Data-driven Mixed Precision Sparse Matrix Vector Multiplication for GPUs

KHALID AHMAD, HARI SUNDAR, and MARY HALL, University of Utah

51

We optimize Sparse Matrix Vector multiplication (SpMV) using a mixed precision strategy (MpSpMV) for Nvidia V100 GPUs. The approach has three benefits: (1) It reduces computation time, (2) it reduces the size of the input matrix and therefore reduces data movement, and (3) it provides an opportunity for increased parallelism. MpSpMV's decision to lower to single precision is *data driven*, based on individual nonzero values of the sparse matrix. On all real-valued matrices from the Sparse Matrix Collection, we obtain a maximum speedup of 2.61 $\times$ and average speedup of 1.06 $\times$ over double precision, while maintaining higher accuracy compared to single precision.

CCS Concepts: • **Mathematics of computing** $\rightarrow$ **Mathematical software performance**; • **Computing methodologies** $\rightarrow$ *Shared memory algorithms*; • **Computer systems organization** $\rightarrow$ *Single instruction, multiple data*;

Additional Key Words and Phrases: Mixed precision, sparse matrices, SpMV, correctness

ACM Reference format:

Khalid Ahmad, Hari Sundar, and Mary Hall. 2019. Data-driven Mixed Precision Sparse Matrix Vector Multiplication for GPUs. *ACM Trans. Archit. Code Optim.* 16, 4, Article 51 (December 2019), 24 pages. <https://doi.org/10.1145/3371275>

1 INTRODUCTION

Sparse matrix-vector multiplication (SpMV) is a fundamental performance bottleneck in iterative methods for solving large-scale linear systems, eigenvalue problems, and least squares problems. In SpMV, the operation $y = A \cdot x + y$ is performed, where A is a sparse matrix and x and y are dense vectors. To reduce computation and storage requirements, SpMV implementations attempt to store only nonzero elements of sparse matrix A and use auxiliary arrays to map nonzero elements to their corresponding row and column [35]. These structures introduce indirect and irregular memory accesses, which, when combined with the lack of data reuse, lead to low computational intensity. SpMV performance is significantly impacted by architecture, matrix representation, and nonzero structure of the input matrix. The extensive literature dealing with the development, optimization,

This is a new article, not an extension of a conference article.

This research was funded by NSF award 1704715. This research was funded in part by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

Authors' addresses: K. Ahmad (corresponding author), H. Sundar, and M. Hall, School of Computing, University of Utah, 50. S Central Campus Drive, Salt Lake City, UT 84112; emails: {khalid, hari, mhall}@cs.utah.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

1544-3566/2019/12-ART51

<https://doi.org/10.1145/3371275>

and autotuning of parallel SpMV kernels for both multi-core and many-core architectures is well summarized by Langr et al. [25].

As an example architecture well suited for memory-intensive kernels like SpMV, Nvidia V100 graphics processors (GPUs) exhibit peak performance of more than 7 Tflops in double precision and peak memory bandwidth in excess of 900 GB/s. This article seeks to optimize SpMV using a novel mixed precision strategy (MpSpMV) on Nvidia V100 GPUs. As of this writing NVIDIA's cuSPARSE and CUSP libraries [11] do not support mixed precision, and it is an active area of research. Our goal does not include a general extended precision capability like Python, Mathematica, Maple, and others, where all variables, arithmetic operations, and intrinsic functions can have arbitrary precision. Instead, we are interested in optimizing conventional floating point algorithms written in C, which natively supports only single and double precision data types. In other words, we are interested in applications where performance remains a primary concern, and high levels of accuracy are required.

While most implementations of SpMV used in scientific computing applications typically choose double precision floating point, it is well established that higher performance can be achieved by relaxing the precision of a computation. Hasan et al. store the matrix in single precision to reduce its memory footprint [2]. However, they observe that achieving numerical stability for Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG) algorithm requires using double precision arithmetic for Many Fermion Dynamics for nuclear structure (MFDn) calculations. So they promote the data type of the produced results from the Sparse Matrix Multi-vector Multiplication (SpMM) phase to double precision and then start the LOBPCG computation. Sven et al. discuss the growing cost of scientific computations and the possibility of using mixed precision algorithms for exascale projects [26]. Dominik et al. surveyed [16] precision issues in finite elements method (FEM) simulations and how GPUs were used to accelerate double precision computations for high accuracy before double precision was supported. Mixed precision is also used in deep learning applications [18] where a large number of accumulations are required and mixed precision is used to reduce the accumulated rounding errors. Additional application examples include molecular dynamics [17], climate models [15], and several deep learning applications [14, 31, 32].

Taking a piece of code and simply downcasting its precision by rewriting all doubles as floats may lead to intolerable inaccuracies in the floating point computation, which may negatively impact the convergence of the solvers that employ mixed precision SpMV. Therefore, any mixed precision approach must identify data and computations that can be lowered to single, half or other floating point representations without harming the underlying computation. Prior work exploits this opportunity numerically by extending the use of iterative refinement to solve sparse linear systems [10]. Their technique assigns single precision to the computationally intensive parts of the code to boost its performance while maintaining the use of double precision at critical stages that directly affect the final results.

MpSpMV takes an alternative *data-driven approach*, deciding to lower from double precision to single precision at the granularity of individual nonzero values of the sparse matrix. At the beginning of the computation, the input matrix is split into single and double precision nonzero elements, where values in the range $(-1, 1)$ (or other ranges that are well represented by lower precision floating point) are multiplied by the input vector x in single precision and other values remain in double precision, combining the result into a double precision output. With this simple change, the approach has three benefits: (1) it reduces computation time, since single precision computation is faster; (2) it reduces the size of the input matrix, and therefore reduces data movement; and (3) it provides an opportunity for increased instruction-level parallelism such as on the integer units used for address calculation [33]. Our findings suggest that MpSpMV is capable of outperforming double precision SpMV and has better accuracy than single precision SpMV.

Table 1. Single and Double Precision Bit Distribution Defined by the IEEE 754 Floating Point Arithmetic Standard

Data type	Sign bit-width (s)	Exponent bit-width (e)	Mantissa/Fraction bit-width (m)	Decimal digits of accuracy
Float	1	8	23	7
Double	1	11	52	16

This article describes our data-driven mixed precision approach and demonstrates its advantages for accelerating SpMV on large matrices. The novel contributions of the article are as follows:

- It describes our MpSpMV approach, which splits the input matrix at runtime, and performs the calculation across the split matrix.
- It presents an optimized mixed precision SpMV kernel, implemented for GPUs.
- It presents an evaluation that shows the advantages of using mixed precision in terms of performance and accuracy of the final result. It demonstrates speedups over double precision on average 1.06 $\times$ and a maximum of 2.61 $\times$ while maintaining higher significant decimal digits of accuracy compared to single precision.

The remainder of the article is organized as follows. The next section provides background on relevant sparse storage formats and floating point accuracy. Section 3 goes through the data-driven approach for mixed precision, Section 4 details the mixed precision SpMV kernel implementation, which builds on CUSP library implementation of CSR-vector SpMV [8]. We then present the results of our performance evaluation experiments and accuracy measurements and discuss the experimental setup and provide a performance comparison of the three precision types. Section 6 discusses related work. Finally, we conclude this work with a summary of contributions and ideas for possible future work.

2 BACKGROUND AND MOTIVATION

This section provides the necessary background on floating point accuracy and the sparse matrix representation that will be used throughout the article.

2.1 Accuracy Differences between Single and Double Precision

Scientific simulations typically use double precision floating point by default, under the assumption that higher precision leads to higher accuracy [6]. To understand the difference between single and double precision accuracy it is helpful to first look at the bit distributions for 32-bit and 64-bit floating points, as shown in Table 1. Floating point formats are used to represent real continuous values. They consist of a sign, an exponent, and a mantissa. Exponent bits give the floating point format a wide range while mantissa bits influence its precision.

The IEEE 754 standard for floating point arithmetic guarantees that the relative error of a floating point value is bounded by Equation (1), where η is referred to as rounding error unit or machine precision or machine epsilon, β is the base of the numbering system, and m is precision or number of digits.

$$\eta = \frac{1}{2} \cdot \beta^{-m} \quad (1)$$

Applying Equation (1) for single precision, there are 23 significant binary digits, which gives us approximately 7 decimal digits:

$$\eta = \frac{1}{2} \cdot \beta^{-m} = \frac{1}{2} \cdot 2^{-23} \approx 6.0 \times 10^{-8} \quad (2)$$

11	12	13	14	0	0
0	22	23	0	0	0
0	0	33	34	35	36
0	0	0	44	45	0
0	0	0	0	0	56
0	0	0	0	0	66

Fig. 1. A 6*6 example sparse matrix.

```

for (i=0; i<N; i++) {
  for (j=index[i]; j<index[i+1]; j++) {
    y[i] += A[j] * x[col[j]];
  }
}

```

Fig. 2. Sequential SpMV computation.

A	11	12	13	14	22	23	33	34	35	36	44	45	56	66
index (row pointer)	0	4	6	10	12	13	14							
col	0	1	2	3	1	2	2	3	4	5	3	4	5	5

Fig. 3. The CSR representation.

Applying Equation (1) for double precision, there are 52 significant binary digits, which gives us approximately 16 decimal digits:

$$\eta = \frac{1}{2} \cdot \beta^{-m} = \frac{1}{2} \cdot 2^{-52} \approx 1.1 \times 10^{-16} \quad (3)$$

2.2 Sparse Storage Format: Compressed Sparse Row

One of the most common representations of sparse matrices, used throughout this article, is Compressed Sparse Row (CSR), depicted in Figures 1 through 3. CSR stores the nonzero elements of the matrix in a data array A and corresponding column indices for each of these nonzero elements in an integer array col. The third array index stores the indices in A that correspond to the first nonzero element of each row. We see in Figure 2 how indirection through index and col make it possible to identify the proper columns and rows corresponding to dense input vector x and dense output vector y, respectively.

The index array is of size N+1, where N is the number of matrix rows. The last element in index contains the total number of nonzero elements in the matrix. For matrices with more than one nonzero per row, CSR requires less storage for row indices than column indices, since it stores pointers to the beginning of the rows instead of repeated row indices of nonzero elements in the same row. In addition, the index array allows for easy computations of some quantities of interest for a matrix such as the number of nonzero elements in a row $nnz_i = index[i+1] - index[i]$ and the total number of nonzero elements $index[N]$.

The overall data storage required for the SpMV computation is generally dominated by matrix A and is therefore primarily impacted by the floating point representation and computations related to matrix A.

3 A DATA-DRIVEN APPROACH TO MIXED PRECISION

The IEEE 754 standard guarantees that the relative error is bounded by Equation (1), but the absolute error is dependent on the value. During SpMV, the basic operation involves an inner product, i.e., multiplying and accumulating values. The net result is influenced by both the relative error as well as the absolute error depending on the values of the matrix and the vector. To understand this clearly, consider the number $x = (s, e, m)$, where s is the sign, e is the exponent, and m is the mantissa. Recall that the exponent e specifies the range of values that the number is between, i.e.,

Fig. 4. Spy plot of the nonzero elements of cop20k_A sparse matrix (a) and the values that are inside (b) and outside (c) the range -1 to 1 .

$x \in [2^e, 2^{e+1})$, and the mantissa tells us where x is in that range. In other words, we have the same number of bits (μ providing 2^μ discrete levels) between 1 and 2 ($e = 1$) as between $1,024$ and $2,048$ ($e = 10$). Mathematically, we can use this to define the precision in any interval. The precision p_e in the interval $[2^e, 2^{e+1})$ is

$$p_e = \frac{2^{e+1} - 2^e}{2^\mu} = \frac{2^e}{2^\mu}$$

We want p_e to be as small as possible, and, given that μ is fixed and typically $\mu > e$, smaller e equates to better precision. We use this property to develop our data-driven SpMV algorithm.

The above property implies that the absolute error is smaller for smaller values being represented. In other words, if we consider a set of irrational numbers, then the representation error is smaller for smaller numbers in this set compared to the larger irrational numbers. For problems such as computing the inner product, where such errors would get accumulated over the reduction, the error corresponding to the largest number(s) will be significantly larger. This allows us to relax the precision requirements for computing the multiplication of the smaller numbers. Assuming we can identify an appropriate interval, we effectively split the sparse matrix into two (more) sparse matrices, one with values within this interval and one outside (see Figure 4). Without loss of generality, let us assume this interval is $(-1, 1)$. All values of the matrix within this interval will have a smaller representation error compared to values outside, and therefore the values outside the range need to be computed in a higher precision so that the overall representation error is reduced. Therefore, we use single precision for values inside the range and double precision for those outside.

Performing the SpMV computation in such a data-driven mixed-precision mode will ensure that the overall precision is higher than just using single-precision and will have additional performance benefits. The first performance benefit is due to the reduction in the overall data movement. Since the matrix values within the interval are stored in single precision rather than double, the memory for these values is cut in half. While we will need to move two copies of x , the savings from reduction of the matrix memory footprint are likely more. A second important performance boost is due to increased parallelism. On modern GPUs such as the Nvidia V100, we have separate execution units for single and double precision computations and integer addressing calculations. By splitting the matrix, thus exposing two independent streams of address calculation and even computation, we can potentially utilize these simultaneously [33]. Second, the single precision units have increased SIMT parallelism as well. These reasons together provide an effective way to achieve significant performance boost to the mixed-precision implementation in exchange for

Typical Sparse Matrix Library

Fig. 5. Integrating matrix format conversion into application, as is done in typical sparse matrix libraries.

a minor reduction to the overall precision. In many applications, the full precision provided by double might not be necessary, and our methods allow such a choice without requiring hardware support for arbitrary precision datatypes. More importantly, the precision can be adjusted by adjusting the interval, something that can be completely controlled at the application level.

Splitting of the matrix based on values in some cases creates matrices with additional structure that can be exploited to improve data reuse and, therefore, further reduce data movement. The merits of such an approach are visualized in Figure 4. We see a new structure emerge in the portions of the computation that are outside the range $(-1,1)$ and would be executed in double precision. This new double precision matrix is likely to have significant data reuse on input vector x , along with reuse with the single precision portion of the matrix from Figure 4. Note that this exposed structure may make portions of the split matrix more amenable to further optimization. For example, exploiting the structure with a different sparse matrix representation such as a diagonal matrix (DIA) [7] or block compressed sparse row (BCSR) [11] could lead to better locality, possibly combined with tiling.

4 MPSPMV METHODOLOGY

This section presents details of the mixed precision approach. We describe the conversion of the matrix into its double and single precision components, provide the optimized MpSpMV GPU kernel, and provide analysis of the increased parallelism and reduced data set size for the mixed precision computation.

4.1 Format Conversion: Splitting the Input Matrix

We briefly discuss the placement of the matrix format conversion in the application's execution. Sparse matrix libraries, such as the open source CUSP library for Nvidia GPUs, often employ a data transformation where an input matrix is converted from a standard format, such as the CSR representation from Section 2.2 or Coordinate format (COO) that expands the representation of the row index and produces a new format that is specially suited for the input matrix, application domain, or architecture.

As depicted in Figure 5, this conversion takes place once at runtime, prior to running the iterative solver. Since iterative solvers such as conjugate gradient use the same input matrix throughout, the

ALGORITHM 1: Sparse matrix splitting for mixed precision computation.

Input: Sparse matrix A and Range

Output: Sparse matrix AxS and AxD

```

1 Read matrix in COO storage format if  $A$  nonzero values are inside the Range then
2   | Assign value to  $AxS$ ;
3 else
4   | Assign value to  $AxD$ ;
5 end
6 Convert matrix from COO storage format to CSR storage format.
```

Table 2. Comparison of Matrix Storage Requirements for Single, Double, and Mixed Precision

Single		Double		Mixed	
Matrix	Size	Matrix	Size	Matrix	Size
A	NNZ*4 bytes	A	NNZ*8 bytes	AxS	NNZ_s*4 bytes
col	NNZ*4 bytes	col	NNZ*4 bytes	col_S	NNZ_s*4 bytes
index	(N+1)*4 bytes	index	(N+1)*4 bytes	index_S	(N+1)*4 bytes
x	N*4 bytes	x	N*8 bytes	xS	N*4 bytes
y	N*4 bytes	y	N*8 bytes	y	N*8 bytes
				AxD	NNZ_d*8 bytes
				col_D	NNZ_d*4 bytes
				index_D	(N+1)*4 bytes
				xD	N*8 bytes
TOTAL	8NNZ + 12N + 4 bytes		12NNZ + 20N + 4 bytes		8NNZ_s + 12NNZ_d + 28N + 8 bytes

one-time cost of this conversion is amortized over numerous iterations, and the performance gain of the specialized format is assumed to significantly reduce execution time of each iteration. As with CUSP, this matrix format conversion occurs at runtime on the CPU, and then the converted matrix is copied to the GPU for processing.

As shown in Algorithm 1, the output of our matrix conversion is a split matrix A and its auxiliary indexing structures, col and index. As x is a vector, we also retain both a single and double precision version. A comparison of matrices and their sizes for single, double or mixed precision is shown in Table 2. We expect the matrix size to be dominated by the NNZ terms. It is clear from the table that if a substantial percentage of the nonzeros can use single precision, then the volume of data moved through the memory system is reduced by $4 \cdot \text{NNZ}_s - 8N - 4$ bytes. The point where mixed precision is unprofitable in terms of space reduction, which may occur for either small matrices or those that are very sparse, is $\text{NNZ}_s < 2N + 1$.

4.2 Optimized MpSpMV GPU Kernel

The optimized SpMV GPU single precision and optimized mixed precision kernels are shown in Listings 1 and 2, respectively. The single precision baseline code is adapted from Nvidia's open source CUSP library [8]. The double precision baseline, not shown, is identical but uses double data types everywhere. We modified the baseline code to arrive at the code in Listing 2.

All kernels use a *CSR vector* strategy for parallelizing the SpMV computation, as defined by Bell and Garland and shown to be the most consistently performing implementation across different kinds of matrix structures [7]. Architecture aside, the most natural way to parallelize SpMV is across rows of the matrix, where each row calculation is independent and can be computed by a separate thread. In contrast, the key motivation behind *CSR vector* is the performance gain achieved by *global memory coalescing* on the V100. If adjacent threads are operating on adjacent elements of A and col, then the memory system can *coalesce* these accesses across threads and, thus, dramatically reduce the memory transfers. If we parallelize the computation within a row, then we must use a reduction to combine the results for each thread, shown in lines 35–41 of Listing 1. *CSR vector* often achieves performance that is integer factors faster than parallelizing across rows, depending on matrix structure.

The code in Listing 2 is an adaptation of the code in Listing 1. As compared to the single precision baseline implementations, the following adaptations were required for mixed precision. First, the function prototype must pass the two parts representing the split matrix A, one storing the single precision values AxS and a second part storing double precision values AxD. Similarly, A's auxiliary

```

1  __global__ void spmv_GPU(const unsigned int num_rows, float *y, const float *Ax, const float *x,
2                        const int *Aj, const int *Ap)
3  {
4      const size_t VECS_PER_BLOCK = THRDS_PER_BLOCK / THRDS_PER_VECTOR;
5      __shared__ volatile float sdata[VECS_PER_BLOCK*THRDS_PER_VECTOR+THRDS_PER_VECTOR/2];
6      __shared__ volatile int ptrs[VECS_PER_BLOCK][2];
7      const int thread_id = THRDS_PER_BLOCK * blockIdx.x + threadIdx.x;
8      const int thread_lane = threadIdx.x & (THRDS_PER_VECTOR - 1);
9      const int vector_id = thread_id / THRDS_PER_VECTOR;
10     const int vector_lane = threadIdx.x / THRDS_PER_VECTOR;
11     const int num_vectors = VECS_PER_BLOCK * gridDim.x;
12
13     for(int row = vector_id; row < num_rows; row += num_vectors)
14     {
15         if(thread_lane < 2)
16             ptrs[vector_lane][thread_lane] = Ap[row + thread_lane];
17         const int row_start = ptrs[vector_lane][0];
18         const int row_end = ptrs[vector_lane][1];
19
20         float sum = (thread_lane == 0) ? y[row] : 0.0;
21
22         if (THRDS_PER_VECTOR == 32 && row_end - row_start > 32) {
23             int jj = row_start - (row_start & (THRDS_PER_VECTOR - 1)) + thread_lane;
24             if(jj >= row_start && jj < row_end) sum += Ax[jj] * x[Aj[jj]];
25             for(jj += THRDS_PER_VECTOR; jj < row_end; jj += THRDS_PER_VECTOR)
26                 sum += Ax[jj] * x[Aj[jj]];
27         }
28         else{
29             for(int jj = row_start + thread_lane; jj < row_end; jj += THRDS_PER_VECTOR)
30                 sum += Ax[jj] * x[Aj[jj]];
31         }
32
33         sdata[threadIdx.x] = sum;
34         float tmp;
35         if (THRDS_PER_VECTOR > 16) {tmp=sdata[threadIdx.x+16]; sdata[threadIdx.x]= sum+= tmp;}
36         if (THRDS_PER_VECTOR > 8) {tmp=sdata[threadIdx.x+ 8]; sdata[threadIdx.x]= sum+= tmp;}
37         if (THRDS_PER_VECTOR > 4) {tmp=sdata[threadIdx.x+ 4]; sdata[threadIdx.x]= sum+= tmp;}
38         if (THRDS_PER_VECTOR > 2) {tmp=sdata[threadIdx.x+ 2]; sdata[threadIdx.x]= sum+= tmp;}
39         if (THRDS_PER_VECTOR > 1) {tmp=sdata[threadIdx.x+ 1]; sdata[threadIdx.x]= sum+= tmp;}
40
41         if (thread_lane == 0) y[row] = (sdata[threadIdx.x]);
42     }
43 }

```

Listing 1. Baseline single precision kernel code.

indexing structures must also be split, producing AjS , AjD , ApS and ApD . We also pass two versions of dense input vector x , xS in single precision and xD in double precision. We do this so that the single precision portion of the multiplication can be performed in single precision; if we tried to use xD for both computations, the *nvcc* compiler would promote the data type of the single precision computation to double precision rendering the whole kernel calculation a double precision one.

We perform two SpMV partial calculations on the split input matrix. The single precision calculation and the double precision calculation. Multiple threads are computing a row, so a parallel reduction is used to aggregate these partial computations into a final result. To achieve high performance, the partial results and reduction calculation use data in the GPU's shared memory. The accumulation is always performed in double precision.

Note that the partial computations on the split matrices are completely independent, offering opportunities to achieve parallelism across single and double precision floating point units and the


```

1  __global__ void spmv_GPU(const unsigned int num_rows, const float *AxS, const float *xS,
2      const double *xD, const int *AjS, const int *ApS, double *y,
3      const double *AxD, const int *AjD, const int *ApD)
4  {
5      const size_t VECS_PER_BLOCK = THRDS_PER_BLOCK / THRDS_PER_VECTOR;
6      __shared__ volatile double sdata[VECS_PER_BLOCK*THRDS_PER_VECTOR+THRDS_PER_VECTOR/2];
7      __shared__ volatile int ptrsS[VECS_PER_BLOCK][2];
8      __shared__ volatile int ptrsD[VECS_PER_BLOCK][2];
9
10     const int thread_lane = threadIdx.x & (THRDS_PER_VECTOR - 1);
11     const int vector_id = THRDS_PER_BLOCK * blockIdx.x + threadIdx.x / THRDS_PER_VECTOR;
12     const int vector_lane = threadIdx.x / THRDS_PER_VECTOR;
13
14     for(int row = vector_id; row < num_rows; row += VECS_PER_BLOCK * gridDim.x)
15     {
16         if(thread_lane < 2) {
17             ptrsS[vector_lane][thread_lane] = ApS[row + thread_lane];
18             ptrsD[vector_lane][thread_lane] = ApD[row + thread_lane];
19         }
20         const int row_startS = ptrsS[vector_lane][0];
21         const int row_endS = ptrsS[vector_lane][1];
22         const int row_startD = ptrsD[vector_lane][0];
23         const int row_endD = ptrsD[vector_lane][1];
24
25         double sum = (thread_lane == 0) ? y[row] : 0.0;
26
27         if (THRDS_PER_VECTOR == 32 && row_endS - row_startS > 32) { //Single precision portion
28             int jj = row_startS - (row_startS & (THRDS_PER_VECTOR - 1)) + thread_lane;
29             if(jj >= row_startS && jj < row_endS) sum += AxS[jj] * xS[AjS[jj]];
30             for(jj += THRDS_PER_VECTOR; jj < row_endS; jj += THRDS_PER_VECTOR)
31                 sum += AxS[jj] * xS[AjS[jj]];
32         }
33         else {
34             for(int jj = row_startS + thread_lane; jj < row_endS; jj += THRDS_PER_VECTOR)
35                 sum += AxS[jj] * xS[AjS[jj]];
36         }
37
38         //Double precision portion omitted to save space. It is similar
39         //to the single precision code, but uses xD, AxD, AjD, and *ApD instead
40
41         sdata[threadIdx.x] = sum;
42         double tmp;
43         // Reduction of local sums to row sum omitted to save space.
44         // It is similar to single precision code in Listing~1.
45
46         if (thread_lane == 0) y[row] = double(sdata[threadIdx.x]);
47     }
48 }

```

Listing 2. Mixed precision SpMV kernel code excerpt.

independent integer units of the V100 [33]. Overall, we find that this implementation achieves the optimization goals of mixed precision: increased parallelism, reduced computation time, reduced data movement due to smaller matrix sizes and, in some cases, improved data locality.

5 EXPERIMENTAL RESULTS AND EVALUATION

This section presents performance and accuracy results for MpSpMV. We begin by describing the experimental methodology.

5.1 Experimental Methodology

We describe the target architectures, input matrices and how they were chosen, and approach to data collection.

5.1.1 Target Architecture. The primary target architecture for these experiments is the Nvidia V100 GPU, part of the University of Utah Center for High Performance Computing (CHPC) Notchpeak cluster; each node of Notchpeak has a 32-core Intel Xeon Gold 6130 CPU running at 2.10 GHz and 192 GB of memory. The V100 GPU has 16 GB of global memory and peak memory bandwidth of 900 GB/s. Its peak double precision performance is 7 TFlops and peak single precision performance is 14 TFlops. The architecture has compute capability 7.0; CUDA 9.2 was used to compile and execute all codes.

For comparison, we also include performance result for the P100 GPU from the CHPC Kingspeak cluster; each node of Kingspeak has two 14-core Intel Broadwell processors E5-2680 v4 running at 2.4 GHz and 256 GB of memory. The Tesla P100 GPU Pascal generation has 16 GB global memory and peak memory bandwidth of 732 GB/s. Its peak double precision performance is 4.7 TFlops and single precision performance is 9.3 TFlops. The architecture has compute capability 6.0; CUDA 9.2 was also used to compile and execute all codes.

ALGORITHM 2: Sparse matrix selection process for maximizing mixed precision usefulness.

Input: Sparse matrix A

Output: True if MpSpMV can be used else False

```

1  if  $A \in \mathbb{R}^{n \times n}$  then
2      if  $A$  is totally inside the range then
3          use Sp;
4      else
5          if  $A$  is totally outside the range then
6              use Dp
7          else
8              return True, MpSpMV is useful;
9          end
10     end
11 else
12     return False, MpSpMV can't be used;
13 end

```

5.1.2 Input Matrices. For this experiment, we wanted to explore a cross-section of the matrices with real values in the Sparse Matrix Collection, available online [13]. To select among these matrices, we first examined the entire collection of 2,851 matrices using Algorithm 2. This algorithm filters out matrices whose values are not floating point and marks matrices for which all values should be either double or single precision. The results of this analysis are shown in Table 3. We see that 1,359 of the 2,202 real-valued matrices are candidates for the mixed precision optimization; 101 of these can be computed in single precision only.

We then identified the largest matrices in the collection for which mixed precision is useful, as these have sufficient computation to keep the floating point units busy and significant data movement that makes reuse in cache unlikely. In other words, they are most likely to strain the GPU resources. For this purpose, we started with the largest 100 sparse matrices from the Sparse Matrix Collection. This set was identified by sorting the sparse matrix collection and choosing the matrices with the highest number of nonzeros. Four of these were too large to fit in the memory

Table 3. Data-type Breakdown for the 2,851 Sparse Matrices from the Sparse Matrix Collection

#	Data Type	Number of Matrices
1	Binary	600
2	Complex	49
3	Real values in single range	944 (101+843)(Sp+Dp)
4	Real values in dual range	1,258
Total 2,851

Fig. 6. Performance speedup of mixed precision over double precision SpMV on V100 for all 2,802 sparse matrices.

capacity of our GPU cluster, and after filtering out only real-valued matrices suitable for mixed precision, we were left with 29 matrices, for which we will take a cursory look at speedup.

Unfortunately, these large matrices only traverses a small subset of application areas available from the online Sparse Matrix Collection. Therefore, for the detailed analysis in the article, we used the k -dimension algorithm [9] with four dimensions to select a subset of matrices that are expected to be more representative of the collection. The four dimensions we used were as follows:

- (i) Mixed precision SpMV speedup over double precision SpMV
- (ii) Number of nonzeros in a sparse matrix
- (iii) Density of the sparse matrix
- (iv) Ratio of nonzero values inside the range $(-1,1)$

Using this approach, we derived 32 representative matrices from the 2,802 real-valued and binary matrices.

As speedup was one of the partitioning criteria, we measured mixed precision speedup for all 2,802 matrices, comparing mixed precision performance over double precision performance; these results are shown in the scatter plot of Figure 6 (see Section 5.1.3 for how we obtained these

measurements). On all real-valued matrices, mixed precision obtains average speedup of $1.06\times$ over double precision and maximum speedup of $2.61\times$, while single precision CSR-vector SpMV maximum speedup was $2.84\times$ and average of $1.13\times$. Note that 101 of the real-valued matrices can be represented in single precision as all of their values are in the range $(-1,1)$. These, too, benefit from mixed precision in identifying that only single precision is needed. For exclusively single precision calculations resulting from mixed precision analysis, we obtained a maximum speedup of $2.61\times$ and average speedup of $1.12\times$. Overall, 61% of sparse matrices we ran obtained a speedup with mixed precision. Moreover, our mixed precision version of the code maintained higher accuracy compared to single precision.

Table 4 shows the name, number of rows, number of nonzero values, application domain, and double precision performance on V100 for the selected 32 sparse matrices.

5.1.3 Data Collection. The mixed precision code of Listing 2 has parameters that, when used in CUSP, are adjusted at runtime based on the structure of the input matrix. These derive the thread and block decomposition for GPU execution. CUSP defines four variables for the assignment of block and thread decomposition:

- `THREADS_PER_BLOCK`, which is used to specify the kernel's block size, is preset to 128.
- `THREADS_PER_VECTOR`, which is the number of threads used for a given row of the matrix.
- `VECTORS_PER_BLOCK` is equal to `THREADS_PER_BLOCK` divided by `THREADS_PER_VECTOR`, identifying how many rows are computed in a block.
- `NUM_BLOCKS`, the kernel's grid size, is the minimum between (1) maximum number of active blocks allowed on a device and (2) number of rows of the input matrix divided by `VECTORS_PER_BLOCK`.

We record the execution time of 500 SpMV kernel computations, and we report the average of the 500 runs. Our performance comparisons are across different implementations of the GPU kernels. Comparing execution time of the kernels themselves is appropriate as the time for transferring data between host (CPU) and device (GPU) memory in iterative solvers occurs at the beginning and end of iterations, which renders transfer overhead over large number of SpMV operations negligible. This approach is consistent with how any matrix representation transformation is supported in standard libraries such as CUSP. As the relevant data is stored on the device memory for the duration of the computation for many applications, the preprocessing time is negligible. For example, in the conjugate gradient method, the transfer overhead is amortized over all iterations of the solver.

The matrix is split once on the CPU prior to execution of the solver. In the implementation used in this article, the split is happening sequentially on the CPU, and we did not spend any effort to optimize it. Nevertheless, for the sake of completeness, we recorded the time it takes to split the matrix in mixed precision experiments and computed how many iterations are required to reach the break even cost. We compare the time required to split a matrix into its single and double precision parts with the difference between double precision execution time and mixed precision execution time to figure out how many iterations are needed to amortize the cost of the split. The range of required iterations was from 15 to 9,707 on a V100 GPU. We recorded these split times using only one split iteration and not averaged over multiple iterations as that is the way the program would run in a real life situation.

For all of the experiments conducted, we used the same random seed to populate the x-vectors. We tested several random seeds, and we did not see any noticeable differences on correctness of the final result or on performance. We also studied the impact of increasing the magnitude of random

Table 4. 32 Matrices Used from the Total Number of Sparse Matrices

#	Matrix	Rows	NNZ	Application Area	Dp Gflops
1	mark3jac100sc	45,769	268,563	Economic Problem	63.4
2	scagr7-2r ¹	32,847/46,679	120,141	Linear Programming Problem	48.64
3	c-73	169,422	1,279,274	Optimization Problem	63.42
4	para-4	153,226	2,930,882	Semiconductor Device Problem	73.75
5	rdb968	968	5,632	Computational Fluid Dynamics	2.85
6	fs_760_3	760	5,816	Subsequent 2D/3D Problem	2.63
7	oscl_dcop_42	430	1,544	Subsequent Circuit Simulation	0.54
8	PR02R	161,070	8,185,136	Computational Fluid Dynamics	110.6
9	celegans_metabolic	453	4,065	Undirected Multigraph	0.68
10	cavity01	317	7,280	Computational Fluid Dynamics	3.72
11	c-54	31,793	385,987	Optimization Problem	13.24
12	freeFlyingRobot_3	1,718	12,922	Optimal Control Problem	1.29
13	bcsstk04	132	3,648	Structural Problem	0.34
14	ms2010	171,778	839,980	Undirected Weighted Graph	47.8
15	bcircuit	68,902	375,558	Circuit Simulation	72.44
16	oscl_dcop_12	430	1,544	Subsequent Circuit Simulation	0.47
17	bcsstk03	112	640	Structural Problem	0.07
18	ASIC_320ks	321,671	1,316,085	Circuit Simulation Problem	58.81
19	GD01_a	311	645	Directed Weighted Graph	0.17
20	oscl_dcop_56	430	1,544	Subsequent Circuit Simulation	0.42
DOUBLE PRECISION VALUES ONLY					
21	mycielskian18	196,607	300,933,832	Undirected Graph	95.09
22	halfb	224,617	12,387,821	Structural Problem	98.85
23	rgg_n_2_17_s0	131,072	1,457,506	Undirected Random Graph	60.45
24	pkustk05	37,164	2,205,144	Structural Problem	79.64
25	pkustk02	10,800	810,000	Structural Problem	80.66
26	coAuthorsDBLP	299,067	1,955,352	Undirected Graph	51.41
27	G19	800	9,322	Undirected Weighted Random Graph	2.44
28	germany_osm	11,548,845	24,738,362	Undirected Graph	32.48
29	nasa2910	2,910	174,296	Structural Problem	38.32
30	GD96_c	65	250	Undirected Graph	0.02
31	dwt_607	607	5,131	Structural Problem	1.22
32	lshp_265	265	1,753	Thermal Problem	0.28

¹ Not a square shaped matrix.

values in the x-vector using the following ranges [1, 10, 100, 1000] on both execution time and accuracy. We observed no significant change in either accuracy of the final result or the execution time.

We validate correctness of the output as compared to both a CPU double precision version and with respect to GPU double precision execution by examining each element of the output vector, as discussed in Section 5.3.

5.2 Performance Evaluation

We begin the performance evaluation discussion by simply examining the speedup of mixed precision code as compared to double precision. For reference, we also show the speedup of using single precision over using double precision (ignoring accuracy). We first show speedup for the

Fig. 7. Performance speedup of single and mixed precision over double precision SpMV on V100 for the largest 29 real-valued sparse matrices.

29 largest matrices, to see whether our hypothesis that data movement reduction is the primary benefit of mixed precision. These results are shown in Figure 7. We see that most of these matrices benefit from mixed precision, and only hardesty slows down as compared to double precision. It is notable that this matrix also has rather low gains from single precision, and so little opportunity for improvement coupled with additional overhead of mixed precision likely influences the slow-down. Three other matrices have effectively the same performance as mixed precision, and 25 of the 29 achieve speedup.

Now we perform a detailed analysis of the representative matrices listed in Table 4. As 12 of these are double precision only and speedup is 1 $\times$, we only show the 20 matrices that use mixed precision on the V100 in Figure 8. We see that 11 of the matrices exhibit speedup using mixed precision, and one of these, mark3jac100sc achieves a speedup of 1.90 $\times$. For those that do not speedup, mixed precision performance is very close to double precision, ranging from 0.91 $\times$ to 1.0 $\times$. In three cases, rdb968, fs_76_3, and oscil_dcop_42, mixed precision even outperforms single precision arithmetic.

For comparison, we show performance on the previous generation P100 GPU in Figure 8. On this previous architecture, the overheads of mixed precision eliminate speedup for 4 of the matrices. This is also partially due to the fact that even the single precision speedups are reduced on the P100. Interestingly, speedup for fs_760_3 is over 1.2 $\times$ on the P100 and, like the V100, outperforms single precision, as does oscil_dcop_42.

To tease out the difference between single and mixed precision performance, we collected functional unit and memory statistics from the nvprof tool (see Listing 3 for invocation) and results of splitting each matrix into single and double precision components. These results are presented in Table 5.

Table 5 shows percentage of instruction counts for both single and double precision instructions generated by the nvcc compiler for a single CSR-SpMV iteration. The table also shows the decomposition of the original sparse matrix into two sub-matrices: one that holds values in single

Fig. 8. Performance speedup of both single precision and mixed precision over double precision SpMV on V100 and P100.

```
1 nvprof --metrics all (Kernel_name) (Arguments)
```

Listing 3. Nvidia profiler nvprof command used to obtain values shown in columns 3 and 4 of Table 5 and in Figures 9 and 10.

precision and another holding values represented in double precision. As can be seen, the percentage of double precision instructions dominate single precision instructions, and this makes total sense if we take a look at the kernel code in Listing 2. To force single precision execution of a portion of the computation and limit overhead of mixed precision, we perform the multiplication of $A \times S$ by xS in single precision. The result of this product is converted to double precision, added and stored to a double precision shared array in global memory. Then the mixed precision code performs a fused multiply add (FMA instruction) of $A \times D$ with xD and stores the results in the same shared array in global memory. The reduction sum on the partial results uses double precision arithmetic. So as we can see, single precision floating point arithmetic addition is used once for multiplication for each single precision nonzero, which explains the high percentages of double precision instruction counts in Table 5.

Table 5 also shows the reduction in storage using mixed precision as compared to double precision, calculated using the formulas from Table 2. All but a few matrices show reduced storage requirements; a few that can mostly be represented with single precision are substantially reduced. The ones that are slightly larger than double precision mostly require double precision except for a small portion in single precision, and so the overhead of the split indexing matrices and extra copy of dense input x are the reason.

To understand the reasons behind the performance gains in these results, we examined the output of nvprof and show some of these results in Figures 9 and 10. Overall, mixed precision code introduces overheads and benefits, and sometimes the benefits outweigh the overheads but not always. The benefits over double precision include much higher hit rates in L1 cache as compared to double precision. These are usually somewhere between single and double precision results,

Table 5. Percentage of Floating Point Instruction and Split Ratio for Matrices in Detailed Study

#	Matrix	%IC Sp	%IC Dp	% NNZ Sp	%NNZ Dp	Storage (Dp/Sp)	Storage (Dp/Mp)
1	mark3jac100sc	3.60	96.40	44.18	55.82	1.53	1.02
2	scagr7-2r	0.52	99.48	10.07	89.93	1.55	1.09
3	c-73	5.41	94.59	56.72	43.28	1.53	1.05
4	para-4	5.65	94.35	21.71	78.29	1.51	1.26
5	rdb968	7.69	92.31	0.01	99.99	1.53	0.92
6	fs_760_3	0.98	99.02	9.94	90.06	1.53	1.19
7	oscil_dcop_42	2.30	97.70	45.60	54.40	1.55	1.00
8	PR02R	9.47	90.53	30.85	69.15	1.50	1.27
9	celegans_metabolic	7.49	92.51	68.44	31.56	1.52	1.03
10	cavity01	1.39	98.61	6.63	93.37	1.51	1.35
11	c-54	2.74	97.26	19.71	80.29	1.52	1.22
12	freeFlyingRobot_3	5.09	94.91	53.53	46.47	1.53	1.06
13	bcsstk04	21.57	78.43	6.91	93.09	1.51	1.00
14	ms2010	6.63	93.37	0.01	99.99	1.54	0.91
15	bcircuit	2.11	97.89	29.06	70.94	1.54	1.10
16	oscil_dcop_12	2.39	97.61	47.34	52.66	1.55	0.99
17	bcsstk03	7.53	92.47	0.62	99.38	1.53	0.92
18	ASIC_320ks	0.00	99.99	0.00	99.99	1.54	0.94
19	GD01_a	0.01	99.99	0.16	99.84	1.57	1.00
20	oscil_dcop_56	2.30	97.70	45.60	54.40	1.55	1.00
DOUBLE PRECISION VALUES ONLY							
21	mycielskian18	0.00	100.00	0.00	100.00	1.50	1.00
22	halfb	0.00	100.00	0.00	100.00	1.50	1.00
23	rgg_n_2_17_s0	0.00	100.00	0.00	100.00	1.52	1.00
24	pkustk05	0.00	100.00	0.00	100.00	1.50	1.00
25	pkustk02	0.00	100.00	0.00	100.00	1.50	1.00
26	coAuthorsDBLP	0.00	100.00	0.00	100.00	1.53	1.00
27	G19	0.00	100.00	0.00	100.00	1.52	1.00
28	germany_osm	0.00	100.00	0.00	100.00	1.57	1.00
29	nasa2910	0.00	100.00	0.00	100.00	1.50	1.00
30	GD96_c	0.00	100.00	0.00	100.00	1.55	1.00
31	dwt_607	0.00	100.00	0.00	100.00	1.53	1.00
32	lshp_265	0.00	100.00	0.00	100.00	1.53	1.00

although in the case of bcsstk03 and bcsstk04 it is actually higher than single precision, presumably because the pattern of data accesses is more regular in the split matrix. Mixed precision usually also achieves higher instructions per cycle than double precision; for five of the matrices, it achieves higher IPC than single precision. The V100 has separate integer and floating point units that can execute in parallel; the mixed precision code has two streams of data (single and double) that may be taking advantage of this parallelism.

So what are the scenarios where mixed precision sees a slight slowdown? The smaller matrices are less likely to see a significant speedup, perhaps because they are less limited by memory bandwidth. For bcircuit, which is extremely sparse, mixed precision requires actually more data movement than double precision both from DRAM and from L2 for misses in L1. For

Fig. 9. Instructions per cycle for the 20 matrices selected for detailed analysis, as measured by nvprof.

Fig. 10. The global hit rate in first level cache for the 20 matrices selected for detailed analysis, as measured by nvprof.

freeflyingRobot_3, which has low performance, there is limited warp-level parallelism as compared to other matrices, with very little improvement in eligible warps per cycle over double precision.

We also analyzed why the three matrices rdb968, fs_76_3, and oscil_dcop_42 all showed speedup of mixed precision over single precision. These three matrices are all relatively small

(roughly 5K nonzeros), and overhead due to extra type conversion instructions not present in double precision code are impacting performance, something that would not be a significant factor for large matrices. These instructions are reduced in the mixed precision code.

5.3 Accuracy Evaluation

In our accuracy evaluation, we use the concept of significant digits explained in reference [5]. If the error $X_\tau - X_A$ has magnitude less than or equal to 5 in the $(t + 1)$ th digit of X_τ , counting to the right from the first nonzero digit in X_τ , then we say that X_A has t significant **decimal** digits with respect to X_τ . For example:

$$\begin{aligned} x_\tau &= \frac{1}{9} \\ x_A &= 0.111 \\ |x_\tau - x_A| &= 0.00011 \end{aligned} \tag{4}$$

Since the error is less than 5 in the fourth digit to the right of the first nonzero digit in x_τ , we say that x_A has three significant digits with respect to x_τ . We use this same concept to show accuracy results in Tables 6 and 7; each column in both tables is a significant decimal digit of accuracy, and since single precision can only provide up to six decimal digits of accuracy compared to double precision while mixed precision is expected to provide less accuracy than double but more accuracy than single precision, we show up to eight significant decimal digits of accuracy. The inf column shows the number of values that poses at least nine significant decimal digit of accuracy or more, the larger the number in the last column the more accurate the results are compared to double precision. The addition of all values across a row in either table equals the number of columns of the corresponding sparse matrix.

From Tables 6 and 7, we see that in general mixed precision provides more decimal digits of accuracy compared to single precision. Even in the cases where we did not see any performance gains from using mixed precision, we did see an improvement in accuracy of the final result. Our results conform with the expectation that single precision maintains around six decimal digits of accuracy compared to double precision as mentioned in the introduction.

We also performed experiments to study the effect of widening the split range on the performance and accuracy of mixed precision results compared to double precision. We used increasing split ranges $[\frac{1}{8}, \frac{1}{4}, \frac{1}{2}, 1, 8, 16, 32, 64, 128, 256, 512]$ for the 32 matrices, and the results behave as expected. As the split range increases, the final result becomes less accurate (further away from the double precision result). This effect is due to increase of nonzero values that are now represented in single precision format instead of double precision.

6 RELATED WORK

In this section, we go over several approaches researchers have devised to optimize floating point arithmetic while maintaining accuracy, and we summarize SpMV survey work.

6.1 Numerical Approach to Mixed Precision

Alfredo et al. [10] extended the use of iterative refinement that is used to solve dense linear systems to also solve sparse linear systems, both direct and iterative solvers. The main idea is to improve the performance of the most computationally intensive tasks by exploiting single precision operations and only using double precision at critical stages to maintain the 64-bit accuracy of the final solution. They tested their implementation of a mixed precision sparse direct solver using seven different platforms on a test suite of 41 matrices taken from the University of Florida's Sparse Matrix Collection [13]. The matrices were selected randomly from the collection, since there is

Table 6. This Table Shows the Number of Significant Decimal Digits When Using Single Precision Compared to Double Precision Values When Computing CSR-vector SpMV

#	Matrix	0	1	2	3	4	5	6	7	8	inf
1	mark3jac100sc	0	0	0	1	68	2,854	24,256	14,535	1,483	2,572
2	scagr7-2r	0	0	0	0	0	241	7,795	6,321	279	18,211
3	c-73	0	0	0	0	26	1,443	90,029	69,057	7,936	931
4	para-4	0	2	14	195	2,223	25,644	90,526	31,092	3,185	345
5	rdb968	0	0	0	0	0	51	604	287	21	5
6	fs_760_3	0	0	0	0	0	6	248	239	64	203
7	oscl_dcop_42	0	0	0	0	3	29	206	80	13	99
8	PR02R	0	1	8	155	2,828	32,739	92,165	29,598	3,206	370
9	celegans_metabolic	0	0	0	0	0	1	227	190	8	27
10	cavity01	0	0	0	0	0	33	158	110	14	2
11	c-54	0	0	0	0	3	1,840	18,308	10,315	1,180	147
12	freeFlyingRobot_3	0	0	0	0	5	133	1,010	506	56	8
13	bcsstk04	0	0	0	0	0	12	88	31	1	0
14	ms2010	0	0	0	0	0	2,045	101,589	60,495	6,887	762
15	bcircuit	0	0	0	7	502	5,848	31,001	20,937	7,015	3,592
16	oscl_dcop_12	0	0	0	0	3	37	206	88	15	81
17	bcsstk03	0	0	0	0	2	7	70	31	1	1
18	ASIC_320ks	0	0	0	0	0	2	199	339	25,410	295,721
19	GD01_a	0	0	0	0	0	0	50	66	2	193
20	oscl_dcop_56	0	0	0	0	3	29	207	78	14	99
21	mycielskian18	0	0	0	0	0	1,150	109,290	76,459	8,751	957
22	halfb	0	0	0	0	0	6	109,347	102,433	11,460	1,371
23	rgg_n_2_17_s0	0	0	0	0	0	175	85,612	40,293	2,905	2,087
24	pkustk05	0	0	0	0	0	0	18,210	16,662	1,986	306
25	pkustk02	0	0	0	0	0	0	6,324	4,122	336	18
26	coAuthorsDBLP	0	0	0	0	0	123	99,564	128,500	6,200	64,680
27	G19	0	0	0	0	0	35	436	268	21	40
28	germany_osm	0	0	0	0	0	0	3,110,508	5,458,210	80,072	2,900,055
29	nasa2910	0	0	0	0	0	0	1,458	1,321	122	9
30	GD96_c	0	0	0	0	0	0	16	42	0	7
31	dwt_607	0	0	0	0	0	0	348	227	19	13
32	lshp_265	0	0	0	0	0	0	131	120	7	7

no information available about their condition number. For their tests, they used the MUMPS [4] software package and SuperLU [27], since only those two packages are implemented in both single and double precision. This was not the case for other solvers such as UMFPACK [12] at that time. It is notable that there is a penalty for using this method that is an increase of $1.5\times$ in storage of mixed precision compared to the standard double precision solution method.

Azzam et al. developed a framework [23] for accelerating the general $A * x = b$ solver, where A is a large dense matrix, using hardware-accelerated FP16 arithmetic on GPUs. Their approach is based on the mixed-precision iterative refinement technique. The DSGESV (FP32 $\rightarrow$ FP64) routine performs the LU factorization in single precision (FP32) and uses the preconditioned FGMRES (GM method) to achieve the FP64 solution to system of linear equations $A * x = B$ for general matrices. They conclude that dsgesv can be used for all matrix types and for any sizes, and it provides a

Table 7. This Table Shows the Number of Significant Decimal Digits When Using Mixed Precision Compared to Double Precision Values When Computing CSR-vector SpMV

#	Matrix	0	1	2	3	4	5	6	7	8	inf
1	mark3jac100sc	0	0	0	2	39	2,131	19,110	16,795	2,600	5,092
2	scagr7-2r	0	0	0	0	0	156	1,727	1,151	195	29,618
3	c-73	0	0	0	0	18	465	65,911	89,261	12,294	1,473
4	para-4	0	1	12	174	1,873	19,787	80,988	41,914	4,890	3,587
5	rdb968	0	0	0	0	0	52	552	330	33	1
6	fs_760_3	0	0	0	0	0	1	183	187	30	359
7	oscl_dcop_42	0	0	0	0	3	37	169	84	15	122
8	PR02R	0	0	4	112	1,954	23,957	84,868	41,028	4,938	4,209
9	celegans_metabolic	0	0	0	0	0	0	36	250	66	101
10	cavity01	0	0	0	0	0	18	107	58	15	119
11	c-54	0	0	0	0	0	1,059	14,397	11,672	2,111	2,554
12	freeFlyingRobot_3	0	0	0	0	2	95	768	664	78	111
13	bcsstk04	0	0	0	0	0	6	71	50	5	0
14	ms2010	0	0	0	0	0	0	36,053	111,328	21,903	2,494
15	bcircuit	0	0	0	6	499	5,586	22,529	10,005	1,254	29,023
16	oscl_dcop_12	0	0	0	0	3	40	163	87	14	123
17	bcsstk03	0	0	0	0	2	6	49	48	7	0
18	ASIC_320ks	0	0	0	0	0	2	6	3	1	321,659
19	GD01_a	0	0	0	0	0	0	0	1	0	310
20	oscl_dcop_56	0	0	0	0	3	37	169	84	15	122
21	mycielskian18	0	0	0	0	0	0	0	0	0	196,607
22	halfb	0	0	0	0	0	0	0	0	0	224,617
23	rgg_n_2_17_s0	0	0	0	0	0	0	0	0	0	131,072
24	pkustk05	0	0	0	0	0	0	0	0	0	37,164
25	pkustk02	0	0	0	0	0	0	0	0	0	10,800
26	coAuthorsDBLP	0	0	0	0	0	0	0	0	0	299,067
27	G19	0	0	0	0	0	0	0	0	0	800
28	germany_osm	0	0	0	0	0	0	0	0	0	11,548,845
29	nasa2910	0	0	0	0	0	0	0	0	0	2,910
30	GD96_c	0	0	0	0	0	0	0	0	0	65
31	dwt_607	0	0	0	0	0	0	0	0	0	607
32	lshp_265	0	0	0	0	0	0	0	0	0	265

speedup of about $1.7\times$. The DHGESV (FP16 $\rightarrow$ FP64) routine, which performs the LU factorization in half precision (FP16) and uses the preconditioned FGMRES (GM method) to achieve the FP64 solution, is advantageous when the matrix has nice properties, such as diagonal dominance, or when eigenvalues are always positive. In this latter case, they expect a factor of $2.5\times$ to $2.7\times$ speedup. Based on their analysis, they predict that (FP16 $\rightarrow$ FP64) will also be beneficial in the case of the Cholesky factorization. They plan to release their framework through the MAGMA library. In more recent work [22], Azzam et al. use half-precision and double-precision mixed-precision operations to utilize the GPU's Tensor cores that support half precision arithmetic to speed up iterative refinement solvers.

Piotr Luszczek et al. [29] show that numerical implementation of building blocks in benchmark procedures can use a half precision data type on modern accelerator hardware. Li Xiaoye et al.

[28] describe the design rationale, a reference C implementation, and testing code for 10 routines in the Extended and Mixed Precision BLAS namely, inner product (DOT), sum, scaled vector accumulation (AXPBY), scaled vector addition (WAXPBY), general matrix vector product (GEMV), banded matrix vector product (GBMV), symmetric matrix vector product (SPMV), triangular solve (TRSV), general matrix matrix product (GEMM), symmetric matrix matrix product (SYMM). Mixed precision is also widely used in deep learning applications [14, 18, 31, 32], specifically at the output feature map computation phase, a large number of accumulations are required where rounding errors are also accumulated, and to reduce this error, accumulation using higher precision is suggested.

6.2 Heuristic Approach to Mixed Precision

Rubio-Gonzales et al. [34] used a sampling-based approach to tune codes using a tool they developed called Precimonious. They require the user to select a small subset of inputs from input intervals of interest. The user also has to provide an error threshold for the final program outputs. They heuristically consider different operator precision allocations and rerun the whole program on the given inputs, until all of them satisfy the user-provided error threshold. Given the huge number of inputs available within typical input intervals of interest and the sensitivity of floating-point error to actual data inputs, no guarantees are produced or implied for the remaining, unexplored inputs.

6.3 Non-standard Approach to Floating Point Arithmetic

A different approach is to reinvent the representation of floating point numbers, work by Gustafson et al. includes type-1 [20] and type-2 [19] unums, as well as their fixed-precision successor, posits [21]. These representations show a good tradeoff between precision and dynamic range in the floating point representation, which increases the accuracy per bit stored. However, there is currently no hardware vendor support for these representations.

6.4 SPMV

Langr et al. surveyed a total of 51 sparse matrix storage formats [25]. They mention that authors of new storage formats usually focus on the performance of sparse matrix-vector multiplication (SpMV) in FLOPS as the only evaluation criterion, even though such an evaluation is essential, it is not enough to compare the format with other existing formats. They establish 10 evaluation criteria for sparse matrix storage formats, and provide general suggestions to make storage format developers work more valuable for the HPC community.

Stefano Markidis et al. [30] use Tensor cores in V100 Nvidia GPUs to speed up the performance of GEMM by $6\times$ for large matrices and $1.2\times$ – $2.5\times$ when multiplying many small matrices in parallel. They then discuss programmability, performance, and precision of using Tensor cores for HPC applications. For programmability, they mention three programming strategies: (1) CUDA 9 WMMA API provides direct access to CUDA Tensor Cores; (2) indirectly, using CUTLASS and cuBLAS libraries; and (3) viewing Tensor cores from a high-level point of view as accelerators within an accelerator. For performance in their test environment, they obtain a maximum of 83 Tflops/s with cuBLAS GEMM. The naive GEMM implementation with CUDA WMMA did not lead to any performance improvement; however, when using the implementation with CUDA shared memory, they measured a $5\times$ performance improvement. They note that memory traffic still has high impact on the overall performance of the matrix multiplications. For precision, their matrix multiplication inputs are in half precision. Precision loss occurs when they are rounded from single to half precision and is considerable when using large input values. When input matrix size increases, error increases, because $O(N^2)$ operations are required to calculate a matrix entry in the matrix

multiplication. They used a simple method to decrease precision loss by taking into account the rounding error when converting values from single to half precision. This method reduces the precision loss at the cost of increased computation. They also note that the measured Tensor Core maximum performance is 74% the theoretical peak. Because precision loss might be an obstacle for the uptake of NVIDIA Tensor Cores in HPC, they showed that it is possible to decrease it at the cost of increased computation.

7 CONCLUSION AND FUTURE WORK

This article demonstrated the effectiveness of MpSpMV, a mixed precision implementation of SpMV for GPUs. We executed MpSpMV on Nvidia V100 GPU on matrices from the Sparse Matrix Collection and compared its performance and accuracy to single precision and double precision counterparts. The key finding is that MpSpMV can outperform double precision SpMV due to reduced computation time, increased parallelism, and reduced data movement. At the same time, MpSpMV maintains acceptable accuracy due to the partial use of double precision values and accumulation of partial results in progressively larger data-type variables.

As a continuation of this work, we are planning to automate the CUDA code generation and the format conversion for MpSpMV as in prior work [1]. Specifically, we previously employed compiler technology to optimize LOBPCG solver and show that it outperforms manually tuned code for the same algorithm [3]. Using an inspector-executor approach, we used data transformation to convert a massive symmetric sparse matrix from a CSR format to a compressed sparse block (CSB) format. To reduce the data movement associated with indices of the matrices, a short integer was used as the type for the matrices that pointed to the beginning of each CSB block. Targets for AVX SIMD code generation were marked with pragmas for the native Intel ICC compiler.

As another direction for this work, MpSpMV can be extended to support new precision types such as half precision (as used in the V100 tensor cores) and double-double precision [24]. We will also study the benefits of using our mixed precision on other sparse matrix representations and kernels such Diagonal and Ellpack formats. We will further consider other sparse matrix computations, such as multiple vector multiplication SpMM [36]. In this computation, the sparse matrix is also reused in the multiplication; consequently, any optimization made on the sparse matrix will have a larger effect on the total performance of the computation.

Using automated data transformation, we could expand on the underlying data-driven concept in this work in new directions related to sparse tensor computations. For example, instead of lowering precision, we could zero out small values that have little effect on the final answer, due to the reduction sum computational stage occurring near the end of the SpMV algorithm as explained in Section 3. Zeroing out values can also be used by deep learning intermediate steps where tensor values and weights can be determined to be unimportant to the computation.

REFERENCES

- [1] Khalid Ahmad, Anand Venkat, and Mary Hall. 2017. Optimizing LOBPCG: Sparse matrix loop and data transformations in action. In *Languages and Compilers for Parallel Computing*, Chen Ding, John Criswell, and Peng Wu (Eds.). Springer International Publishing, Cham, 218–232.
- [2] H. M. Aktulga, M. Afibuzzaman, S. Williams, A. Buluc, M. Shao, C. Yang, E. G. Ng, P. Maris, and J. P. Vary. 2017. A high performance block eigensolver for nuclear configuration interaction calculations. *IEEE Trans. Parallel Distrib. Syst.* 28, 6 (Jun. 2017), 1550–1563. DOI: <https://doi.org/10.1109/TPDS.2016.2630699>
- [3] H. M. Aktulga, A. Buluc, S. Williams, and C. Yang. 2014. Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 1213–1222. DOI: <https://doi.org/10.1109/IPDPS.2014.125>
- [4] Patrick R. Amestoy, Iain S. Duff, Jean Yves L'Excellent, and Jacko Koster. 2000. MUMPS: A general purpose distributed memory sparse solver. In *Proceedings of the International Workshop on Applied Parallel Computing*. Springer, 121–130.
- [5] Kendall E. Atkinson. 2008. *An Introduction to Numerical Analysis*. John Wiley & Sons.

- [6] D. H. Bailey. 2005. High-precision floating-point arithmetic in scientific computation. *Comput. Sci. Eng.* 7, 3 (May 2005), 54–61. DOI : <https://doi.org/10.1109/MCSE.2005.52>
- [7] Nathan Bell and Michael Garland. 2008. *Efficient Sparse Matrix-vector Multiplication on CUDA*. Technical Report. Nvidia Technical Report NVR-2008-004, Nvidia Corporation.
- [8] Nathan Steven Bell and Michael Dalton Garland. 2015. CUSP: Generic parallel algorithms for sparse matrix and graph computations, 2015. *Version 0.5.0*. Retrieved March 9, 2015 from <http://cusplibrary.github.io>.
- [9] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.
- [10] Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Piotr Luszczek, and Stanimir Tomov. 2008. Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy. *ACM Trans. Math. Softw.* 34, 4 (2008), 17.
- [11] NVIDIA CUSPARSE. 2019. CUBLAS libraries.
- [12] Timothy A. Davis. 2003. *Umfpack Version 4.1 User Guide*. Department of Computer and Information Science and Engineering, University of Florida (2003).
- [13] Timothy A. Davis and Yifan Hu. 2011. The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. DOI : <https://doi.org/10.1145/2049662.2049663>
- [14] Manuel Le Gallo, Irem Boybat, Bipin Rajendran, Abu Sebastian, Evangelos Eleftheriou, et al. 2017. Mixed-precision training of deep neural networks using computational memory. *arXiv preprint arXiv:1712.01192* (2017).
- [15] L. Gan, H. Fu, W. Luk, C. Yang, W. Xue, X. Huang, Y. Zhang, and G. Yang. 2013. Accelerating solvers for global atmospheric equations through mixed-precision data flow engine. In *Proceedings of the 2013 23rd International Conference on Field programmable Logic and Applications*. 1–6. DOI : <https://doi.org/10.1109/FPL.2013.6645508>
- [16] Dominik Göddeke, Robert Strzodka, and Stefan Turek. 2007. Performance and accuracy of hardware-oriented native-, emulated-and mixed-precision solvers in FEM simulations. *Int. J. Parallel Emerg. Distrib. Syst.* 22, 4 (2007), 221–256.
- [17] Scott Le Grand, Andreas W. Götz, and Ross C. Walker. 2013. SPFP: Speed without compromise—A mixed precision model for GPU accelerated molecular dynamics simulations. *Comput. Phys. Commun.* 184, 2 (2013), 374–380. DOI : <https://doi.org/10.1016/j.cpc.2012.09.022>
- [18] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In *Proceedings of the International Conference on Machine Learning*. 1737–1746.
- [19] John L. Gustafson. 2016. A radical approach to computation with real numbers. *Supercomput. Front. Innov.* 3, 2 (2016), 38–53.
- [20] John L. Gustafson. 2017. *The End of Error: Unum Computing*. Chapman & Hall/CRC.
- [21] John L. Gustafson and Isaac T. Yonemoto. 2017. Beating floating point at its own game: Posit arithmetic. *Supercomput. Front. Innov.* 4, 2 (2017), 71–86.
- [22] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J. Higham. 2018. Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'18)*. IEEE Press, Los Alamitos, CA.
- [23] Azzam Haidar, Panruo Wu, Stanimire Tomov, and Jack Dongarra. 2017. Investigating half precision arithmetic to accelerate dense linear system solvers. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. ACM, 10.
- [24] Yozo Hida, Xiaoye S. Li, and David H. Bailey. 2007. *Library for Double-double and Quad-double Arithmetic*. Technical Report. NERSC Division, Lawrence Berkeley National Laboratory (2007).
- [25] Daniel Langr and Pavel Tvrdik. 2016. Evaluation criteria for sparse matrix storage formats. *IEEE Trans. Parallel Distrib. Syst.* 27, 2 (2016), 428–440.
- [26] Sven Leyffer, Stefan M. Wild, Mike Fagan, Marc Snir, Krishna V. Palem, Kazutomo Yoshii, and Hal Finkel. 2016. Doing moore with less—Leapfrogging Moore’s law with inexactness for supercomputing. *CoRR abs/1610.02606* (2016). arxiv:1610.02606 <http://arxiv.org/abs/1610.02606>.
- [27] Xiaoye S. Li. 2005. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.* 31, 3 (2005), 302–325.
- [28] Xiaoye S. Li, James W. Demmel, David H. Bailey, Greg Henry, Yozo Hida, Jimmy Iskandar, William Kahan, Suh Y. Kang, Anil Kapur, Michael C. Martin, et al. 2002. Design, implementation and testing of extended and mixed precision BLAS. *ACM Trans. Math. Softw.* 28, 2 (2002), 152–205.
- [29] Piotr Luszczek, Jakub Kurzak, Ichitaro Yamazaki, and Jack Dongarra. 2017. Towards numerical benchmark for half-precision floating point arithmetic. In *Proceedings of the High Performance Extreme Computing Conference (HPEC'17)*. IEEE, 1–5.
- [30] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter. 2018. NVIDIA tensor core programmability, performance precision. In *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'18)*. 522–531. DOI : <https://doi.org/10.1109/IPDPSW.2018.00091>

- [31] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. 2017. Mixed precision training. *arXiv preprint arXiv:1710.03740* (2017).
- [32] S. R. Nandakumar, Manuel Le Gallo, Irem Boybat, Bipin Rajendran, Abu Sebastian, and Evangelos Eleftheriou. 2018. Mixed-precision architecture based on computational memory for training deep neural networks. In *Proceedings of the 2018 IEEE International Symposium on Circuits and Systems (ISCAS'18)*. IEEE, 1–5.
- [33] Tesla NVIDIA. 2017. *NVIDIA Tesla V100 GPU Architecture*.
- [34] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. 2013. Precimonious: Tuning assistant for floating-point precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 27.
- [35] Yousef Saad. 2003. *Iterative Methods for Sparse Linear Systems*. Vol. 82. SIAM.
- [36] W. A. Sufah and K. Ahmad. 2014. On implementing sparse matrix multi-vector multiplication on GPUs. In *Proceedings of the 2014 IEEE International Conference on High Performance Computing and Communications, 2014 IEEE 6th International Symposium on Cyberspace Safety and Security, and 2014 IEEE 11th International Conference on Embedded Software and Systems (HPCC-CSS-ICSS'14)*. 1117–1124. DOI: <https://doi.org/10.1109/HPCC.2014.165>

Received May 2019; revised October 2019; accepted November 2019