A Hybrid Format for Better Performance of Sparse Matrix-Vector Multiplication on a GPU

Dahai Guo
dahai@illinois.edu

William Gropp
wgropp@illinois.edu

Luke N. Olson
lukeo@illinois.edu

Abstract

In this paper, we present a new sparse matrix data format that leads to improved memory coalescing and more efficient sparse matrix-vector multiplication (SpMV) for a wide range of problems on high throughput architectures such as a graphics processing unit (GPU). The sparse matrix structure is constructed by sorting the rows based on the row length (defined as the number of non-zero elements in a matrix row) followed by a partition into two ranges: short rows and long rows. Based on this partition, the matrix entries are then transformed into ELLPACK (ELL) or vectorized compressed sparse row (vCSR) format. In addition, the number of threads are adaptively selected based on the row length in order to balance the workload for each GPU thread. Several computational experiments are presented to support this approach and the results suggest a notable improvement over a wide range of matrix structures.

Keywords— SpMV, GPU, EVC-HYB format, adaptive

1 Introduction

Sparse Matrix-Vector multiplication SpMV ($\mathtt{Y} = \beta\mathtt{Y} + \alpha\mathtt{AX}$) represents a significant computational kernel in most iterative methods for solving large, sparse linear systems and eigenvalue problems. The use of these methods in both the physical and data sciences motivates the need for a highly efficient SpMV operation that achieves high performance across a range of problem types. Yet, the performance of the SpMV is often only a small fraction of the peak throughput of a processor. In this paper we propose a modified form of a hybrid format that allows for generally improved efficiencies, particularly for problem types where previous attempts have observed reduced efficiencies.

The SpMV operation on a GPU has been well studied in recent years and several approaches have been introduced to directly address the memory access overheads through the choice of data structure and the thread assignment. The scalar CSR format assigns one GPU thread to each matrix row, but while straightforward the efficiency on the GPU remains low. In contrast, the vectorized CSR format employs a warp of GPU threads to handle each row and yields an efficient SpMV operation if the number of nonzero elements in each row is both moderately large and similar from row to row. However, for matrices having few nonzero elements per row or having a variable number of nonzero elements per row, as found in data problems or in algebraic preconditioners [5], the vectorized CSR format deteriorates in performance. The ELL format excels at banded matrices by storing the bands directly (padded with zeros as necessary). In addition, columns of the matrix are aligned in groups of sixteen to satisfy the conditions for memory coalescing. To address high variability, it is common to use the coordinate format (COO) with one thread per entry, yielding low memory coalescing, but also consistent (and low) performance. The SpMV based on the COO format on a GPU often takes advantage of a computational primitive termed *segmented reduction* [7] to accumulate the sum from different threads. Thus, the number of operations across GPU threads is more uniform, however it requires additional and complex program logic that results in reduced performance. A full performance evaluation of these matrix data structure choices is presented in [6].

Some tuning approaches for the SpMV on a GPU, such as synchronization-free parallelism, thread mapping, global memory access, and data reuse have also been developed [4], yielding notable performance gains. In addition, a model-driven approach with auto-tuning has been successfully used with a block ELL format [9] to achieve higher throughput. Moreover, tuning the number of threads assigned to each row (TPR) using the vector-CSR format affects the performance of the SpMV on a GPU [21]. If the row lengths in a sparse matrix are relatively uniform, that method may yield a more balanced workload for each GPU thread. However, since the approach uses a *constant* value of TPR for the entire matrix, the gains are limited when it is employed to matrices with a large range of row lengths. In addition, determining the optimal choice of TPR may require an exhaustive search. The jagged diagonal (JAD) and diagonal (DIA) formats also achieve large performance improvements for some matrices [15], compared to the vectorized CSR format, and a storage structure called Compressed Row Segment with Diagonal-pattern (CRSD) is used for matrices with diagonal structure [24]. The numerical results also demonstrate the notable speedups in comparison with optimized implementations of the CSR, ELL, COO, and HYB formats. Finally, the block structure of a matrix may be exploited [22] by noting the relationship between threads and the vector form of the CSR matrix.

There are also many other research and publications, using different methods to accelerate SpMV on GPUs. Most of these approaches implement variants of the ELL or CSR format to reduce the zero paddings, better using threads in the warp, and yielding improved memory coalescing on GPUs [12, 16, 2]. Recently, a SIMD-friendly data format, called SELL-C-σ, was introduced that combines long-standing ideas from GPU and vector computer programming [14]. The advantage of that format is tested and discussed on a variety of hardware platforms, such as Intel MIC and NVIDIA Kepler GPU. Deep insight into the data transfer properties of the format is developed with appropriate performance models, which are based on the idea of the so-called *roofline* performance model [25]. Although it achieves improved performance for some matrices, the format wasn't tested with matrices having very irregular sparse patterns.

In previous work, we developed a straightforward approach to automatically tune SpMV on a GPU [13], which adaptively allocates GPU threads based on the size of each row range in a matrix. The matrix is stored in the CSR format and is first sorted in increasing order based on the row length. The sorted matrix is then partitioned into several row ranges, and each range is assigned with an adaptively chosen TPR value based on the maximum row length in that range. The number of GPU blocks is then calculated to handle the computation in each row range. For example, in our tests we use the value of 512 ($= 2^9$) as the GPU block size (thread number per block); therefore, the matrix is partitioned into ten row ranges so that a TPR value of 2^0, 2^1, ..., 2^i, ..., 2^9 is assigned to a range. The TPR number equals 2^0 for the row range in which each row has at most 8 nonzero elements, 2^1 for the rows between 8 and $8*2$, four for those in the range $[8*2, 8*4]$, etc., and 2^9 for the matrix rows with more than $8*512$ nonzero elements. If there is no row in a certain range, then no GPU block is distributed to that range. Figure 1 illustrates this process. Ten GPU blocks are assigned to the matrix, two with TPR $= 2^0$, three with TPR $= 2^1$, ..., and two with TPR $= 2^4$. The preprocessing is straightforward and significantly reduces the tuning time compared to other approaches such as exhaustively searching for optimal formats or tuning parameters. Furthermore, the cost of this strategy is low in comparison to matrix conversion into more efficient formats. To achieve this, shared memory inside the GPU block is used for collective sum reduction.

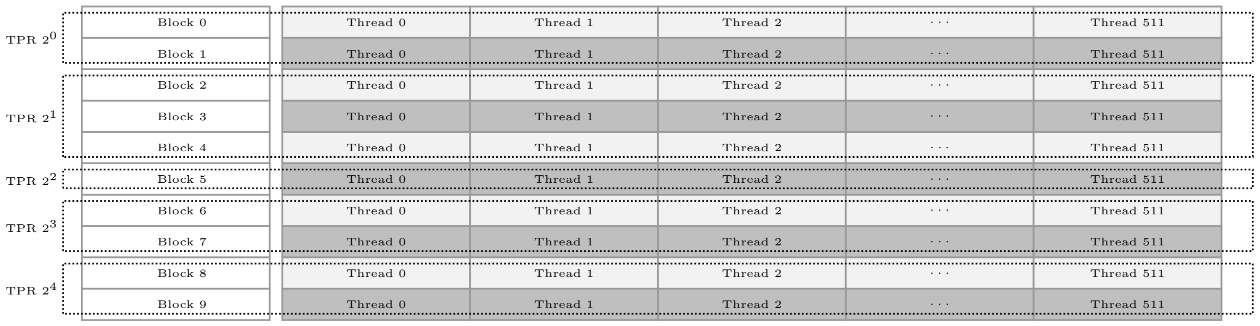

Figure 1: An example of using adaptive threads per row (TPR).

Tests on a NVIDIA Fermi M2070 GPU show that the adaptive method provides improved performance for most of the tested matrices: significant performance improvements are achieved for some matrices that are hard to tune with regular methods. It is also easily implemented because it is based on the CSR format and avoids a matrix conversion. On the other hand, adaptively assigning TPR is of limited value if the structure of the matrix makes it difficult to take advantage of memory coalescing. Indeed, if most of the rows in a matrix are short, this becomes a notable challenge. In order to take advantage of memory coalescing on a GPU, to improve the loading throughput, and to ultimately accelerate the computation of the resulting SpMV operation, we develop a new format in this paper, called the EVC-HYB format. The benefit of this format is two-fold: (1) the format combines the ELL format for short rows and the vectorized CSR format for the longer rows resulting in increased coalescing across a wider range of nonzero patterns, and (2) the format allows for the use of adaptive thread distributions that have previously been shown to be effective.

2 The EVC-HYB Format

Here we consider two important factors that affect the performance of SpMV on a GPU. The first is *memory coalescing*, where a group of consecutive threads access consecutive addresses in the global memory simultaneously. Fully using memory coalescing can significantly increase throughput when the data is read from the global memory. The second is the efficient use of parallel threads with a balanced workload, by which idle threads are reduced or eliminated during the computation.

2.1 Description of the format

The ELL and vectorized CSR formats both work well for certain classes of sparse matrices, but they also yield poor performance for others. The traditional ELL format compresses the matrix data into a rectangular dense matrix, and adds zeros to force each column in the matrix to have the same number of elements. The matrix data is then stored column by column, followed by their column indices in the original sparse matrix. Since each column is aligned to a 32-element boundary, the format is able to fully utilize the mechanism of the memory coalescing on a GPU. This leads to a significant increase in the loading throughput and improved performance of the SpMV for some matrices. The format handles matrices with similar short row lengths well because of the local column storage. However, if the row length for a matrix varies significantly and many zeros are needed, the ELL format becomes inefficient. The `ibm-dc1` matrix is one example, where most of the rows have less than 288 nonzero elements. This matrix also has two rows that have extremely large numbers of nonzero elements: 47193 and 114190 respectively, which is approximately 21% of the total non-zero elements. Even when the matrix is sorted based on the number of non-zero elements per row, the ELL format is not efficient due to the additional zero padding needed for this matrix example. The vectorized CSR format avoids this problem. The vectorized CSR format also takes full advantage of memory coalescing and significantly accelerates performance of the SpMV for matrices where the row lengths are uniform and of sufficient length — for example, approximately multiples of 32 (a warp on a GPU). Yet, there are many matrices that do not satisfy this requirement. For the short rows in a matrix, particularly for those with lengths less than 16, it is difficult for the vectorized CSR format to efficiently utilize the mechanism of memory coalescing and accelerate the computation of the SpMV. Indeed, either too many zeros are added or GPU threads are left idle while others in the same warp are busy.

In order to take advantage of the merits and avoid the weakness from both formats, we develop a hybrid format (EVC-HYB) that combines the two formats together. In the first step, the matrix is sorted row by row in increasing order based on the row length. Each range of rows having the same length is then further individually sorted based on the column position of the first non-zero element in each row, resulting in an ordered load of vector `X` leading to possible reuse. In the second step, the sorted matrix is split into two main groups of rows. We apply the ELL format to the group of shorter rows, for which the row lengths are not larger than 128. The empirical value of 128 is chosen so that the vectorized CSR format is more efficiently employed for the long matrix group. The ELL format is employed for each row length (from 0 to 128) at 32 row boundaries separately. In this way, the mechanism of the memory coalescing and SIMD operations in a warp are fully realized. The last ($nrow_i \mod 32$) rows of length i are simply merged into the next sub part with length $i + 1$, where $nrow_i$ is the number of rows with length of i. At most 31 zeros are needed for the remaining rows with the length i to merge into the subpart with the row length $i+1$. The vectorized CSR format is applied to the part with rows longer than 128. Each row is aligned at 32-element boundaries by adding at most 31 zeros per row. The EVC-HYB format then uses the mechanism of memory coalescing on a GPU for loading the sparse matrix data from the global memory to cache. As a result, the loading throughput increases significantly, which accelerates the computation. Figure 2 illustrates how the matrix is partitioned and stored with the ELL and vectorized CSR format, respectively.

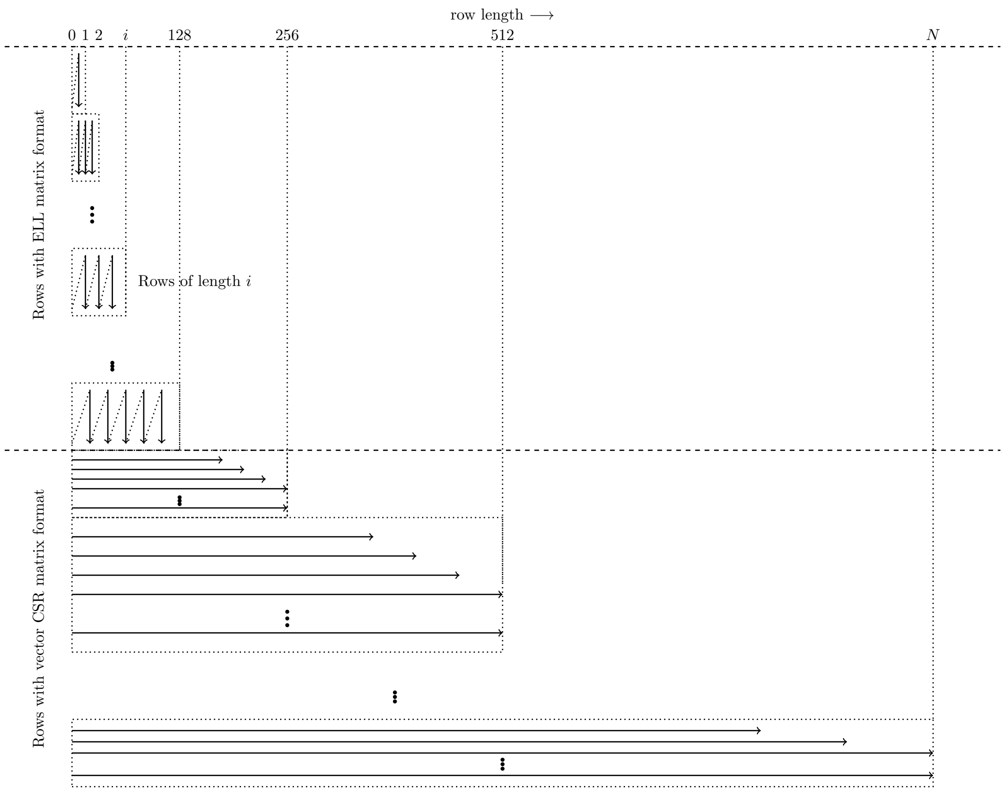
row length ⟶
0 1 2
i
128
256
512
N
Rows with ELL matrix format
Rows of length i
Rows with vector CSR matrix format

Figure 2: Illustrating the EVC-HYB format.

In order to further balance the workload for the threads in GPU blocks, GPU threads are adaptively distributed to both the ELL format and vector CSR format [13]. For the group of short rows with the ELL format, we first find the maximum row length, called $LR_{\max}$ (≤ 128), and distribute one row per thread for the length range $[LR_{\max}/2, LR_{\max}]$. For the rows whose lengths are in the range $[LR_{\max}/4, LR_{\max}/2]$, we allocate two rows per thread, and so on. For the group of long rows (including a few short rows merged from the short rows because of the 32-row alignment boundaries) using the vectorized CSR format, we directly apply adaptive thread distribution. Consequently, calculation of the total number of GPU blocks needed for the two row parts is straightforward when using this matrix partition and GPU thread distribution.

2.2 GPU Implementation

To implement this format on a GPU, we first define two data structures called `SubELL` and `SubCSR`, which are employed to store the sub matrices that use the ELL and CSR formats respectively. The C codes for the two structures are shown in Table 1. Besides the regular parameters and arrays needed for storage, we also introduce two arrays, `rmark` and `gbmark`, to mark row ranges and record related GPU blocks needed for the computation respectively.

The kernel `SpMV_EVCHYB_adaptive` listed in Algorithm 1 describes the basic idea used to compute the SpMV on a GPU using the EVC-HYB format. The parameter `EVC_BOUNDARY` determines the allocation of the GPU

SubELL Structure	
`nz`	total number of non-zeros
`bnz[]`	array of non-zeros per column block
`clen[]`	array of column lengths
`j[]`	column index array
`as[]`	data array
`np`	number of row blocks for adaptive block distributions distributions
`rmark[]`	row markers for different column lengths
`gbmark[]`	array for GPU block marks
`gpublks`	number of blocks
`LCmax`	maximum column length (≤ 128)
SubCSR Structure	
`n`	number of rows
`nz`	total number of non-zeros
`i[]`	row pointer
`j[]`	column indices
`as[]`	data array
`np`	number of row blocks for adaptive block distributions
`rmark[]`	row markers for different column lengths
`gbmark[]`	array for GPU block marks
`gpublks`	number of blocks

Table 1: Sub-matrix data structures for ELL and CSR formats.

blocks to the ELL and CSR computation device kernels respectively. GPU blocks with indices that are smaller than `EVC_BOUNDARY` are distributed in the SpMV computation for the ELL parts with the device function `SpMV_ELL_adaptive` (line 1). Other blocks are used in the device function `SpMV_CSR_adaptiveTPR` (line 2) for the CSR part.

Algorithm 1: `SpMV_EVCHYB_adaptive`

```
Input  : ell:            SubELL Struct
         csr:            SubCSR Struct
         X:              vector
         EVC_BOUNDARY:   block marker
         $\alpha$:              input scalar used for multiplication
         $\beta$:              input scalar used for multiplication
Result: Y ← $\beta$Y + $\alpha$Ax

if blockIdx.x < EVC_BOUNDARY then
    // ELL Part
1   SpMV_ELL_Adaptive ()
else
    // CSR Part with adaptive TPR
2   SpMV_CSR_adaptiveTPR ()
end
```

The device function `SpMV_ELL_adaptive` is used to compute the SpMV for the short rows stored in the adaptive ELL format, which is shown in Listing 1. The array `ell.rmark` stores the matrix row indices where the new row length begins, and `ell.gbmark` records the information about the adaptive GPU blocks distributed for rows with different lengths. Based on the information the two arrays provide and the GPU block ID, each GPU block searches the row block range number it should apply for (lines 12–22), and each

GPU thread can easily determine which matrix rows it should handle (lines 24–27). The GPU block dimension is required to be larger than the total number of row partitions stored in the ELL format (`ell.np`, which is 128 in the test), so that every GPU block has enough threads to perform the search process in parallel. Every thread then calculates the results for those rows (lines 29 – 39). The device function `SpMV_CSR_adaptiveTPR` shown in Listing 2 uses a similar strategy for the adaptive computation for the long matrix rows stored in the CSR format. The code in lines 15 – 24 searches the related row block range number and compute GPU thread numbers (TPR) applied to each row, while the code in lines 26–34 calculates the parameters needed by the local thread. And finally each thread do their work and store the reduced sum results to the related row. The GPU-specific `const __restrict__` keywords are used for the `X` array, in order to use the 48 KB read-only cache for load and reuse.

```
 1 __device__ void SpMV_ELL_adaptive
 2 (SubELL ell, const double* __restrict__ x, double alpha, double *y, double beta )
 3 {
 4   const int tid  = threadIdx.x;
 5   const int bid  = blockIdx.x;
 6   const int bDim = blockDim.x;
 7   __shared__ int  ib, ivstart, vlen, NROWB, rstride;
 8
 9   int *gbmark = ell.gbmark;
10   int *rmark   = ell.rmark;
11
12   if ( tid < ell.np) {
13     if ( bid >= gbmark[tid] && bid < gbmark[tid+1] )
14     {
15       ib      = tid;
16       ivstart = ell.bnz[ib];
17       NROWB   = rmark[ib+1] - rmark[ib];
18       rstride = bDim * (gbmark[ib+1] - gbmark[ib]);
19       vlen    = ell.clen[ib] * NROWB;
20     }
21   }
22   __syncthreads();
23
24   double *yb    = y    + rmark[ib];
25   double *vb    = ell.as + ivstart;
26   int    *idxb = ell.j  + ivstart;
27   int row_id    = bDim * ( bid - gbmark[ib] ) + tid;
28
29   int    *idx1;
30   double *v1;
31   for ( int row = row_id; row < NROWB; row += rstride ) {
32     v1   = vb   + row;
33     idx1 = idxb + row;
34     double sum1 = 0.0;
35     for (int j = 0; j < vlen; j += NROWB) {
36       sum1 += v1[j] * x[idx1[j]];
37     }
38     yb[row] = beta * yb[row] + alpha * sum1;
39   }
40 }
```

Listing 1: The device kernel `SpMV_ELL_adaptive` used for the SpMV for the ELL sub-matrix

```
 1 __device__ void SpMV_CSR_adaptiveTPR
 2 (SubCSR csr, const double* __restrict__ x, double alpha, double *y, double beta, int EC_BOUNDARY)
 3 {
 4      __shared__  double sd1[GPU_BLOCKSIZE];
 5
 6     const int tid  = threadIdx.x;
 7     const int bid  = blockIdx.x - EC_BOUNDARY;
 8     const int bDim = blockDim.x;
 9
10     __shared__ int  ib, TPR, NROWB, rstride;
11
12     int *gbmark = csr.gbmark;
13     int *rmark  = csr.rmark;
14
15     if (tid < csr.np ) {
16       if (bid >= gbmark[tid] && bid < gbmark[tid+1])
17       {
18          ib  = tid;
19          TPR = GPU_BLOCKSIZE / pow2( csr.np - 1 - ib );
20          NROWB   = rmark[ib+1] - rmark[ib];
21          rstride = bDim * (gbmark[ib+1] - gbmark[ib]) / TPR;
```

```
      }
    }
    __syncthreads();

    int      irc      = rmark[ib];
    int      ivstart = csr.i[ irc ] - csr.i[ 0 ];
    int     *iib      = csr.i    + irc;
    int     *idxb     = csr.j    + ivstart;
    double *vb       = csr.as   + ivstart;
    double  *yb      = y       + irc;

    int row_id      = ( bDim * ( bid - gbmark[ib] ) + tid ) / TPR;
    int thread_lane = tid % TPR;

    for(int row = row_id; row < NROWB; row += rstride) {
        int n   = iib[row] - iib[0];
        double *v0    = vb +  n;
        int    *idxp = idxb + n;
        double sum = 0.0;
        for (int j = thread_lane; j < iib[row+1] - iib[row]; j += TPR) {
           sum += v0[j]*x[idxp[j]];
        }

        sd1[tid] = sum;
        __syncthreads();

        int i = TPR >> 1;
        while(i > 0) {
           if(thread_lane < i) sd1[tid] += sd1[tid + i];
           __syncthreads();
           i >>= 1;
        }

        if (thread_lane == 0) yb[row] = beta * yb[row] + alpha * sd1[tid];
    }
}
```

Listing 2: The device kernel `SpMV_CSR_adaptiveTPR` used for the SpMV for the CSR sub-matrix

3 Test Results and Analysis

We performed tests on the XK7 nodes of the *Blue Waters* system, which is installed at the National Center for Supercomputing Applications (NCSA), University of Illinois at Urbana-Champaign [8]. All test codes are compiled with `nvcc --O3 --arch=sm_35` with ECC memory protection enabled. The XK7 compute node combines an AMD 16-core Opteron 6200 Series processor (Interlagos) and an NVIDIA Tesla K20X GPU (Kepler) Accelerator. Each Kepler GPU accelerator [19] includes 14 streaming multiprocessors (SMX), and each SMX has 192 single precision CUDA cores and 64 double precision units. The clock rate is 732 MHz, giving a peak performance of 1.31 ($= 2 * 732 * 64 * 14/10^6$) TFLOP/s for double precision floating-point operations, while the total global memory is 6 GB per GPU. The STREAM benchmark for GPU [23] yields a bandwidth of around 180 GB/s for double precision arrays with size of 2×10^7, which is shown in Table 2.

Function	Rate (GB/s)
Copy	182.75
Scale	182.75
Add	180.93
Triad	180.93

Table 2: `STREAM` benchmark on the Blue Waters Kepler GPU.

Test matrices are selected from The University of Florida Sparse Matrix Collection [11]. The matrices are drawn from applications that include computational fluid/solid mechanics, bioengineering, economics, circuit simulation and web search. Table 3 lists their dimensions (`NROW` x `NCOL`), the total number of the non-zero elements in the matrices, amount of the non-zero elements and the percentages for the ELL and vector CSR (vCSR) formats respectively, and the percentage of zeros that are needed for the EVC-HYB format. The column $\mathtt{I_{split}}$ shows the row number where the sorted matrix is split. Twelve matrices are dominated by the adaptive ELL format — i.e., where $\mathtt{NNZ_{ELL}}$ is larger than 99.0%, while one matrix is dominated by the vCSR format — i.e. $\mathtt{NNZ_{vCSR}}$ is larger than 99.0%. The number of zeros added for padding range from 0.00% (for the `ldoor` matrix) to 6.86% (for the `pdb1HYS` matrix) of the number of non-zero elements in the matrices, and for most of the matrices, it is less than 1.0%. An empirically selected GPU blocksize of 256 is used in the test. All floating point numbers are computed in double precision.

Name	NROW (NCOL)	NNZ	$\mathtt{N_{nzr}}$	$\mathtt{I_{split}}$	$\mathtt{NNZ_{ELL}}$	% $\mathtt{NNZ_{ELL}}$	$\mathtt{NNZ_{vCSR}}$	% $\mathtt{NNZ_{vCSR}}$	% NNZ added	CUSPARSE
pdb1HYS	36417	4344765	119.3	19936	1922112	41.40	2720768	58.60	6.86	CSR
pwtk	217918	11634424	53.4	217888	11633280	99.97	3456	0.03	0.02	HYB
TSOPF_RS_b2383	38120	16171169	424.2	21728	98976	0.61	16220544	99.39	0.92	HYB
rma10-cfd	46835	2374001	50.7	46752	2365664	99.45	13088	0.55	0.20	CSR
audikw_1	943695	77651847	82.3	844096	58302912	73.16	21392800	26.84	2.63	CSR
cage14	1505785	27130349	18.0	1505728	27128544	99.99	3648	0.01	0.01	HYB
shipsec1	140874	7813404	55.5	140864	7813824	99.98	1280	0.02	0.02	HYB
webbase-1M	1000005	3105536	3.1	999232	2768192	88.69	353120	11.31	0.51	HYB
raefsky3	21200	1488768	70.2	21184	1488384	99.90	1536	0.10	0.08	HYB
cop20k_A	121192	2624331	21.7	121184	2624960	99.97	768	0.03	0.05	HYB
msdoor	415863	20240935	48.7	415840	20239808	99.99	2208	0.01	0.01	HYB
ldoor	952203	46522475	48.9	952192	46522368	100.00	1056	0.00	0.00	HYB
Circuit	170998	958936	5.6	170944	954272	99.20	7744	0.80	0.32	HYB
torso1	116158	8516500	73.3	113376	1020384	11.97	7501152	88.03	0.06	CSR
Stanford	281903	2312497	8.2	281728	2283392	98.49	35040	1.51	0.26	HYB
att-pre2	659033	5959282	9.0	658240	5476288	91.72	494176	8.28	0.19	HYB
Economics	206500	1273389	6.2	206496	1273952	99.98	256	0.02	0.06	HYB
ibm-dc1	116835	766396	6.6	116768	595776	77.58	172160	22.42	0.20	HYB
indochina-2004	7414866	194109311	26.2	7345728	126110176	64.58	69171328	35.42	0.60	CSR
web-Google	916428	5105039	5.6	916352	5093728	99.69	15808	0.31	0.09	HYB
Stanford_Berkeley	683446	7583376	11.1	678400	3638208	47.45	4028832	52.55	1.10	HYB
RM07R	381689	37464962	98.2	311776	24249408	63.74	13792384	36.26	1.54	CSR

Table 3: Matrix statistics.

3.1 Performance comparison of the EVC-HYB, CSR, and HYB formats

The two popular formats, CSR and HYB, are chosen for comparison with the performance of our EVC-HYB format. As a baseline we employ the NVIDIA cuSPARSE package [20] for these implementations. The last column in Table 3 lists the CSR or HYB format that result in the best performances with the cuSPARSE package. Six matrices perform best with the CSR format, while most perform best with the HYB format.

Figure 3 shows the performance speedups of EVC-HYB format compared to the best results using the CSR or HYB formats. Because our format permutes the matrix rows, it is necessary to reorder the results after the matrix-vector multiply. The time for this operation is included in the results labeled “with Reordering.” However, in an interative algorithm that applies the same matrix at each iteration, it is not necessary to perform the reordering until the end of the iteration. To better show the performance in this case, we include the performance where there is no reordering in the results labeled “without Reordering.” The performance of the EVC-HYB format is measured based on the average of 500 iterations, including the tuning process for different adaptive parameters to balance the thread workload. Without reordering, the EVC-HYB format yields performance improvement for ten matrices, where the speedup ranges from 1.10 (for the `msdoor` matrix) to 1.64 (for the `Stanford_Berkeley` matrix), while it results in similar performance ($[0.90, 1.10]$) for other matrices, and worse performance (0.90) for one matrix (`pwtk`). If reordering is applied in each iteration, the performance of the EVC-HYB format degrades as expected. The EVC-HYB format yields similar performance for many matrices, while it still achieves good speedup for some matrices, such as `torso1` (1.14), and `indochina-2004` (1.43).

Figure 4 describes the relative performance degradation along with the average NNZ elements per row (`NNZ/NROW`), due to the cost of reordering the `Y` array. The degradation ranges from 3.52 to 34.15%. For

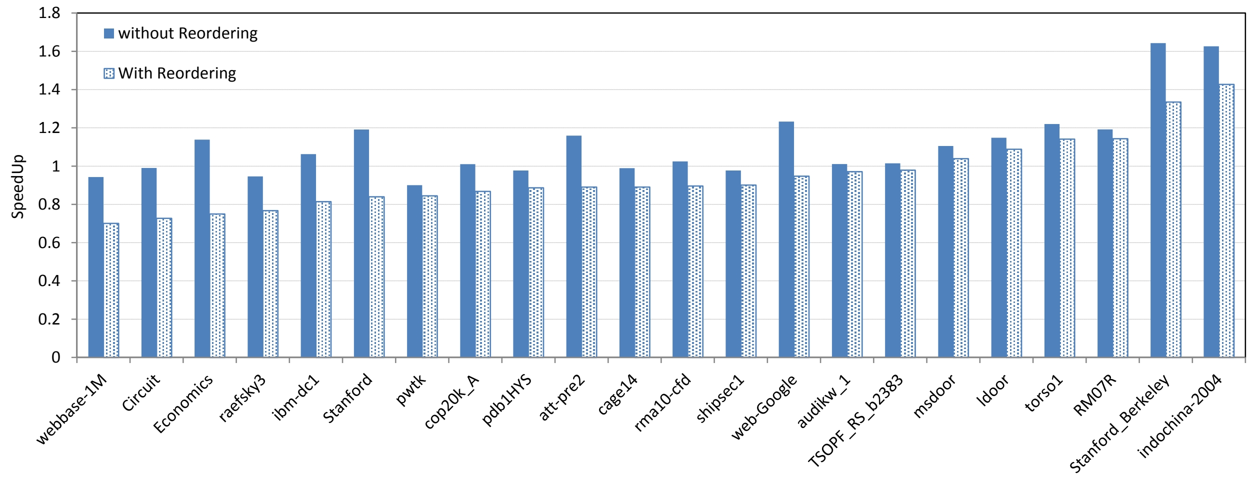

Figure 3: Speedup of the EVC-HYB format compared to the best results using the CSR or HYB formats in the cuSPARSE package.

those matrices that have less than 12 elements per row on the average, the degradation is from 18.75 to 34.15%. For those with `NNZ/NROW` between 18 and 119, the degradation is from 3.52 to 18.90%. And for the `TSOF_RS_b2383` matrix with `NNZ/NROW` = 424, the degradation is only 3.52%. This indicates that the relative complexity of reordering `Y` is decreased when the number of nonzero elements per row is increased.

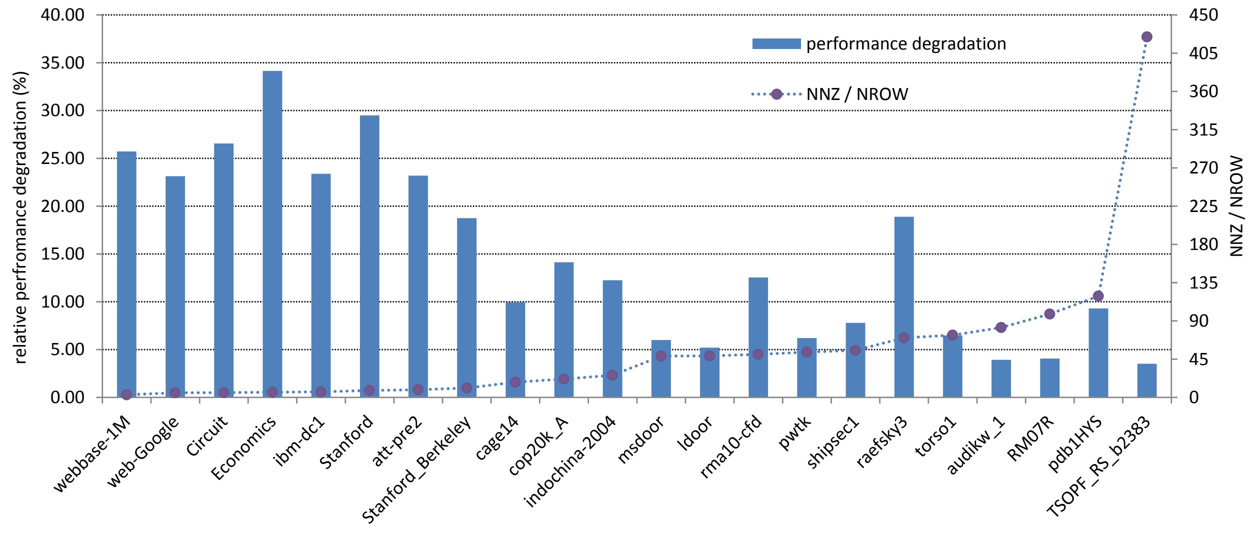

Figure 4: Relative performance degradation due to the reordering of the `Y` array and average `NNZ` elements per row (`NNZ/NROW`).

Figure 5 shows the FLOP rates for the tested matrices. The FLOP rate is computed ignoring the zeros added for padding. The best performance is around 24.7 GFLOP/s for all the formats. The EVC-HYB format achieves similar or better performance for most of the tested matrices, compared to the CSR and HYB formats, although we observe some performance degradation due to the reordering of the `Y` array.

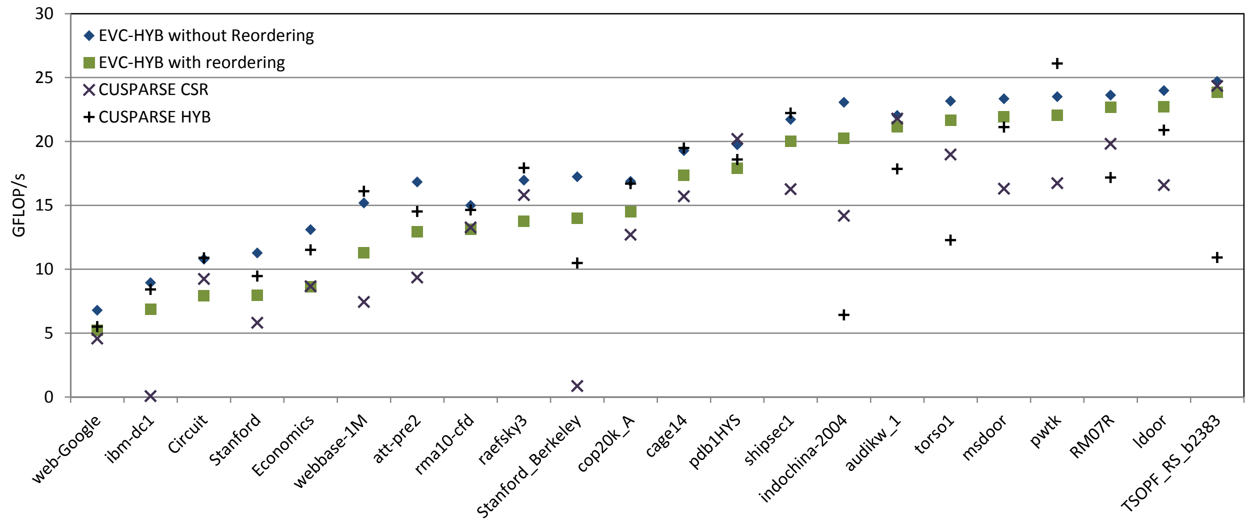

Figure 5: FLOP rates for the EVC-HYB, CSR, and HYB formats.

3.2 Throughput of the hierarchical memory and Performance model for the EVC-HYB format

To consider EVC-HYB in more depth, we measure the data throughput of the hierarchical memory using the `nvprof` profiling tool the hierarchical memory for the kernel `SpMV_ECHYB_adaptive` on the Kepler GPU, which is shown in Figure 6. The reordering process for the `Y` array is not included here. The `dram_read` data series represents reading throughput from the GPU device memory to the L2 cache and it varies from 119.08 (for `Circuit`) to 206.79 GB/s (for `audikw_1`). `L2_read` shows the throughput seen at the L2 cache for all read requests, and it ranges from 125.56 GB/s (for `web-Google`) to 275.94 GB/s (for `cop20k_A`). `L2_tex_read` is a part of the L2 cache throughput, which is only for read requests from the texture cache for the RHS `X` array. Its value varies from 6.04 (for `raesky3`) to 160.29 GB/s (for `cop20k_A`). `tex_cache` is for the texture cache throughput, which varies from 22.03 GB/s (for `web-Google`) to 103.42 GB/s (for `TSOPF_RS_b2383`).

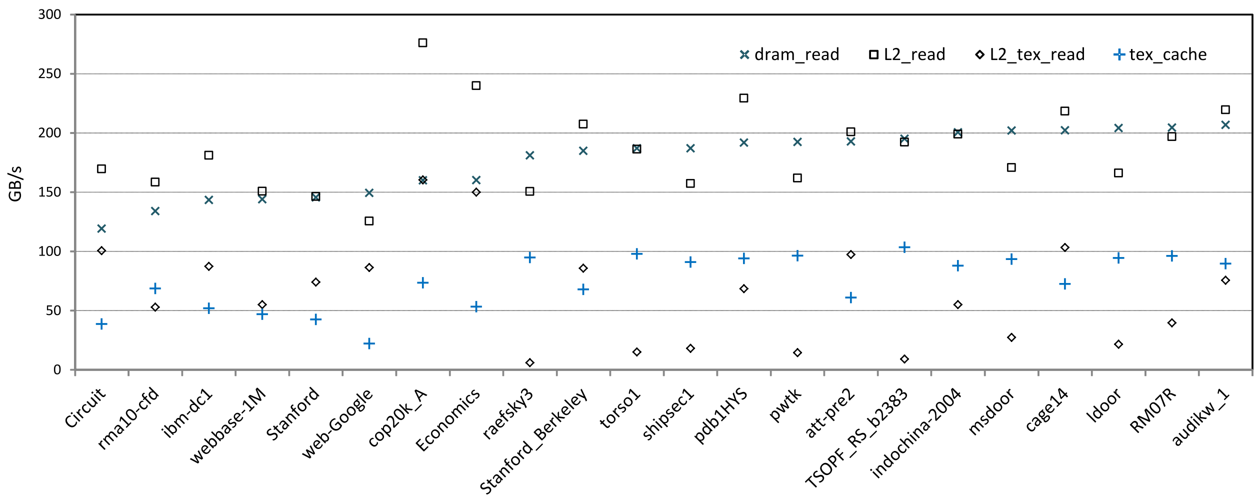

Figure 6: Throughput of device memory, L2 cache, and texture cache.

There are four metrics in `nvprof` that measure the cache hit rates and are related to the data loading for the SpMV operation: `l2_texture_read_hit_rate` (h_{L2t}) and `tex_cache_hit_rate` (h_t) are for loading of the

`X` array, and `l1_cache_global_hit_rate` (h_{L1}) and `l2_l1_read_hit_rate` (h_{L2_L1}) are for data loading of matrix and other accessory arrays. h_{L1} is equal to zero for all matrices tested. Figure 7 shows the values of the other three cache hit rates. h_{L2_L1} is low, from 0.83 to 15.21%, which indicates that most of the matrix and the related accessory data are loaded from the DRAM, as expected. h_{L2t} and h_t vary from 39.33 to 94.45% and 1.95 to 95.31% respectively. The lower values of h_{L2t} and h_t indicates the lower chance of the reuse of the `X` array data in the caches.

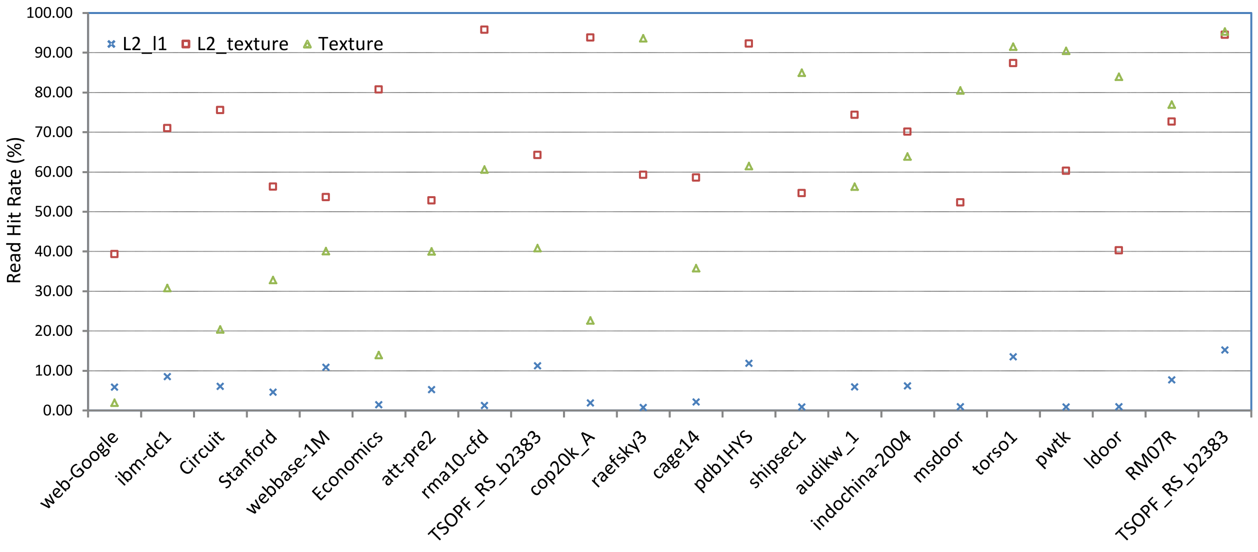

Figure 7: Cache hit rates.

Using these hit rates, we further estimate the average latencies for loading the `X` array (C_X) and the matrix (C_A) respectively, which are defined as the average number of cycles needed for loading matrix and `X` data across the GPU hierarchal memory:

$$C_X = C_t h_t + C_{L2t} h_{L2t}(1 - h_t) + C_d(1 - h_{L2t})(1 - h_t),$$
$$C_A = C_{L1} h_{L1} + C_{L2\_L1} h_{L2\_L1}(1 - h_{L1}) + C_d(1 - h_{L2\_L1})(1 - h_{L1}),$$

where $C_t \approx 110$, $C_{L2t} \approx 220$, $C_{L2\_L1} \approx 230$, and $C_d \approx 600$ denote the number of clock cycles needed to load from texture, L2 cache for texture cache, L2 cache for L1 cache, and DRAM memory, respectively [17, 18]. Figure 8 shows the relationship between performance and the average latencies C_A and C_X. The values of C_A are similar at around 580 cycles, while C_X varies in a large range for the different matrices. The higher value of C_X usually results in the lower performance for the tested matrices. The `web-Google` matrix needs many more cycles (444) for each load, compared to that for other matrices. As expected, its performance is the lowest (5.34 GFLOP/s). On the other hand, the `TSOPF_RS_b2383` matrix achieves the best performance (24.61 GFLOP/s), and it only needs 116 clock cycles on the average for each `X` load request. There are also some exceptions. The C_X value for the `rma10-cfd` matrix is smaller, but its SpMV performance is lower, compared to many other matrices.

The EVC-HYB format achieves much higher performance for the `Standford_Berkeley` matrix, compared to the HYB format. The throughput comparison of hierarchical memory for the two formats, shown in Figure 9, is revealing. The `ellmv_val` and `coo_fastpass_val` are two main device kernels used in the HYB format, which are shown in the output of the metrics from `nvprof` profiling tool. As we can see, the EVC-HYB format outperforms HYB format in terms of throughput at each memory level.

As we mention, the texture cache is utilized to load the RHS `X` array in the tests. We further test the performance of the `web-Google` matrix without using the texture cache. Table 4 lists the FLOP rate and some metrics for the tests with and without the texture cache respectively. When the texture cache is turned off, the throughputs and hit rates related to the texture cache become zero, while that related to the L1 cache

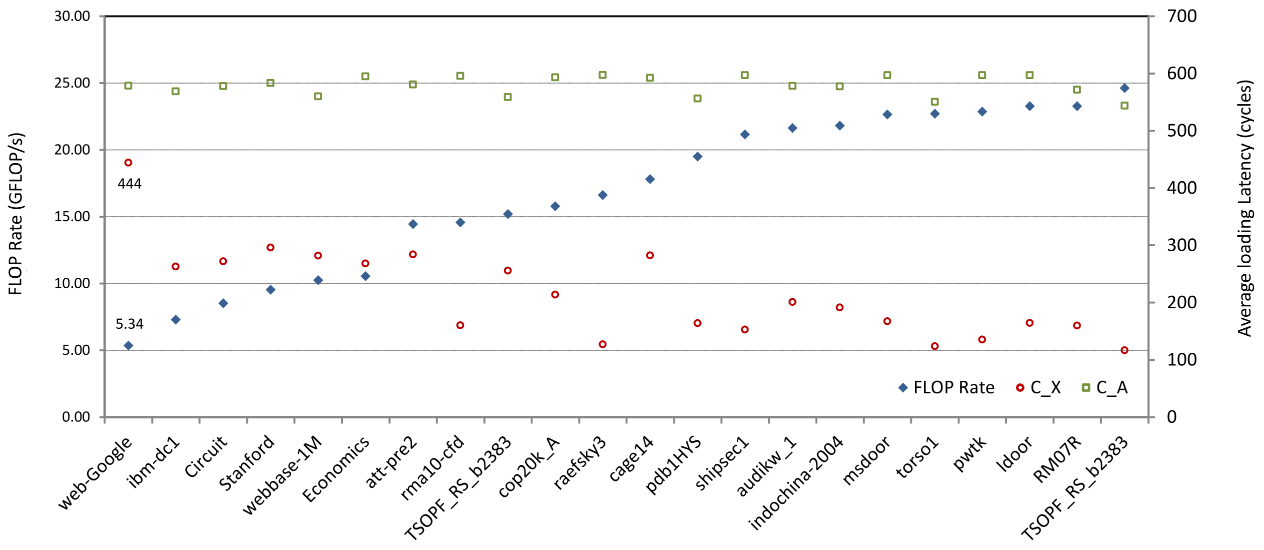

Figure 8: FLOP rate and average latency.

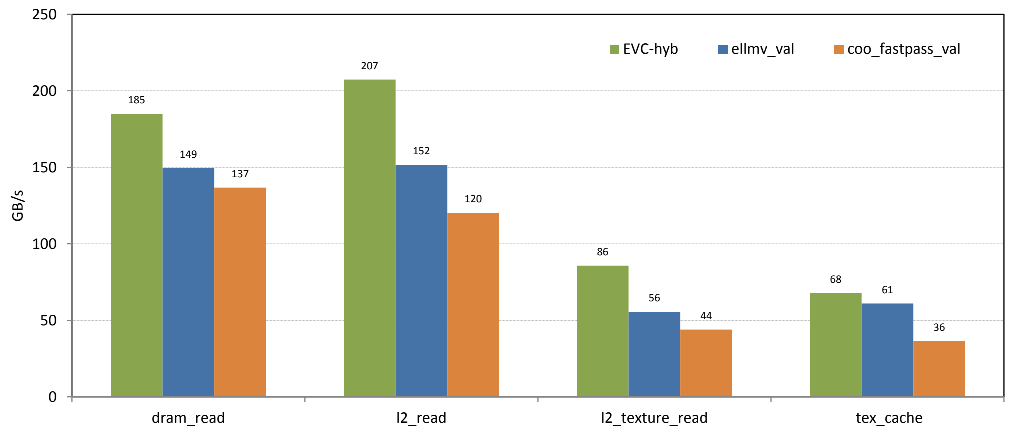

Figure 9: Throughput of hierarchical memory for the `Stanford_Berkeley` matrix.

significantly increases. The DRAM and L2 read throughputs decrease. As a result, the FLOP rate decreases from 5.34 GFLOP/s to 4.89 GFLOP/s. It might be because all of the data needs to be read through the L1 cache, and this slows down the general throughput of the DRAM memory and the small L2 cache. The test data indicates that the texture cache could provide an extra channel for data loading and therefore improve the performance, even if its hit rate is low.

A model to evaluate performance of the SpMV in the hybrid format is based on the *sliced* ELL format [14], using concepts of Roofline modeling [25]. The model is suitable for our EVC-HYB format. The code balance B measures the amount of data (bytes) needed to be loaded from the hierarchical memory for each floating point operation on the average. Using 8-byte double precision floating numbers for nonzero elements and 4-byte integers for their indices, the total bytes needed for the computation is

$$\text{TBytes} = \left(\frac{1}{\beta}(8+4)+8\alpha\right) NNZ + 16N,$$

the total flops is

$$\text{TFlops} = 2NNZ + 3N,$$

metric	texture	no_texture
FLOP Rate (GFLOP/s)	5.34	4.89
dram_read_throughput (GB/s)	146.80	134.83
l2_read_throughput (GB/s)	124.15	114.81
l2_l1_read_throughput (GB/s)	36.10	114.80
l2_texture_read_throughput (GB/s)	88.04	0.00
l2_l1_read_hit_rate (%)	6.48	29.87
l2_texture_read_hit_rate (%)	39.64	0.00
tex_cache_hit_rate (%)	2.00	0.00

Table 4: The test comparison with and without the texture cache for the `web-Google` matrix.

and therefore the code balance B is defined as

$$B = \frac{\text{TBytes}}{\text{TFlops}} = \left(\frac{1}{\beta} \cdot \frac{8+4}{2+3/N_{nzr}} + \frac{8\alpha + 16/N_{nzr}}{2+3/N_{nzr}} \right) \text{bytes/flop},$$

where β is the ratio of the nonzero elements used in the CSR format and the elements (inlcuding zero paddings) in the test format, for example EVC-HYB. The parameter α estimates how the `X` array is reused in the computation. Theoretically, a value of α close to zero means the `X` array is a resident of the registers, while $\alpha \geq 1$ indicates many elements of the `X` array need to be loaded from the GPU hierarchical memory, such as the texture cache, the L2 cache or DRAM memory. The term N_{nzr} is the average number of nonzero elements per row so that $16/N_{nzr}$ represents the average bytes from `Y` used in each floating point operation. Since the additional index data used for adaptive computing is relatively small in our kernel, we ignore their impact in the model.

We measure the data volume V_{meas} using `nvprof`, which estimates the total bytes. Therefore, the parameter α is calculated with the following formula:

$$\alpha = \frac{1}{4} \left(\frac{V_{meas}}{2NNZ} - \frac{6}{\beta} - \frac{8}{N_{nzr}} \right),$$

and then the code balance B can be calculated. The values of the parameter α and the code balance B for the tested matrices are shown in Figure 10. For all the matrices except `web-Google`, the code balance is between 7.45 to 11.52, while α ranges from 0.37 to 1.66. The value of α is notably high for the `web-Google` matrix (≈ 4.93), which indicates that the matrix is highly irregular resulting in `X` loads mainly from global memory at every use. A read-only transaction loads 32 bytes through the memory hierarchy on the Kepler GPU. As a result, loading an isolated 8-byte double precision number requires reading 24 additional and unused bytes. This has a noticeable impact on loading in the SpMV and explains why the parameter α and the related code balance B (21.4) for the `web-Google` matrix are much higher than that for other matrices.

In conclusion, Figure 11 shows the measured performance P_{meas} and the estimated maximum achievable performance P_{max} based on the above performance model. Here, P_{max} is defined b/B, where b is the achievable memory bandwidth and is measured using the `COPY` kernel in the `STREAM` benchmark program in Table 2. The EVC-HYB format achieves a performance similar to the roofline model prediction for most of the tested matrices. Note that our P_{max} is not a rigorous bound on performance; rather, it is an estimate based on mandatory data motion and the memory bandwidth measured by the `STREAM` package. Thus, our P_{meas} is sometimes higher than P_{max}.

3.3 Qualitative analysis of time profiling for SpMV

Sparse matrix operations require high and sustained memory bandwidth, and the performance of these operations can be estimated from measurements of the sustained memory bandwidth [1] when these operations are programmed efficiently. The time for a SpMV is spent largely on the movement of the three parts of

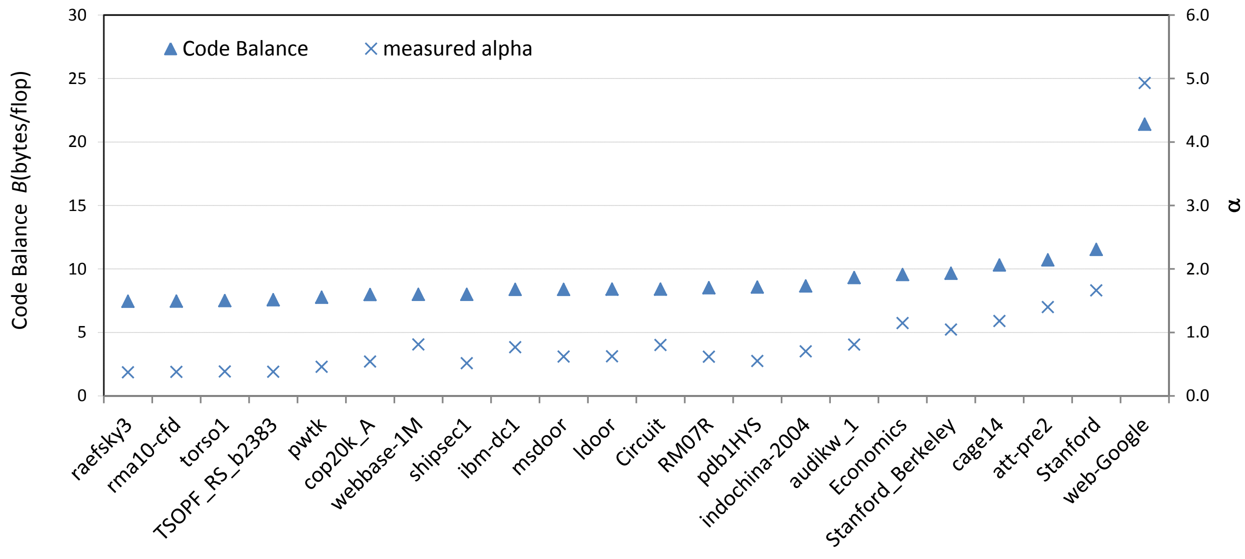

Figure 10: The code balance and the associated α values.

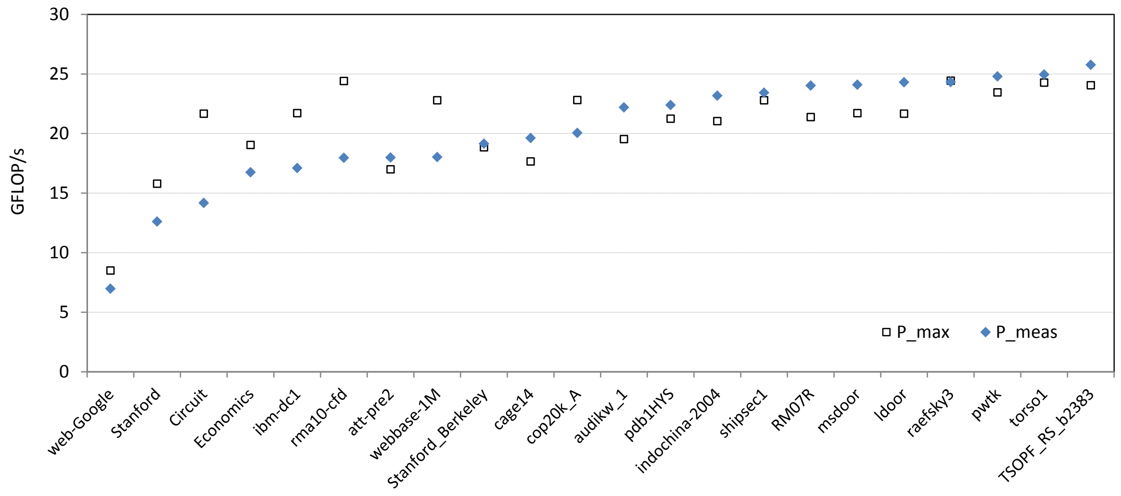

Figure 11: The measured and modeled maximum achievable performance for the EVC-HYB SpMV.

data. The first part, which we denote with time $T_{\mathtt{A}}$, is the non-zero and zero-padding elements of the sparse matrix, which are defined as double precision floating point (8-byte) numbers in this paper. The second part is X and its index array, which is labeled by time $T_{\mathtt{x}}$. The total data due to repeatedly loading X is the same as that for matrix A when the ELL or CSR format is employed. The index array reflects the sparsity pattern of the matrix. Elements of the index array are typically defined as 4-byte integers, and the array's data amount is half of that for the non-zero elements in 8-byte double precision floating numbers for the ELL and CSR formats. The texture cache on a GPU is used to read the X array in order to accelerate the loading throughput. However, the texture cache is only a 48KB read-only data cache accessible by the Texture unit on a Kepler GPU, which is limiting when X becomes large. In this case, the elements of X are repeatedly reloaded from the global memory. Therefore, the performance of SpMV is also significantly affected by the loading throughput of the vector X. Finally, the third part, denoted by time $T_{\mathtt{Y}}$, is the loading and storing of the array Y and some other auxiliary arrays used to help allocate the GPU blocks. The total time cost can be approximated as $T_{\text{total}} = T_{\mathtt{A}} + T_{\mathtt{X}} + T_{\mathtt{Y}}$.

We designed three special tests of SpMV with the EVC-HYB format (without reordering Y) to estimate these three stages of execution. In the first experiment, we set `X` as constant, which means `X` and the associated index array do not need to be loaded. The measured time is regarded as $T_{\mathtt{A}} + T_{\mathtt{Y}}$. In the second experiment, we set `A` as constant, thus avoiding the Load of the matrix data. The measured time is then regarded as $T_{\mathtt{X}} + T_{\mathtt{Y}}$. Finally, `A` and `X` are both set as constant in the third experiment, leading to a measured time of $T_{\mathtt{Y}}$. Figure 12 shows the time ratios $T_{\text{total}}/T_{\text{SpMV}}$ for the tested matrices, where T_{SpMV} is the time for the complete SpMV. The ratio is less than 1.15 for most of the tested matrices, and only four matrices result in the ratios of 1.17, 1.17, 1.19 and 1.21 respectively. This highlights the accuracy of our measurement method and provides a qualitative estimate of the three time components. In practice, there is likely some overlap between these three parts of data movement. The matrix `A` and left-hand side `Y` utilize the L1 cache to load/store, while the right-hand side `X` array is loaded into the read only cache. This may reduce the total time as compared to the synthetic total time.

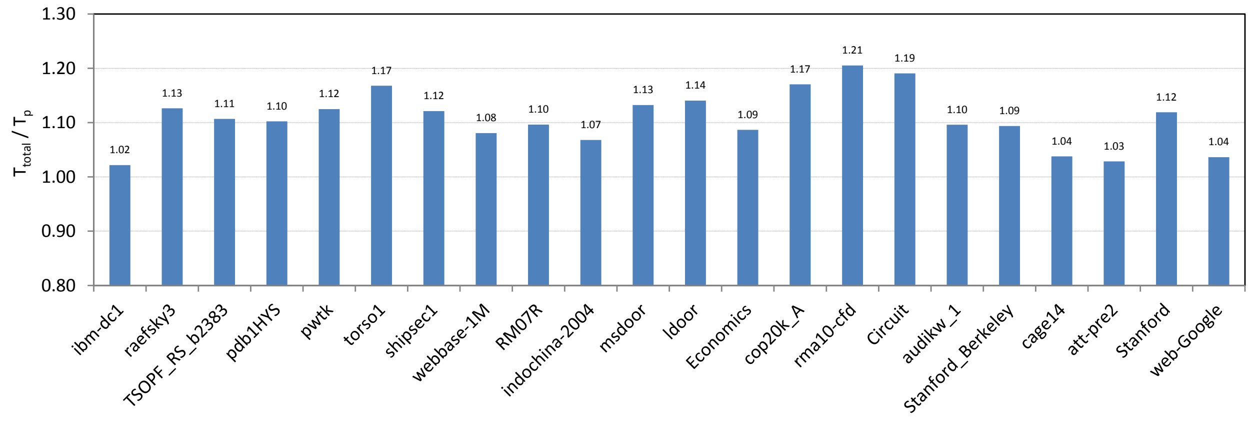

Figure 12: Time ratios of the synthetic total time for the complete SpMV operation.

Figure 13 shows the normalized times $T_{\mathtt{x}}/T_{\mathtt{total}}$, $T_{\mathtt{A}}/T_{\mathtt{total}}$, and $T_{\mathtt{Y}}/T_{\mathtt{total}}$. Generally, $T_{\mathtt{A}}$, $T_{\mathtt{x}}$ and $T_{\mathtt{Y}}$ occupy between 14%–49%, 34%–80%, and 7%–35% respectively, for the tested matrices. For the `webase-1M` matrix, the time for $T_{\mathtt{Y}}$ is 20% of the total time spent on the kernel, and all three parts are important. For the `Stanford` and `web-Google` matrix, time $T_{\mathtt{x}}$ represents 60% and 80% of the total time respectively, which significantly dominates the whole computation and contributes to the low FLOP rates for these two matrices even with the EVC-HYB format (9.53 and 5.34 GFLOP/s respectively).

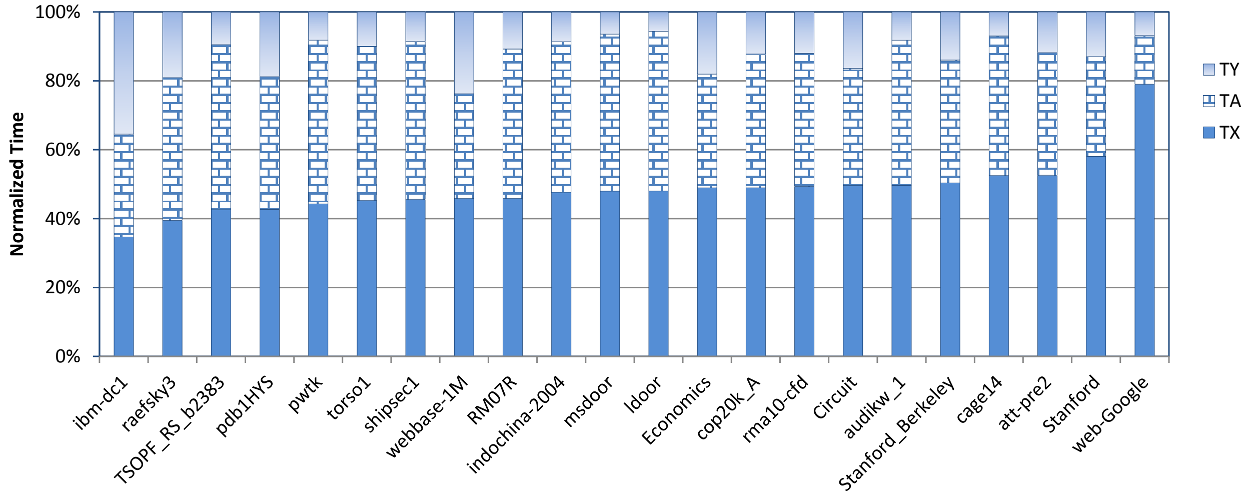

Figure 13: The measured normalized time for $T_{\mathtt{A}}$, $T_{\mathtt{x}}$, and $T_{\mathtt{Y}}$.

The FLOP rates, shown in Figure 14, also illustrate the problem: the performance of the complete SpMV and SpMV when `A` is constant are similar for the `web-Google` matrix, while it is significantly better when `X` is constant. The figures indicate that the repeat loading of the vector `X` based on the sparity pattern of a matrix also significantly affects performance, and should be optimized to further improve the performance of SpMV.

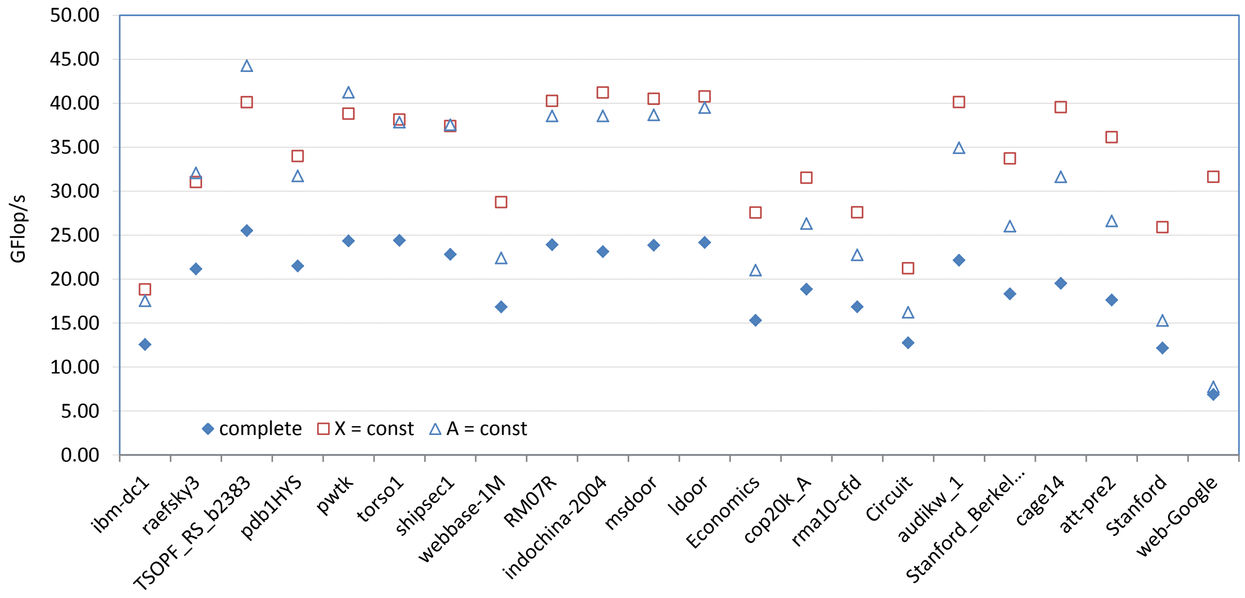

Figure 14: The FLOP rates of SpMV for three cases: complete, constant `X`, and constant `A`.

3.4 Application of the EVC-HYB format to synthetic matrices and PETSc

In this section we consider a class of random, synthetic matrices to test to test and compare the performance of our EVC-HYB, CSR, and HYB formats. In this test, we randomly generate nonzero elements in each row, ranging from 1 nonzero to $0.05N$ nonzeros (or 5%) per row. Figure 15 shows details of the matrix along with performance. The matrix dimension varies from 5000 to 65,000, while the total number of the NNZ elements for a matrix varies from 0.61 to 183.54 million, which is 1% to 5% of the dense matrix with the same dimension. The CUSPARSE CSR and HYB formats consistently yield between 20.0 and 13.0 GFLOP/s respectively when the number of non-zero elements in a matrix is larger than 15.0 million. The EVC-HYB format achieves around 19.0 GFLOP/s, which is similar as performance with the CSR format.

Finally, we apply our EVC-HYB format to a problem in PETSc [3] which includes GPU vector and matrix types to accelerate the computation, such as `vecusp` for the vector and `aijcusp` for matrix computation. We also implement an instance of the EVC-HYB format in PETSc for the computation of $\mathtt{Y} = \mathtt{AX}$.

The Bratu problem (solid fuel ignition) is considered in a 3D rectangular domain (PETSc `SNES` example 14) with a dimension of $200 \times 200 \times 200$. Four MPI tasks are employed in the computation and each MPI task is on an individual XK node, since each XK node has only one GPU. Two matrix types are tested. One is our EVC-HYB format (`mpievchyb`) and the other is `mpiaijcusp`, which uses the CUSP library [10] and with the PETSc vectorized CSR format. Figure 16 shows the performance comparisons of the SpMV and a general `SNESSolve` between these two matrix types. The `SNESSolve` represents the entire computational process to solve the nonlinear system, which includes the SpMV operation. The performance gains of the SpMV with the EVC-HYB format is notable, yet the total throughput of the nonlinear solve is limited by other operations.

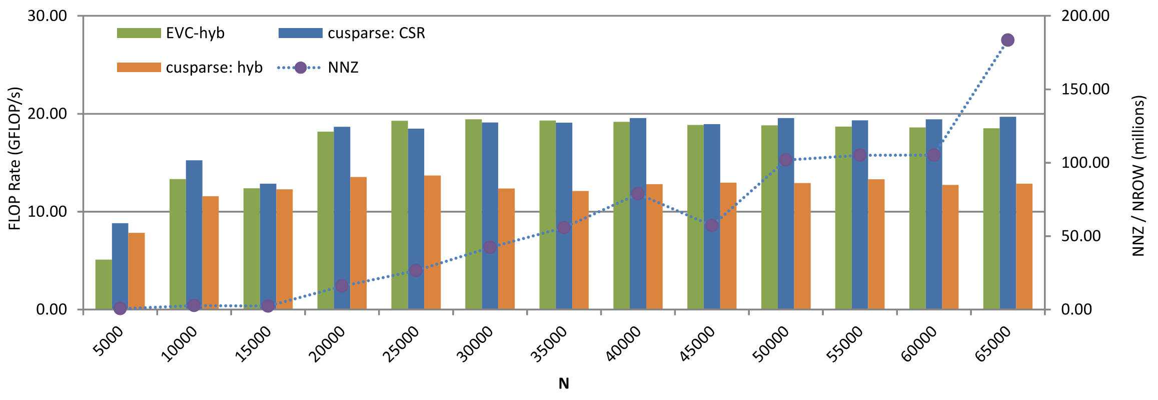

Figure 15: Performance comparisons for random matrices.

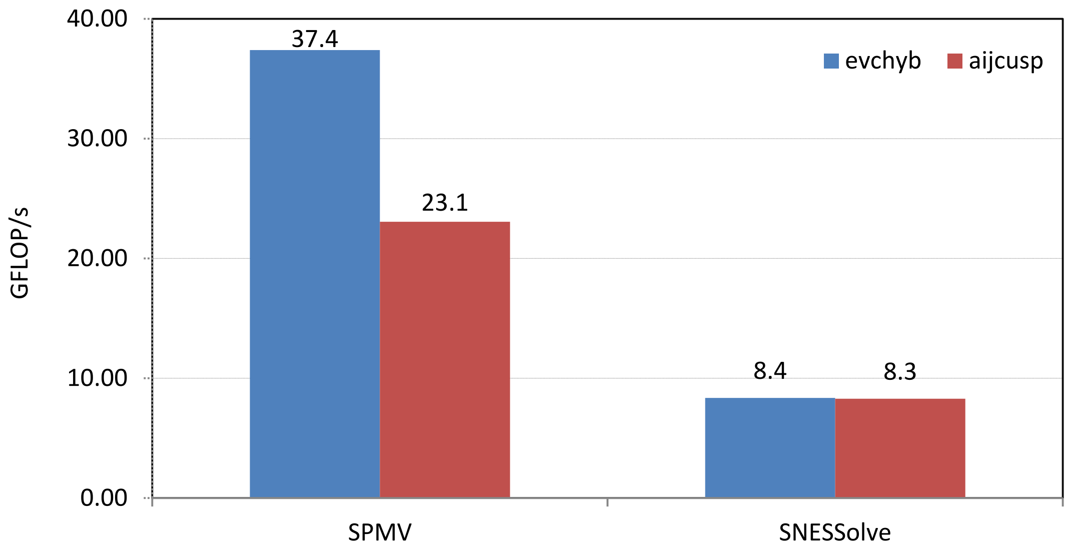

Figure 16: Performance comparison of the SpMV and general `SNESSolve` using `aijcusp` and `evchyb` for the Bratu problem. Four XK nodes were used.

4 Summary and Conclusion

In this paper we present a new EVC-HYB format that employs the ELL format for short rows and the vectorized CSR format for long rows, respectively. Adaptivly selected numbers of threads are distributed to different rows based on their lengths and further balance the workload for threads to reduce or eliminate idle threads during the computation. This hybrid format is able to better utilize the mechanism of memory coalescing, significantly increase data access throughput, and generally accelerate the computation of SpMV for a large range of matrices on a GPU accelerator, although it is not expected to uniformly yield optimal performance. We analyzed the throughput of hierarchical memory and the impact on SpMV performance for our EVC-HYB format, and make some concluding remarks based on the resulting performance model. We also construct several tests to qualitatively analyze the time distribution of the data movement for loading the matrix, the vector and other time intervals consumed in the computation in an effort to motivate the low SpMV performance of SpMV for certain matrix types. The tests indicate that the sparsity pattern of a matrix has a high impact on SpMV performance and needs to be considered for further optimization. We also implement the EVC-HYB format in PETSc, which results in improved performance compared to CUSP.

It is difficult to achieve optimal performance for SpMV for all sparse matrices using a simple format due to the irregularity and diversity of the access patterns. Thus, it is important to design a hybrid format that adaptively handles different sparsity patterns in a matrix, which is motivated by the hardware architecture, in order to achieve better memory bandwidth.

5 Acknowledgements

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07–25070) and the State of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign, the National Center for Supercomputing Applications, and the Great Lakes Consortium for Petascale Computation. This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy award DE–FG02–13ER26138/DE–SC0010049.

References

[1] W. K. Anderson, W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith, *Achieving high sustained performance in an unstructured mesh CFD application*, in Proceedings of the 1999 ACM/IEEE Conference on Supercomputing, SC '99, New York, NY, USA, 1999, ACM.

[2] A. Ashari, N. Sedaghati, J. Eisenlohr, and P. Sadayappan, *An efficient two-dimensional blocking strategy for sparse matrix-vector multiplication on GPUs*, in Proceedings of the 28th ACM international conference on Supercomputing, ACM, 2014, pp. 273–282.

[3] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, and H. Zhang, *PETSc users manual*, Tech. Rep. ANL-95/11 - Revision 3.5, Argonne National Laboratory, 2014.

[4] M. M. Baskaran and R. Bordawekar, *Optimizing sparse matrix-vector multiplication on GPUs*, IBM Research Report RC24704, IBM, Apr. 2009.

[5] N. Bell, S. Dalton, and L. Olson, *Exposing fine-grained parallelism in algebraic multigrid methods*, SIAM Journal on Scientific Computing, 34 (2012), pp. C123–C152.

[6] N. Bell and M. Garland, *Implementing sparse matrix-vector multiplication on throughput-oriented processors*, in Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09, New York, NY, USA, 2009, ACM, pp. 18:1–18:11.

[7] G. E. Blelloch, M. A. Heroux, and M. Zagha, *Segmented operations for sparse matrix computation on vector multiprocessors*, Tech. Rep. CMU-CS-93-173, School of Computer Science, Carnegie Mellon University, Aug. 1993.

[8] *Blue Waters.* National Center for Supercomputing Applications. https://bluewaters.ncsa.illinois.edu/.

[9] J. W. Choi, A. Singh, and R. W. Vuduc, *Model-driven autotuning of sparse matrix-vector multiply on GPUs*, in Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '10, New York, NY, USA, 2010, ACM, pp. 115–126.

[10] *Cusp: Generic parallel algorithms for sparse matrix and graph computations.* https://github.com/cusplibrary/cusplibrary.

[11] T. Davis and Y. Hu, *The University of Florida sparse matrix collection*, tech. rep., University of Florida, 2015.

[12] A. Dziekonski, A. Lamecki, and M. Mrozowski, *A memory efficient and fast sparse matrix vector product on a GPU*, Progress In Electromagnetics Research, 116 (2011), pp. 49–63.

[13] D. Guo and W. Gropp, *Adaptive thread distributions for SpMV on a GPU*, in Proceedings of the Extreme Scaling Workshop, BW-XSEDE '12, Champaign, IL, USA, 2012, University of Illinois at Urbana-Champaign, pp. 2:1–2:5.

[14] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. Bishop, *A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units*, SIAM Journal on Scientific Computing, 36 (2014), pp. C401–C423.

[15] R. Li and Y. Saad, *GPU-accelerated preconditioned iterative linear solvers*, J. Supercomput., 63 (2013), pp. 443–466.

[16] M. Maggioni and T. Berger-Wolf, *Adell: An adaptive warp-balancing ell format for efficient sparse matrix-vector multiplication on GPUs*, in Parallel Processing (ICPP), 2013 42nd International Conference on, IEEE, 2013, pp. 11–20.

[17] X. Mei, K. Zhao, C. Liu, and X. Chu, *Benchmarking the memory hierarchy of modern GPUs*, in Network and Parallel Computing, C.-H. Hsu, X. Shi, and V. Salapura, eds., vol. 8707 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2014, pp. 144–156.

[18] P. Micikevicius, *GPU performance analysis and optimization*, in GPU Technology Conference 2012, San Jose, CA, May 14–17 2012.

[19] Nvidia Corporation, *NVIDIA's Next Generation CUDA™ Compute Architecture: Kepler™ GK110*, 2012.

[20] Nvidia Corporporation, *cuSPARSE*. http://docs.nvidia.com/cuda/cusparse.

[21] I. Reguly and M. Giles, *Efficient sparse matrix-vector multiplication on cache-based GPUs*, in Innovative Parallel Computing (InPar), 2012, May 2012, pp. 1–12.

[22] S. Rennich, *Leveraging matrix block structure in sparse matrix-vector multiplication*, in GPU Technology Conference 2012, San Jose, CA, May 14–17 2012.

[23] *GPU STREAM benchmark*. NVIDIA Corporation. https://devtalk.nvidia.com/default/topic/381934/stream-benchmark/.

[24] X. Sun, Y. Zhang, T. Wang, X. Zhang, L. Yuan, and L. Rao, *Optimizing SpMV for diagonal sparse matrices on GPU*, in Parallel Processing (ICPP), 2011 International Conference on, Sept 2011, pp. 492–501.

[25] S. Williams, A. Waterman, and D. Patterson, *Roofline: An insightful visual performance model for multicore architectures*, Commun. ACM, 52 (2009), pp. 65–76.