17], Cocktail [18], JAD [19], BCCOO [20], CSR5 [21], BRO [22] and AMB [23].

The problem with these approaches is that CSR is currently the most popular format for representation of sparse matrices and using any other format involves significant computational overheads in recasting the data into the new format - this cost sometimes outweighing any benefits derived from operating in the new formats. Thus it is considered prudent to invest efforts into studying and developing algorithms that optimize CSR-based SpMV and my project focuses on CSR-based SpMV for this reason.

IV. SOLUTION

In order to understand the strengths and weaknesses of different approaches to conducting CSR-based SpMV on GPU, I implemented five previously published algorithms with some modifications and analyzed their performance on 15 different types and sizes of data.

Described below are the details of the algorithms implemented:

A. CSR Scalar

In CSR Scalar [8], each thread is responsible for the multiplication of one row of the input matrix with the input vector. Individual threads load and store all data related to their rows, in addition to performing the arithmetic operations. Thus the number of blocks launched by the kernel is equal to the number of rows divided by the block dimensions chosen even though my implementation allows for a fewer number of blocks with each thread then being responsible for more than one row. There is no usage of shared memory in this algorithm.

Implementation

```
template <typename T>
__global__ void spmv_csr_scalar_kernel(T *
    d_val,T * d_vector,int * d_cols,int *
    d_ptr,int N, T * d_out)
{
   int tid = blockIdx.x * blockDim.x +
       threadIdx.x;

   for (int i = tid; i < N; i += blockDim.x
       * gridDim.x) {
      T sum = 0;
      int start = d_ptr[i];
      int end = d_ptr[i + 1];
      for (int j = start; j < end; j++){
         int col = d_cols[j];
         sum += d_val[j] * d_vector[col];
      }
      d_out[i] = sum;
   }
}
```

B. CSR Vector

In CSR Vector [8], each block is divided into warp sized (32 threads in current day NVIDIA GPUs) portions to which rows are distributed. Thus 32 threads take care of a single row. The number of blocks launched by the kernel is equal to the number of rows divided by the block dimension divided by the warp size. Shared memory of the same size as the block dimensions is used to store the multiplication products of the elements of the input matrix and input vector accessed by the warps. Each warp is allocated a warp sized space in the shared memory which is accumulated into iteratively. After all the elements allocated to a block are stored in shared memory, the warps do a partial reduction for their rows until the first thread of each warp writes the result into the output vector.

Implementation

```
template <typename T>
__global__ void spmv_csr_vector_kernel(T *
    d_val,T * d_vector,int * d_cols,int *
    d_ptr,int N, T * d_out)
{
   int t = threadIdx.x; // Thread ID in block
   int lane = t & (warpSize-1); // Thread ID
       in warp
   int warpsPerBlock = blockDim.x /
       warpSize; // Number of warps per block
   int row = (blockIdx.x * warpsPerBlock) +
       (t / warpSize); // One row per warp

   __shared__ volatile T vals[BlockDim];

   if (row < N) {
      int rowStart = d_ptr[row];
      int rowEnd = d_ptr[row+1];
      T sum = 0;

      // Use all threads in a warp
          accumulate multiplied elements
      for (int j = rowStart + lane; j <
          rowEnd; j += warpSize) {
         int col = d_cols[j];
         sum += d_val[j] * d_vector[col];
      }
      vals[t] = sum;
    __syncthreads();

      // Reduce partial sums
      if (lane < 16) vals[t] += vals[t + 16];
      if (lane < 8) vals[t] += vals[t + 8];
      if (lane < 4) vals[t] += vals[t + 4];
      if (lane < 2) vals[t] += vals[t + 2];
      if (lane < 1) vals[t] += vals[t + 1];
    __syncthreads();

      // Write result
      if (lane == 0)
         d_out[row] = vals[t];
   }
}
```

C. CSR Adaptive

CSR adaptive [12] consists of a GPU kernel and a CPU procedure. The CPU procedure divides the matrix into row-blocks, which is to say that it creates an array of pointers that point to the starting and ending row numbers of all consecutive rows whose elements are less than or equal to a certain block size. If a row is too large to fit into the desired block size it is allotted one full block. Thus for eg., if the block size is 64 and row 1 has 20 elements, row 2 has 21 elements and row 3 has 44 elements, row 1 and row 2 will be allotted to row-block 1 while row 3 will be allotted to row-block 2. The GPU kernel allocates a shared memory space exactly equal to the block dimensions. Each thread then loads one element of the input matrix and input vector, multiplies them and stores the product into shared memory. Finally the products for each row are summed up by one thread per row. The number of blocks launched by the kernel is equal to the number of row blocks as calculated by the CPU procedure. If the size of a row is larger than the size of the shared memory allocated and is therefore in a row-block of its own, the algorithm uses CSR Vector to process that row block. In my implementation of CSR Vector for this algorithm, instead of using only one warp to process the row, all threads of the block are used to process the large row. As discussed in the Results section to follow, this resulted in a big spike in performance for matrices with very large rows, making CSR Adaptive the best performing algorithm for such matrices.

Implementation

```
template <typename T>
__global__ void spmv_csr_adaptive_kernel(T *
    d_val,T * d_vector,int * d_cols,int *
    d_ptr,int N, int * d_rowBlocks, T *
    d_out)
{
   int startRow = d_rowBlocks[blockIdx.x];
   int nextStartRow = d_rowBlocks[blockIdx.x
       + 1];
   int num_rows = nextStartRow - startRow;
   int i = threadIdx.x;
   __shared__ volatile T LDS[1024];
   // If the block consists of more than one
       row then run CSR Stream
   if (num_rows > 1) {
      int nnz = d_ptr[nextStartRow] -
          d_ptr[startRow];
      int first_col = d_ptr[startRow];

      // Each thread writes to shared memory
      if (i < nnz)
         LDS[i] = d_val[first_col + i] *
             d_vector[d_cols[first_col + i]];
      __syncthreads();

      // Threads that fall within a range
          sum up the multiplied results
      for (int k = startRow + i; k <
          nextStartRow; k += blockDim.x){
         T temp = 0;
         for (int j= (d_ptr[k] - first_col);
```

```
            j < (d_ptr[k + 1] - first_col);
            j++)
          temp = temp + LDS[j];
        d_out[k] = temp;
    }
  }
  // If the block consists of only one row
     then run CSR Vector
  else { CSR_Vector() }
```

D. Perfect CSR (PCSR)

PCSR [13] consists of two kernels that run synchronously. The first kernel loads elements of the input matrix and the input vector, multiplies them and stores the result into an array in the global memory. The number of blocks launched by this kernel is equal to the number of non zero elements divided by the block dimensions. The second kernel allocates shared memory to store small portions of two arrays - the row delimiters array component of the CSR format and the multiplied results array calculated by the first kernel. Thus each block has two shared memory arrays. This kernel is mainly composed of the following three stages:
(i) In the first stage, the row delimiters array, ptr, stored in global memory is piece-wise assembled into shared memory ptr_s of each thread block in parallel. Each thread for a thread block is only responsible for loading an element value of ptr into ptr_s except for thread 0, which is used to load a second element into the last ptr_s array position
(ii) The second stage loads element values of the multiplied results in global memory from the position ptr_s[0] to the position ptr_s[BlockDim] into shared memory, v_s.
(iii) The third stage accumulates element values of v_s.

In my implementation, thread 0 performs boundary checking before updating the last ptr_s array position to enable processing of matrices whose number of rows are not perfectly aligned to block dimensions. The number of blocks launched by this kernel is equal to the number of rows divided by the block dimension.

Implementation

```
template <typename T>
__global__ void spmv_pcsr_kernel1(T *
    d_val,T * d_vector,int * d_cols,int
    d_nnz, T * d_v)
{
   int tid = blockIdx.x * blockDim.x +
       threadIdx.x;
   int icr = blockDim.x * gridDim.x;
   while (tid < d_nnz){
      d_v[tid] = d_val[tid] *
          d_vector[d_cols[tid]];
      tid += icr;
   }
}


template <typename T>
__global__ void spmv_pcsr_kernel2(T *
    d_v,int * d_ptr,int N,T * d_out)
{
   int gid = blockIdx.x * blockDim.x +
       threadIdx.x;
   int tid = threadIdx.x;

   __shared__ volatile int
       ptr_s[threadsPerBlock + 1];
   __shared__ volatile T
       v_s[sizeSharedMemory];

   // Load ptr into the shared memory ptr_s
   ptr_s[tid] = d_ptr[gid];
   if (tid == 0) {
      if (gid + threadsPerBlock > N) {
         ptr_s[threadsPerBlock] = d_ptr[N];}
      else {
         ptr_s[threadsPerBlock] = d_ptr[gid
             + threadsPerBlock];}
   }
   __syncthreads();
   int temp = (ptr_s[threadsPerBlock] -
       ptr_s[0])/threadsPerBlock + 1;
   int nlen = min(temp *
       threadsPerBlock,sizeSharedMemory);
   T sum = 0;
   int maxlen = ptr_s[threadsPerBlock];
   for (int i = ptr_s[0]; i < maxlen; i +=
       nlen){
      int index = i + tid;
      __syncthreads();
      // Load d_v into the shared memory v_s
      for (int j = 0; j <
          nlen/threadsPerBlock;j++){
         if (index < maxlen) {
            v_s[tid + j * threadsPerBlock] =
                d_v[index];
            index += threadsPerBlock;
         }
      }
      __syncthreads();

      // Sum up the elements of a row
      if (!(ptr_s[tid+1] <= i || ptr_s[tid]
          > i + nlen - 1)) {
         int row_s = max(ptr_s[tid] - i, 0);
         int row_e = min(ptr_s[tid+1] -i,
             nlen);
         for (int j = row_s;j < row_e;j++){
            sum += v_s[j];
         }
      }
   }
   d_out[gid] = sum;
}
```

E. LightSpMV

LightSpMV dynamically distributes rows to subsections of warps called vector lanes. The number of vector lanes each warp should be divided into is decided by the host code which launches different versions of the kernel depending on the average number of nonzero elements per row of the input matrix. Thus each row is handled by one vector. The row number that each vector lane should handle is passed on to it through a shuffle instruction. A shuffle instruction permits exchanging of a variable between threads within a warp without the use of shared memory. The exchange occurs

simultaneously for all active threads within the warp, moving 4 or 8 bytes of data per thread depending on the type [24]. The shuffle instruction is faster than shared memory since it only requires one instruction versus three for shared memory (write, synchronize, read) [25]. The core of the kernel is the dynamic row retrieval and broadcasting over vectors. This approach relies on a global row management (GRM) data structure, which manages the distribution of rows by accepting new requests for row indices from vectors and responding with the corresponding values. GRM contains an integer-type variable row_counter located in global memory, which is reset to zero before launching the kernel and represents the lowest row index for all unprocessed matrix rows. When a request comes, GRM atomically increases row_counter by one using the atomicAdd atomic function and then responds to the request with the old value. Each vector multiplies elements of the input matrix with the input vector. A scalar reduction is then performed by each vector until the first thread of the vector writes the result to the output vector. The starting and ending row pointers of each vector is stored in shared memory. The number of blocks launched by the kernel can be variable in this algorithm but is set to the number of rows divided by the block dimensions.

Implementation

```
template <typename T, int
    THREADS_PER_VECTOR, int
    MAX_NUM_VECTORS_PER_BLOCK>
__global__ void spmv_light_kernel(int*
    cudaRowCounter, int* d_ptr, int*
    d_cols,T* d_val, T* d_vector, T*
    d_out,int N) {
  int i;
  T sum;
  int row;
  int rowStart, rowEnd;
  int laneId = threadIdx.x %
      THREADS_PER_VECTOR; //lane index in
      the vector
  int vectorId = threadIdx.x /
      THREADS_PER_VECTOR; //vector index in
      the thread block
  int warpLaneId = threadIdx.x & 31; //lane
      index in the warp
  int warpVectorId = warpLaneId /
      THREADS_PER_VECTOR; //vector index in
      the warp

  __shared__ volatile int
      space[MAX_NUM_VECTORS_PER_BLOCK][2];

  // Get the row index
  if (warpLaneId == 0) {
    row = atomicAdd(cudaRowCounter, 32 /
        THREADS_PER_VECTOR);
  }
  // Broadcast the value to other threads in
      the same warp and compute the row
      index of each vector
    row = __shfl(row, 0) + warpVectorId;

  while (row < N) {

    // Use two threads to fetch the row
        offset
    if (laneId < 2) {
      space[vectorId][laneId] = d_ptr[row +
          laneId];
    }
    rowStart = space[vectorId][0];
    rowEnd = space[vectorId][1];

    sum = 0;
    // Compute dot product
    if (THREADS_PER_VECTOR == 32) {

      // Ensure aligned memory access
      i = rowStart - (rowStart &
          (THREADS_PER_VECTOR - 1)) + laneId;

      // Process the unaligned part
      if (i >= rowStart && i < rowEnd) {
        sum += d_val[i] *
            d_vector[d_cols[i]];
      }

        // Process the aligned part
      for (i += THREADS_PER_VECTOR; i <
          rowEnd; i += THREADS_PER_VECTOR) {
        sum += d_val[i] *
            d_vector[d_cols[i]];
      }
    } else {
      for (i = rowStart + laneId; i <
          rowEnd; i +=
           THREADS_PER_VECTOR) {
        sum += d_val[i] *
            d_vector[d_cols[i]];
      }
    }
    // Intra-vector reduction
    for (i = THREADS_PER_VECTOR >> 1; i > 0;
        i >>= 1) {
      sum += __shfl_down(sum, i,
          THREADS_PER_VECTOR);
    }

    // Save the results
    if (laneId == 0) {
      d_out[row] = sum;
    }

    // Get a new row index
    if(warpLaneId == 0){
      row = atomicAdd(cudaRowCounter, 32 /
          THREADS_PER_VECTOR);
    }
    // Broadcast the row index to the other
        threads in the same warp and compute
        the row index of each vector
      row = __shfl(row, 0) + warpVectorId;

  }
}
```

The above five algorithms were chosen since they represent either the most widely used, in the case of CSR Scalar, CSR Vector and CSR Adaptive or the latest, in the case of PCSR and LightSpMV, in SpMV parallel processing.

V. EXPERIMENTAL SETUP

A. Implementation details

The analysis was run on cuda2 on a single GeForce GTX Titan X GPU. Its main characteristics are listed below:
cuda2
name: GeForce GTX TITAN X
Compute capability 5.2
total global memory(KB): 12505920
shared mem per block: 49152
regs per block: 65536
warp size: 32
max threads per block: 1024
max thread dim z:1024 y:1024 x:64
max grid size z:2147483647 y:65535 x:65535
total constant memory (bytes): 65536
multiprocessor count: 24
memory bus width: 384
memory clock rate (KHz): 3505000
L2 cache size (bytes): 3145728
max threads per SM: 2048

The matrices used in the analysis are a mix of the matrices used in the research papers presenting the above algorithms. They were chosen to represent a wide breadth of data type and size. They were downloaded from the SuiteSparse Matrix Collection [26] and are listed below:

TABLE I: Input Matrix characteristics

Matrix Name	Dimensions	NNZ	NNZ/row
Webbase	1M	3.1M	3.1
Circuit Simulation	1.4M	5.5M	3.8
Epidemiology	525K	2.1M	4.0
Freescale	3.4M	18.9M	5.5
Economics	206K	1.2M	6.2
Torso	115K	1M	8.9
DNA electrophoresis	1.5M	27.1M	18.0
Wind Tunnel	217K	5.9M	27.2
FEM/Cantilever	62K	2M	32.6
Ga41As41H72	268K	9.3M	35.0
FEM/Spheres	83K	3M	36.6
F1	343K	13.5M	39.5
Si41Ge41H72	185K	7.5M	40.9
Protein	36K	2.1M	60.2
Italian Railways	4K * 1M	11.2M	2634

NNZ:Number of Non Zeros

A more comprehensive file consisting of the type of data and the download links of the matrices is included in the code package.

The input vector is generated on the fly based on the number of columns in the input matrix with each element being set to 1.0.

To maintain consistency between algorithms, block dimensions are set to 1024 for all algorithms, barring the PCSR Kernel 2 which is set to 64. The reason for this is that since PCSR needs two shared memory arrays, the one holding the multiplied results ideally larger than one holding the row pointers, allocating too high a block dimension led to a severe drop in performance, likely due to the large quantum of shared memory needed per block.

The grid dimensions for all algorithms are set to match the algorithm requirements and where there is no fixed requirement, the dimensions are tuned to ones that performed the best on various data sizes.

All input and output data arrays are located in global memory for all algorithms. Analysis is conducted on both single and double precision versions of the input.

TABLE II: Implementation details

Parameter	CSR Scalar	CSR Vector	CSR Adaptive	PCSR	Light SpMV
Block Dim (BD)	1024	1024	1024	K1: 1024 K2: 64	1024
Grid Dim	Rows/ BD	Rows/ BD/ Warp Size	Row blocks	Rows/ BD	Rows/ BD
Shared mem	0	1024	1024	1024 + 64	Variable (dependent on number of vectors per block)

B. Challenges faced

It was a challenge to discover how to verify the accuracy of the results of the various algorithms due to the challenges of floating point representation. After a lot of reading, I decided to verify the results like below:
For numbers close to zero, verify that the absolute difference in results calculated on CPU and GPU is less than a certain small number.
For numbers away from zero verify that the relative difference between the CPU and GPU results is less than a certain small number [27].
Using this verification scheme, two of the algorithms, CSR Vector and LightSpMV, regularly present with errors for single precision calculations. After some investigation it was found that because these two algorithms do not add the multiplied results sequentially but in a scalar reduction style, they accumulate slightly different results than algorithms that add up the multiplied results sequentially, whether on the CPU or GPU. Double Precision calculations for all algorithms return with correct results.

VI. RESULTS AND DISCUSSION

A. Performance Metrics used

The following performance metrics are calculated from within the code:

- Computational throughput in GFLOP/s: Since there are two floating point operations per element of the input matrix, the calculation for GFLOP/s is as follows:

$$GFLOP/s = 2 * N/(t * 10^9)$$

where N is the number of elements in the input matrix and t is the time taken by the kernel in seconds.
- Effective memory bandwidth in GB/s: This calculation indicates how efficiently the kernel utilizes the memory bandwidth of the GPU. It is calculated as follows:

$$BW_{Effective} = (R_B + W_B)/(t * 10^9)$$

where R_B is the number of bytes loaded by the kernel from device memory, W_B is the number of bytes written by the kernel to device memory and t is the time taken by the kernel in seconds [28].

The following performance metrics are acquired from the CUDA profiler, nvprof [29]:

- Computational throughput related:
  1) Achieved occupancy: Ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor.
  2) Warp execution efficiency: Ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor expressed as percentage.
- Memory bandwidth utilization related:
  1) Global load efficiency: Ratio of requested global memory load throughput to required global memory load throughput expressed as percentage.

All performance reported is the average of three iterations of the algorithm.

B. Results and analysis

Computational Throughput

The first order of business is to understand which factor of the input matrix makes the biggest difference to the performance of SpMV algorithms. The graphs below plot the GFLOPS vs. Number of rows in increasing order, GFLOPS vs. NNZ in increasing order and GFLOPS vs. NNZ/Row in increasing order.

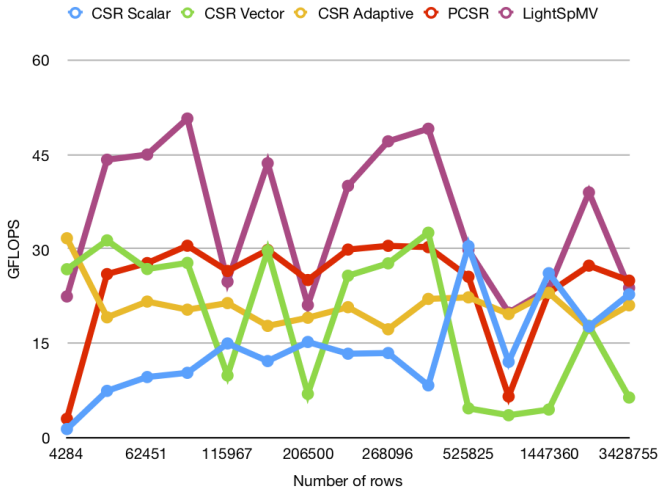

Fig. 1: GFLOPS vs. Number of Rows (Single Precision)

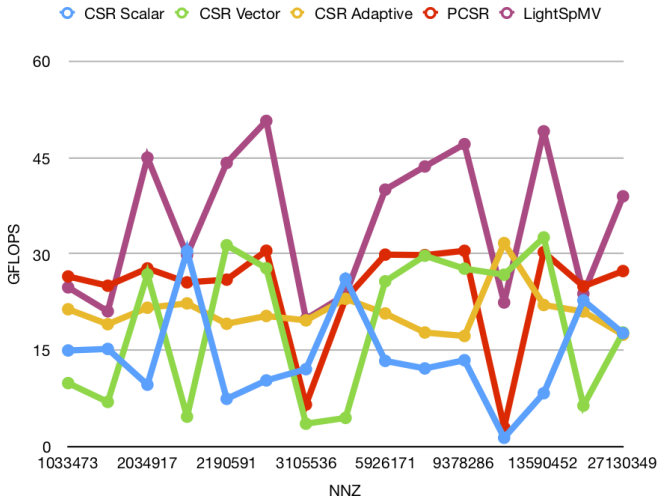

Fig. 2: GFLOPS vs. NNZ (Single Precision)

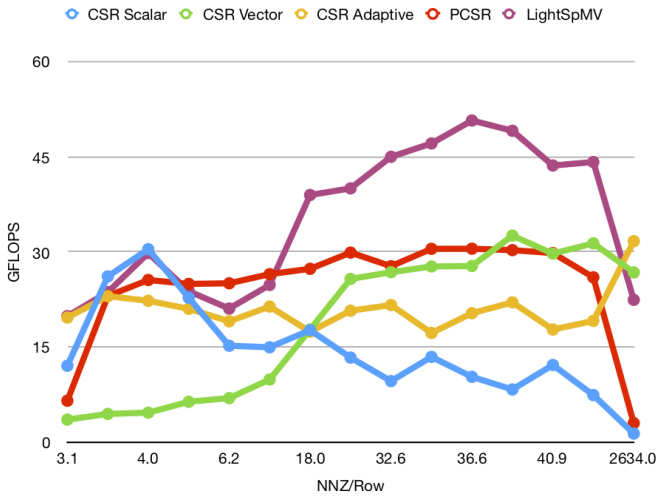

Fig. 3: GFLOPS vs. NNZ/Row (Single Precision)

While the GFLOPS vs. Number of rows graph and the GFLOPS vs. NNZ graph seem erratic and absent of a trend,

the GFLOPS vs. NNZ/Row graph appears to portray a trend for every algorithm. This observation is bolstered by the nature of the analysis presented in other research papers on SpMV [7], [13], [12], which are also based on number of non zero elements per row. The likely reason for this sensitivity to NNZ/Row is that the smallest unit of distribution of work in SpMV algorithms is the row, with every algorithm differing in how many threads work on how many rows. Hence the average number of elements in a row defines the amount of work each unit of execution will be required to do, and thus the extent of parallelism in the application. For this reason, we look in depth at the performance of all five algorithms at different NNZ/Row.

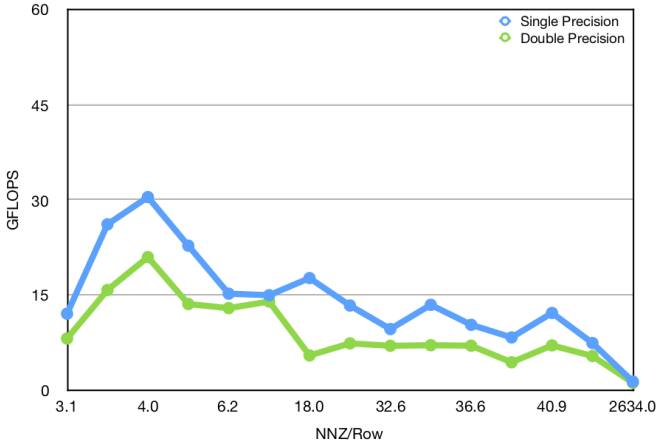

Fig. 4: CSR Scalar GFLOPS vs. NNZ/Row

The performance of CSR Scalar deteriorates as the NNZ/Row increases. The reason for this is that in CSR Scalar, each thread handles one full row so if the average row length is very long, time taken by the algorithm is longer than if the average row length is very short. The variability observed in the trend line (the jagged areas) is likely because some matrices that have greater variability in NNZ/Row than others which means that some threads will take longer to finish than others, increasing the time taken as compared to a matrix that has the same average NNZ/Row as the first one but lesser variability between individual rows and therefore more equally divided work between threads. Expectedly, the performance of the double precision algorithm is lower than the single precision algorithm but follows the same trend. CSR Scalar averages 14.3 GFLOP/s on single precision and 9.15 GFLOP/s on double precision numbers.

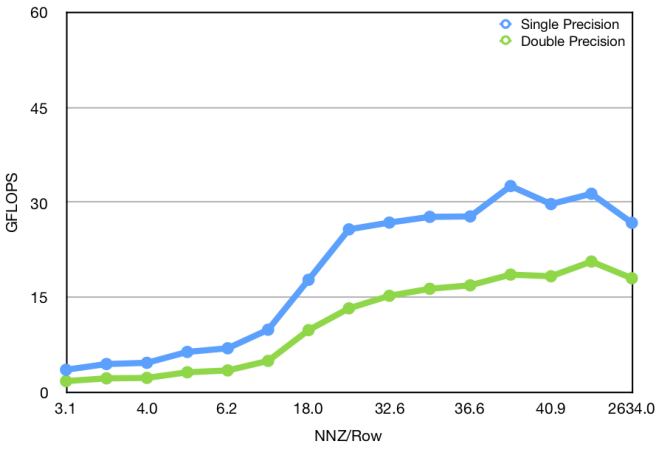

Fig. 5: CSR Vector GFLOPS vs. NNZ/Row

The performance of CSR Vector displays a trend opposite to CSR Scalar in that it does better on matrices with longer rows than shorter rows. The reason for this is that a warp sized chunk of threads is responsible for each row and if that chunk is given very few elements to work on, a lot of processing power is being wasted. So if two matrices have the same number of elements but one has a smaller NNZ/Row than the other, more threads will be involved in processing the matrix with the smaller NNZ/Row which will increase the time taken by the kernel and reduce the GLOP/s. CSR Vector averages 18.8 GFLOP/s on single precision and 11 GFLOP/s on double precision numbers.

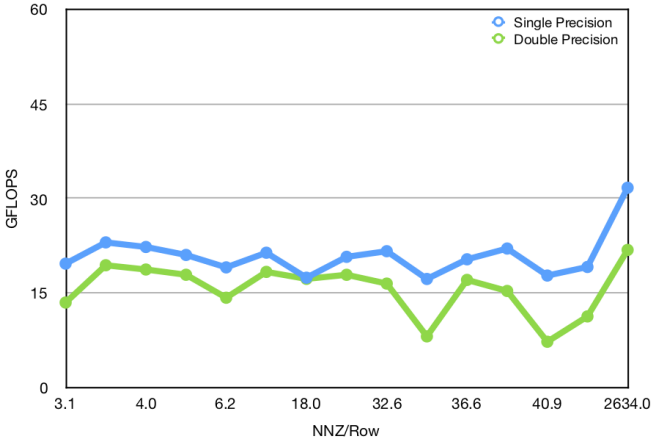

Fig. 6: CSR Adaptive GFLOPS vs. NNZ/Row

The performance of CSR Adaptive is relatively stable across matrices with different NNZ/Row, averaging 20.9 GFLOP/s on single precision numbers and 15.6 GFLOP/s on double precision numbers across all matrices. This is because differently sized rows are handled by different algorithms with large rows being handled by CSR Vector and smaller rows being handled by CSR Stream which dedicates one thread to process each row. This is adaptive part of the algorithm. It utilizes the strengths of CSR Vector while tackling its primary weakness - that of dealing with small sized rows

and this is clear from the graph as well. A point to note is that while we are not considering the time taken by the CPU procedure in our analysis, it does have a cost. However since the complexity of calculating the blocks is related to the number of rows, not the number of non-zeroes in the matrix; this generally takes less than 1% of the time that it takes to generate the CSR data structure [12].
An alteration to the CSR Vector that I implemented resulted in CSR Adaptive being the best algorithm among the five to deal with matrices with very large rows such as the Italian Railways matrix which has an average of 2634 non zero elements per row. The table below illustrates this.
The reason for this difference in performance in processing

TABLE III: Performance of CSR Adaptive on Italian Railways Matrix

Algorithm	GFLOPS Single Precision	GFLOPS Double Precision
CSR Adaptive with unmodified CSR Vector (one warp dealing with the row)	2.076331	1.337608
CSR Adaptive with modified CSR Vector (whole block dealing with the row)	31.698621	21.828796

the Italian Railways matrix is that the row length is, on average, very large and having only one warp to deal with large rows results in the those warps taking a very long time to finish as compared to warps handling shorter rows, thus increasing the time taken by the kernel as a whole. By distributing the work amongst all threads in the block we have increased the parallelism in the algorithm and therefore increased speedup. Also having only 32 threads process the entire block when the block size is much larger is a severe waste of computing resources and reduces parallelism.

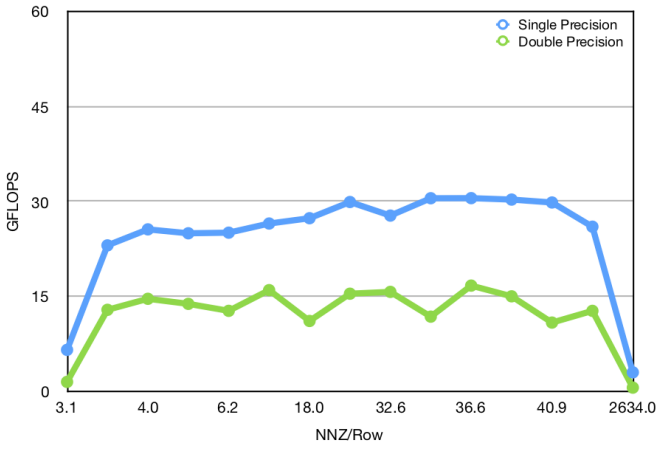

Fig. 7: PCSR GFLOPS vs. NNZ/Row

The performance of PCSR is relatively stable across all NNZ/Row, averaging at 24.4 GFLOP/s on single precision numbers and 12.08 on double precision numbers, except for matrices with very small rows and very large rows. This could be because in the case of matrices with very small rows, a lot of the shared memory allocated for the storage of multiplied products and a lot of processing power is being wasted since only a fixed number of rows is being processed per block, irrespective of their size. With matrices with a variable number of very long rows, there is load imbalance between the blocks with the blocks getting a lot of long rows taking longer to finish thereby reducing the performance of the kernel. Another thing to note is the large gap between the performance of single and double precision versions of the algorithm. Indeed this gap is the largest for PCSR among the five algorithms, with the GFLOP/s reducing by half or more than half when moving from single to double precision numbers. Double precision numbers are double the size of single precision numbers and this doubling is perhaps felt more by PCSR than other algorithms because of the large amount of overall shared memory being allocated per block for this algorithm. Requesting large amounts of shared memory can reduce parallelism due to fewer blocks running in an SM for lack of resources.

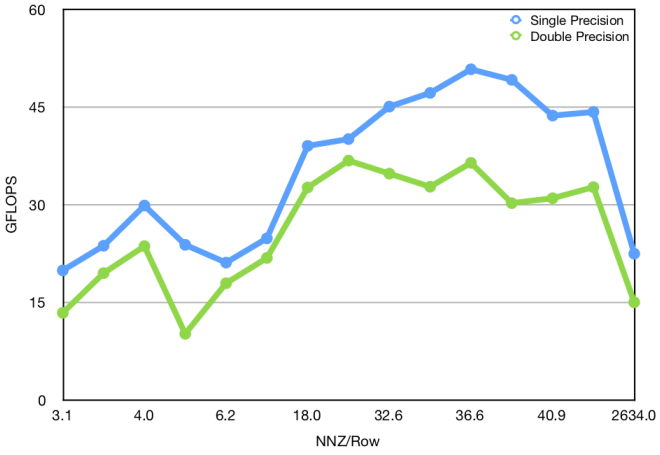

Fig. 8: LightSpMV GFLOPS vs. NNZ/Row

While the performance of LightSpMV is overall quite high, it excels at matrices with 18+ NNZ/Row with GFLOP/s topping at 50 for NNZ/Row of 36.6. This indicates that the kernel, when launched with blocks that have a fewer number number of vector lanes performs better than when launched with a large number of vector lanes. The main cause of this is that there will be better memory coalescing with larger sized vector lanes as we will see in the Memory Bandwidth Utilization section. LightSpMV averages 34.9 GFLOP/s on single precision and 25.8 GFLOP/s on double precision numbers.

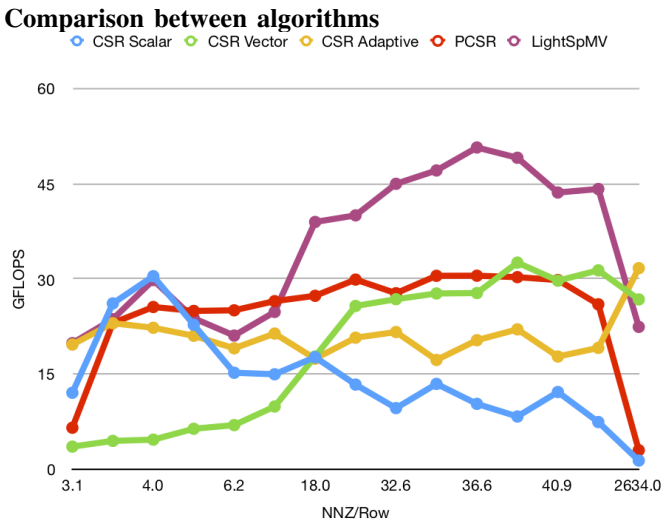

Fig. 9: GFLOP/s of different algorithms vs. NNZ/Row (Single Precision)

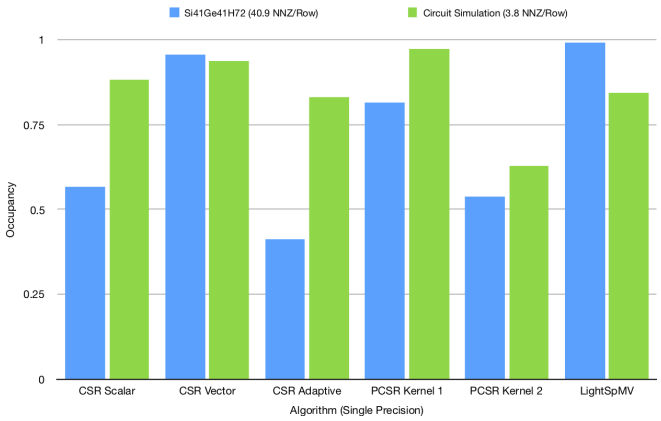

Fig. 10: Achieved Occupancy of different algorithms on two illustrative matrices (Single Precision)

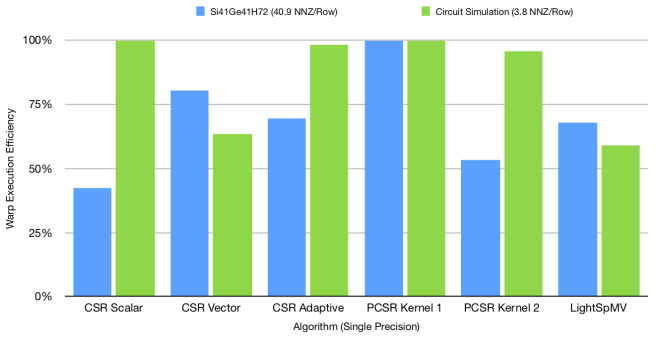

Fig. 11: Warp execution efficiency of different algorithms on two illustrative matrices (Single Precision)

In terms of GFLOP/s as featured in Fig. 9, LightSpMV seems to be the clear winner among all algorithms for most problem sizes (9/15 matrices) with the exception of matrices with small rows where CSR Scalar (2/15) and PCSR (3/15) seem to excel. Its utilization of shuffle functions to enable threads within warps to rapidly communicate and the division of warps into a variable number of vector lanes based on problem characteristics are a clear advancement in the optimization of SpMV.

Fig. 10 and 11 reveals an interesting point about CSR Adaptive, that it is more efficient for shorter row matrices than longer row ones. The reason for this is that when the rows are short, there will be more active threads per warp, since each row is handled by a single thread, and more active warps per cycle, thus more parallelism.

While CSR Vector has a high achieved occupancy for both types of matrices (> 0.9), it has a lower warp execution efficiency pointing to the problem of inactive threads within a warp processing a short row or a row not aligned to warp size.

Both Fig. 10 and 11 show that CSR Scalar exhibits far less parallelism on rows of larger sizes than rows of smaller sizes while LightSpMV seems to have a load imbalance problem between threads in a warp likely due to the fact that certain vectors will be handling more elements than others in the case of matrices with a high degree of NNZ/Row variability.

Memory Bandwidth Utilization

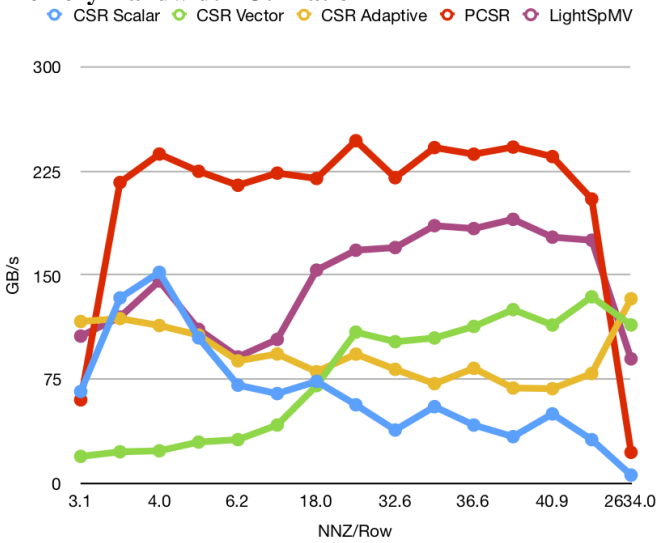

Fig. 12: Effective Memory Bandwidth in GB/s of different algorithms vs. NNZ/Row (Single Precision)

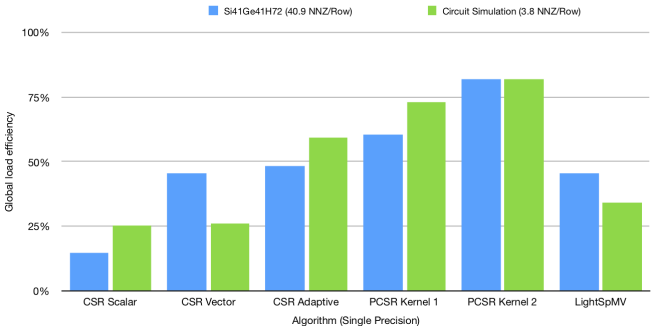

Fig. 13: Global Load Efficiency (Single Precision)

Except for matrices with very small rows, CSR Scalar has

very poor memory bandwidth utilization, going down to 50 and below when NNZ/Row $>$ 27. The main reason for this is that since every thread handles one row and loads all of its values from memory, memory accesses are extremely non-coalesced. The effect of this is minimized when rows are short but as rows get longer, CSR Scalar's bandwidth utilization craters.

CSR Vector was designed to overcome the weakness of CSR Scalar in terms to memory coalescing but it fails to achieve this for small rows and for rows that are not perfectly aligned to warp size. This is apparent in Fig. 10 and confirmed by Fig. 11 where when the row size is hovering around warp size, CSR Vector's bandwidth utilization is 100+ GB/s for the single precision algorithm but is less than 30 when NNZ/Row $<$ 6.

CSR Adaptive has the most even bandwidth utilization among all the algorithms although it is not very high, never exceeding 133 GB/s for the single precision algorithm and 153 GB/s for the double precision algorithm. The likely reason for the low bandwidth is that every thread of CSR Adaptive is responsible for loading only one element each of the input matrix and input vector while in PCSR and LightSpMV, threads load more than two elements typically. According to both Fig. 10 and 11, PCSR has the highest memory bandwidth utilization among all five algorithms due to the fact that it is very particular about coalescing memory accesses in both it's kernels and that it accesses the global memory more due to having to write an array of the multiplied results to it.

From Fig. 11 it is clear that the memory bandwidth utilization of LightSpMV is better for matrices with longer rows since memory will be accessed in a more coalesced manner by larger sized vector lanes than with the small lanes when dealing with short rows.

VII. CONCLUSIONS

It is clear from the analysis that SpMV is far from a solved problem. While later approaches have built on top of earlier ones and improved on them, the arbitrary shapes and sizes of sparse matrices and the necessity to access them in irregular patterns cause a lot of performance pitfalls in the parallelism of SpMV. Algorithms that are careful about memory coalescing and that aim to divide work equally amongst threads are the more efficient ones. The most recent approach utilizing shuffle functions, atomic instructions and customization of kernels based on problem characteristics looks promising.

VIII. APPENDIX

A separate PDF of the GFLOP/s and GB/s for all algorithms is included in the project package. It is labelled 'Appendix.pdf'.

REFERENCES

[1] L. N. Trefethen and D. Bau, III, Numerical Linear Algebra. Society for Industrial and Applied Mathematics, 1997.

[2] E.-J. Im and K. Yelick, Optimization of Sparse Matrix Kernels for Data Mining, in Proc. of the Workshop on Text Mining, 2001.

[3] Aluru, M., Zola, J., Nettleton, D., Aluru, S. (2012). Reverse engineering and analysis of large genome-scale gene networks. Nucleic acids research (p. gks904).

[4] J. R. Gilbert, S. Reinhardt, and V. B. Shah, High performance Graph Algorithms from Parallel Sparse Matrices, in Proc. of the Intl Workshop on Applied Parallel Computing, 2006.

[5] Bell, Nathan Garland, Michael. (2009). Efficient Sparse Matrix-Vector Multiplication on CUDA.

[6] Goumas, G., Kourtis, K., Anastopoulos, N., Karakasis, V., Koziris, N. (2009). Performance evaluation of the sparse matrix-vector multiplication on modern architectures. The Journal of Supercomputing, 50(1), 3677.

[7] Liu, Y., Schmidt, B. (2017). LightSpMV: Faster CUDA-Compatible Sparse Matrix-Vector Multiplication Using Compressed Sparse Rows. Journal of Signal Processing Systems, 90(1), 69-86. doi:10.1007/s11265-016-1216-4

[8] N. Bell and M. Garland, Implementing Sparse Matrix vector Multiplication on Throughput-oriented Processors, in Proc. of the Conference on High Performance Computing, Networking, Storage and Analysis (SC), 2009.

[9] F. Lu, J. Song, F. Yin, and X. Zhu, Performance evaluation of hybrid programming patterns for large CPU/GPU heterogeneous clusters, Computer Physics Communications, vol. 183, no. 6, pp. 11721181, 2012.

[10] M. M. Dehnavi, D. M. Fernandez, and D. Giannacopoulos, Finite-element sparse matrix vector multiplication on graphic processing units, IEEE Transactions on Magnetics, vol. 46, no. 8, pp. 29822985, 2010.

[11] M. M. Dehnavi, D. M. Fernandez, and D. Giannacopoulos, Enhancing the performance of conjugate gradient solvers on graphic processing units, IEEE Transactions on Magnetics, vol. 47, no. 5, pp. 11621165, 2011.

[12] Greathouse, J. L., Daga, M. (2014). Efficient Sparse Matrix-Vector Multiplication on GPUs Using the CSR Storage Format. SC14: International Conference for High Performance Computing, Networking, Storage and Analysis. doi:10.1109/sc.2014.68

[13] He, G., Gao, J. (2016). A Novel CSR-Based Sparse Matrix-Vector Multiplication on GPUs. Mathematical Problems in Engineering, 2016, 1-12. doi:10.1155/2016/8471283

[14] Vazquez, F., Fernandez, J. J., Garzon, E. M. (n.d.). A new approach for sparse matrix vector product on NVIDIA GPUs. CONCURRENCY AND COMPUTATION-PRACTICE EXPERIENCE, 23(8), 815826. https://doi-org.proxy.library.nyu.edu/10.1002/cpe.1658

[15] Ashari, A., Sedaghati, N., Eisenlohr, J., Sadayappan, P. (2015). A model-driven blocking strategy for load balanced sparse matrixvector multiplication on GPUs. Journal of Parallel and Distributed Computing, 76, 315. https://doi-org.proxy.library.nyu.edu/10.1016/j.jpdc.2014.11.001

[16] Choi, J. W., Singh, A., Vuduc, R. W. (2010). Model-driven autotuning of sparse matrix-vector multiply on gpus. In ACM sigplan notices, (Vol. 45 pp. 115126): ACM.

[17] Monakov, A., Lokhmotov, A., Avetisyan, A. (2010). Automatically tuning sparse matrix-vector multiplication for gpu architectures. In High Performance Embedded Architectures and Compilers (pp. 111125): Springer.

[18] Su, B. Y., Keutzer, K. (2012). clspmv: A cross-platform opencl spmv framework on gpus. In Proceedings of the 26th ACM international conference on Supercomputing (pp. 353364): ACM.

[19] Li, R., Saad, Y. (2013). Gpu-accelerated preconditioned iterative linear solvers. The Journal of Supercomputing, 63(2), 443466.

[20] Yan, S., Li, C., Zhang, Y., Zhou, H. (2014). yaspmv: Yet another spmv framework on gpus (Vol. 49, pp. 107118): ACM.

[21] Liu, W., Vinter, B. (2015). Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In Proceedings of the 29th ACM on International Conference on Supercomputing (pp. 339350).

[22] Tang, W., Tan, W., Goh, R. S. M., Turner, S., Wong, W. K. (2015). A family of bit-representation-optimized formats for fast sparse matrix-vector multiplication on the gpu. IEEE Transactions on Parallel and Distributed Systems, 26(9), 23732385.

[23] Nagasaka, Y., Nukada, A., Matsuoka, S. (2016). Adaptive multi-level blocking optimization for sparse matrix vector multiplication on gpu. Procedia Computer Science, 80, 131142.

[24] CUDA C Programming Guide. NVIDIA Developer Documentation. (n.d.). Retrieved December 18, 2018,

from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.htmlwarp-shuffle-functions.
[25] CUDA Pro Tip: Do The Kepler Shuffle. (2016, January 11). Retrieved December 18, 2018, from https://devblogs.nvidia.com/cuda-pro-tip-kepler-shuffle/
[26] SuiteSparse Matrix Collection Formerly the University of Florida Sparse Matrix Collection. (n.d.). Retrieved December 18, 2018, from https://sparse.tamu.edu/
[27] Comparing Floating Point Numbers, 2012 Edition. (2018, November 17). Retrieved December 18, 2018, from https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/
[28] How to Implement Performance Metrics in CUDA C/C. (2018, April 24). Retrieved December 18, 2018, from https://devblogs.nvidia.com/how-implement-performance-metrics-cuda-cc/
[29] Profiler User's Guide. (n.d.). Retrieved December 18, 2018, from https://docs.nvidia.com/cuda/profiler-users-guide/index.html