CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication

Weifeng Liu, Brian Vinter
Niels Bohr Institute, University of Copenhagen
Copenhagen, Denmark
{weifeng.liu, vinter}@nbi.ku.dk

ABSTRACT

Sparse matrix-vector multiplication (SpMV) is a fundamental building block for numerous applications. In this paper, we propose CSR5 (Compressed Sparse Row 5), a new storage format, which offers high-throughput SpMV on various platforms including CPUs, GPUs and Xeon Phi. First, the CSR5 format is insensitive to the sparsity structure of the input matrix. Thus the single format can support an SpMV algorithm that is efficient both for regular matrices and for irregular matrices. Furthermore, we show that the overhead of the format conversion from the CSR to the CSR5 can be as low as the cost of a few SpMV operations.

We compare the CSR5-based SpMV algorithm with 11 state-of-the-art formats and algorithms on four mainstream processors using 14 regular and 10 irregular matrices as a benchmark suite. For the 14 regular matrices in the suite, we achieve comparable or better performance over the previous work. For the 10 irregular matrices, the CSR5 obtains average performance improvement of 17.6%, 28.5%, 173.0% and 293.3% (up to 213.3%, 153.6%, 405.1% and 943.3%) over the best existing work on dual-socket Intel CPUs, an nVidia GPU, an AMD GPU and an Intel Xeon Phi, respectively. For real-world applications such as a solver with only tens of iterations, the CSR5 format can be more practical because of its low-overhead for format conversion.

Categories and Subject Descriptors

G.1.3 [**Numerical Linear Algebra**]: Sparse, structured, and very large systems (direct and iterative methods); G.4 [**Mathematical Software**]: Parallel and vector implementations

General Terms

Algorithms, Experimentation, Performance

ICS'15, June 8–11, 2015, Newport Beach, CA, USA.

ACM 978-1-4503-3559-1/15/06 ...$15.00.
http://dx.doi.org/10.1145/2751205.2751209 .

Keywords

Sparse Matrices, Storage Formats, SpMV, CSR, CSR5, CPU, GPU, Xeon Phi

1. INTRODUCTION

Over the past few decades, sparse matrix-vector multiplication (SpMV) has probably been the most studied sparse BLAS routine because of its importance in many scientific applications. The SpMV operation multiplies a sparse matrix A of size $m \times n$ by a dense vector x of size n and obtains a dense vector y of size m. Its naïve sequential implementation can be very simple, and can be easily parallelized by adding a few pragma directives for the compilers. But to accelerate large-scale computation, parallel SpMV is still required to be hand-optimized with specific data storage formats and algorithms [1, 2, 3, 4, 6, 7, 10, 13, 16, 17, 19, 21, 22, 24, 29, 30, 31, 33, 34].

As a result, a conflict may emerge between the requirements of SpMV and other sparse matrix operations such as preconditioning operations [22] and sparse matrix-matrix multiplication [23]. The reason is that those operations commonly require matrices stored in the basic formats such as the compressed sparse row (CSR). Therefore, when users construct a real-world application, they need to consider a cost of format conversion between the SpMV-oriented formats and the basic formats. Unfortunately, this conversion overhead may offset the benefits of using these specialized formats, in particular when only tens of iterations are needed in a solver.

The conversion cost is mainly from the expensive structure-dependent parameter tuning of a storage format. For example, some block-based formats require finding a good 2D block size [6, 7, 10, 31, 32, 34]. Moreover, some hybrid formats [4, 29] may need completely different partitioning parameters for distinct input matrices.

To avoid the format conversion overhead, a few algorithms have concentrated on accelerating CSR-based SpMV with either row block methods [1, 17] or segmented sum methods [5, 16]. However, each of the two types of methods has its own drawbacks. As for the row block methods, despite their good performance for regular matrices, they may provide very low performance for irregular matrices due to unavoidable load imbalance. In contrast, the segmented sum methods can achieve near perfect load balance, but suffer from high overhead due to more global synchronizations and global memory accesses. Furthermore, none of the above work can avoid an overhead from preprocessing, since certain auxiliary data

for the basic CSR format have to be generated for better load balancing [1, 17] or established primitives [5, 16].

Therefore, to be practical, an efficient format must satisfy two criteria: (1) it should limit format conversion cost by avoiding structure-dependent parameter tuning, and (2) it should support fast SpMV for both regular and irregular matrices.

To meet these two criteria, in this paper, we have designed CSR5 (Compressed Sparse Row 5)[1], a new format directly extending the classic CSR format. The CSR5 format leaves one of the three arrays of the CSR format unchanged, stores the other two arrays in an in-place tile-transposed order, and adds two groups of extra auxiliary information. The format conversion from the CSR to the CSR5 merely needs two tuning parameters: one is hardware-dependent and the other is sparsity-dependent (but structure-independent). Because the added two groups of information are usually much shorter than the original three in the CSR format, very limited extra space is required. Furthermore, the CSR5 format is SIMD-friendly and thus can be easily implemented on all mainstream processors with the SIMD units. Because of the structure-independence and the SIMD utilization, the CSR5-based SpMV algorithm can bring stable high throughput for both regular and irregular matrices.

In this paper, we make the following contributions:

- We propose CSR5, an efficient storage format with low conversion cost and high degree of parallelism.
- We present a CSR5-based SpMV algorithm based on a redesigned low-overhead segmented sum algorithm.
- We implement the work on four mainstream devices: CPU, nVidia GPU, AMD GPU and Intel Xeon Phi.
- We evaluate the CSR5 format in both isolated SpMV tests and iteration-based scenarios.

We compare the CSR5 with 11 state-of-the-art formats and algorithms on dual-socket Intel CPUs, an nVidia GPU, an AMD GPU and an Intel Xeon Phi. By using 14 regular and 10 irregular matrices as a benchmark suite, we show that the CSR5 obtains comparable or better performance over the previous work for the regular matrices, and can greatly outperform the prior work for the irregular matrices. As for the 10 irregular matrices, the CSR5 obtains average performance improvement of 17.6%, 28.5%, 173.0% and 293.3% (up to 213.3%, 153.6%, 405.1% and 943.3%) over the second best work on the four platforms, respectively. Moreover, for iteration-based real-world scenarios, the CSR5 format achieves higher speedups because of the fast format conversion. To the best of our knowledge, this is the first time that a single storage format can outperform state-of-the-art work on all four modern multicore and manycore processors.

2. PRELIMINARIES

2.1 The CSR Format

The CSR format for sparse matrices consists of three arrays: (1) `row_ptr` array which saves the start and end pointers of the nonzeros of the rows. It has size $m+1$, where m is the number of rows of the matrix, (2) `col_idx` array of size nnz stores column indices of the nonzeros, where nnz is the number of nonzeros of the matrix, and (3) `val` array of size nnz stores values of the nonzeros. Figure 1 shows an example.

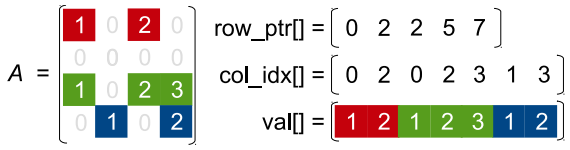


Figure 1: A sparse matrix and its CSR format.

2.2 Parallel Algorithms for CSR-based SpMV

2.2.1 *Row Block Methods*

In a given sparse matrix, rows are independent from each other. Therefore an SpMV operation can be parallelized on decomposed row blocks. A logical processing unit is responsible for a row block and stores dot product results of the matrix rows with the vector x to corresponding locations in the result y. When the SIMD units of a physical processing unit are available, the SIMD reduction sum operation can be utilized for higher efficiency. These two methods are respectively known as the CSR-scalar and the CSR-vector algorithms, and have been implemented on CPUs [33] and GPUs [4, 29]. Algorithm 1 shows a parallel CSR-scalar method.

Algorithm 1 SpMV using the CSR-scalar method.

```
1: for i = 0 to m − 1 in parallel do
2:     y[i] ← 0
3:     for j = row_ptr[i] to row_ptr[i + 1] − 1 do
4:         y[i] ← y[i] + val[j] × x[col_idx[j]]
5:     end for
6: end for
```

Despite the good parallelism, exploiting the scalability in modern multi-processors is not trivial for the row block methods. The performance problems mainly come from load imbalance for matrices which consist of rows with uneven lengths. Specifically, if one single row of a matrix is significantly longer than the other rows, only a single core can be fully used while the other cores in the same chip may be completely idle. Although various strategies, such as data streaming [11, 17], memory coalescing [13], data reordering or reconstruction [3, 18, 25], static or dynamic binning [1, 17] and Dynamic Parallelism [1], have been developed, none of those can fundamentally solve the problem of load imbalance, and thus provide relatively low SpMV performance for the CSR format.

2.2.2 *Segmented Sum Methods*

Blelloch et al. [5] pointed out that the segmented sum may be more attractive for the CSR-based SpMV, since it is SIMD friendly and insensitive to the sparsity structure of the input matrix, thus overcoming the shortcomings of the row block methods.

Segmented sum (which is a special case of the backward segmented scan) performs a reduction sum operation for the entries in each segment in an array. A segment has its first

[1]The reason we call the storage format CSR5 is that it has five groups of data, instead of three in the classic CSR.

entry flagged as TRUE and the other entries flagged as FALSE. Algorithm 2 lists a serial segmented sum algorithm. Vectorized parallel segmented sum algorithms can be found in [9, 15, 28].

Algorithm 2 Serial segmented sum operation.

```
1: function SEGMENTED_SUM(*in, *flag)
2:     length ← SIZEOF(*in)
3:     for i = 0 to length − 1 do
4:         if flag[i] = TRUE then
5:             j ← i + 1
6:             while flag[j] = FALSE && j < length do
7:                 in[i] ← in[i] + in[j]
8:                 j ← j + 1
9:             end while
10:        else
11:            in[i] ← 0
12:        end if
13:    end for
14: end function
```

In the SpMV operation, the segmented sum treats each matrix row as a segment and calculates a partial sum for the entry-wise products generated in each row. The SpMV operation using the segmented sum methods consists of four steps: (1) generating an auxiliary `bit_flag` array of size nnz from the `row_ptr` array. An entry in `bit_flag` is flagged as `TRUE` if its location matches the first nonzero entry of a row, otherwise it is flagged as `FALSE`, (2) calculating all intermediate entries (i.e., entry-wise products) to an array of size nnz, (3) executing the parallel segmented sum for the array, and (4) collecting all partial sums to the result vector y if a row is not empty. Algorithm 3 lists the pseudocode. Figure 2 illustrates an example using the matrix A plotted in Figure 1.

Algorithm 3 Segmented sum method CSR-based SpMV.

```
1: MALLOC(*bit_flag, nnz)
2: MEMSET(*bit_flag, FALSE)
3: for i = 0 to m − 1 in parallel do                ▷ Step 1
4:     bit_flag[row_ptr[i]] ← TRUE
5: end for
6: MALLOC(*product, nnz)
7: for j = 0 to nnz − 1 in parallel do              ▷ Step 2
8:     product[j] ← val[j] × x[col_idx[j]]
9: end for
10: SEGMENTED_SUM(*product, *bit_flag)              ▷ Step 3
11: for k = 0 to m − 1 in parallel do               ▷ Step 4
12:     if row_ptr[k] = row_ptr[k + 1] then
13:         y[k] ← 0
14:     else
15:         y[k] ← product[row_ptr[k]]
16:     end if
17: end for
18: FREE(*bit_flag)
19: FREE(*product)
```

We can see that once the heaviest workload, i.e., step 3, is parallelized through a fast segmented sum method described in [9, 15, 28], nearly perfect load balance can be expected in all steps of Algorithm 3. However, in this context, the load balanced computation does not mean high

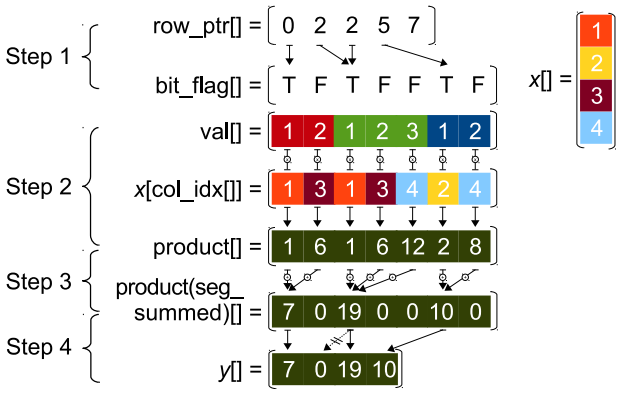


Figure 2: CSR-based SpMV using segmented sum.

performance. Figure 3 shows that the row block method in cuSPARSE v6.5 can significantly outperform the segmented sum method in cuDPP v2.2 [16, 28], while doing SpMV on relatively regular matrices (see Table 2 for the used benchmark suite).

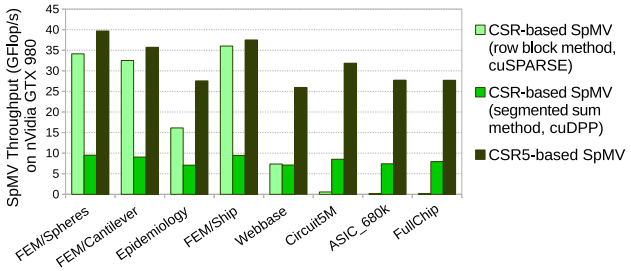


Figure 3: Single precision SpMV performance.

Why is this the case? We can see that the step 1 is a scatter operation and the step 4 is a gather operation, both from the row space of size m. This prevents the two steps from fusing with the steps 2 and 3 in the nonzero entry space of size nnz. In this case, more global synchronizations and global memory accesses may degrade the overall performance. Previous research [4, 29] has found that the segmented sum may be more suitable for the COO (coordinate storage format) based SpMV, since the fully stored row index data can convert the steps 1 and 4 to the nonzero entry space: the `bit_flag` array can be generated by comparison of neighbor row indices, and the partial sums in the `product` array can be directly saved to y since their final locations are easily known from the row index array. Further, Yan et al. [34] and Tang et al. [30] reported that some variants of the COO format can also benefit from the segmented sum. However, it is well known that accessing row indices in the COO pattern brings higher off-chip memory pressure, which is just what the CSR format tries to avoid.

In the following, we will show that the CSR5-based SpMV can utilize both the segmented sum for load balance and the compressed row data for better load/store efficiency. In this way, the CSR5-based SpMV can obtain up to 4x speedup (see Figure 3) over the CSR-based SpMV using the segmented sum primitive [28].

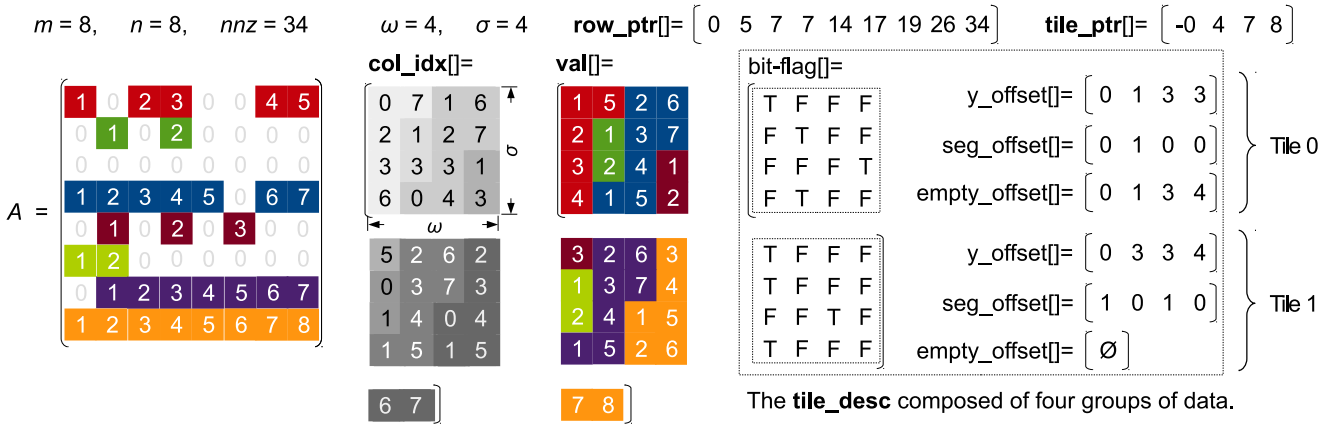


Figure 4: The CSR5 storage format of a sparse matrix A of size 8×8. The five groups of information include `row_ptr`, `tile_ptr`, `col_idx`, `val` and `tile_desc`.

3. THE CSR5 STORAGE FORMAT

3.1 Basic Data Layout

To achieve near-optimal load balance for matrices with any sparsity structures, we first evenly partition all nonzero entries to multiple 2D tiles of the same size. Thus when executing parallel SpMV operation, a compute core can consume one or more 2D tiles, and each SIMD lane of the core can deal with one column of a tile. Then the main skeleton of the CSR5 format is simply a group of 2D tiles. The CSR5 format has two tuning parameters: ω and σ, where ω is a tile's width and σ is its height. In fact, the CSR5 format *only has* these two tuning parameters.

Further, we need extra information to efficiently compute SpMV. For each tile, we introduce a tile pointer `tile_ptr` and a tile descriptor `tile_desc`. Meanwhile, the three arrays, i.e., row pointer `row_ptr`, column index `col_idx` and value `val`, of the classic CSR format are directly integrated. The only difference is that the `col_idx` data and the `val` data in each complete tile are in-place transposed (i.e., from row-major order to column-major order) for coalesced memory access from contiguous SIMD lanes. If the last entries of the matrix do not fill up a complete 2D tile (i.e., nnz mod $(\omega\sigma) \neq 0$), they just remain unchanged and discard their `tile_desc`.

In Figure 4, an example matrix A of size 8×8 with 34 nonzero entries is stored in the CSR5 format. When $\omega = 4$ and $\sigma = 4$, the matrix is divided into three tiles including two complete tiles of size 16 and one incomplete tile of size 2. The arrays `col_idx` and `val` in the two complete tiles are stored in tile-level column-major order now. Moreover, only the first two tiles have `tile_desc`, since they are complete.

3.2 Auto-Tuned Parameters ω and σ

Because the computational power of the modern multicore or manycore processors is mainly from the SIMD units, we design an auto-tuning strategy for high SIMD utilization.

First, the tile width ω is set to the size of the SIMD execution unit of the used processor. Then an SIMD unit can consume a 2D tile in σ steps without any explicit synchronization, and the vector registers can be fully utilized. For the double precision SpMV, we always set $\omega = 4$ for CPUs with 256-bit SIMD units, $\omega = 32$ for the nVidia GPUs, $\omega = 64$ for the AMD GPUs, and $\omega = 8$ for Intel Xeon Phi with 512-bit SIMD units. Therefore, ω can be automatically decided once the processor type used is known.

The other parameter σ is decided by a slightly more complex process. For a given processor, we consider its on-chip memory strategy such as cache capacity and prefetching mechanism. If a 2D tile of size $\omega \times \sigma$ can empirically bring better performance than using the other sizes, the σ is simply chosen. We found that the x86 processors fall into this category. For the double precision SpMV on CPUs and Xeon Phi, we always set σ to 16 and 12, respectively.

As for GPUs, the tile height σ further depends on the sparsity of the matrix. Note that the "sparsity" is not equal to "sparsity structure". We define "sparsity" to be the average number of nonzero entries per row (or nnz/row for short). In contrast, "sparsity structure" is much more complex because it includes 2D space layout of all nonzero entries.

On GPUs, we have several performance considerations on mapping the value nnz/row to σ. First, σ should be large enough to expose more thread-level local work and to amortize a basic cost of the segmented sum algorithm. Second, it should not be too large since a larger tile potentially generates more partial sums (i.e., entries to store to y), which bring higher pressure to last level cache write. Moreover, for the matrices with large nnz/row, σ may need to be small. The reason is that once the whole tile is located inside a matrix row (i.e., only one segment is in the tile), the segmented sum converts to a fast reduction sum.

Therefore, for the nnz/row to σ mapping on GPUs, we define three simple bounds: r, s and t. The first bound r is designed to prevent a too small σ. The second bound s is used for preventing a too large σ. But when nnz/row is further larger than the third bound t, σ is set to a small value u. Then we have

$$\sigma = \begin{cases} r & \text{if} \quad nnz/\text{row} \leq r \\ nnz/\text{row} & \text{if} \quad r < nnz/\text{row} \leq s \\ s & \text{if} \quad s < nnz/\text{row} \leq t \\ u & \text{if} \quad t < nnz/\text{row}. \end{cases}$$

The three bounds, r, s and t, and the value u are hardware-dependent, meaning that for a given processor, they can be fixed for use. For example, to execute double precision SpMV on nVidia Maxwell GPUs and AMD GCN GPUs, we

always set <r, s, t, u> = <4, 32, 256, 4> and <4, 7, 256, 4>, respectively. As for future processors with new architectures, we can obtain the four values through some simple benchmarks during initialization, and then use them for later runs. So the parameter σ can be decided once the very basic information of a matrix and a low-level hardware are known.

Therefore, we can see that the parameter tuning time becomes negligible because ω and σ are easily obtained. This can save a great deal of preprocessing time.

3.3 Tile Pointer Information

The added tile pointer information `tile_ptr` stores the row index of the first matrix row in each tile, indicating the starting position for storing its partial sums to the vector y. By introducing `tile_ptr`, each tile can find its own starting position, allowing tiles to execute in parallel. The size of the `tile_ptr` array is $p + 1$, where $p = \lceil nnz/(\omega\sigma) \rceil$ is the number of tiles in the matrix. For the example in Figure 4, the first entry of Tile 1 is located in the $4th$ row of the matrix, and thus 4 is set as its tile pointer. To build the array, we binary search the index of the first nonzero entry of each tile on the `row_ptr` array. Lines 1–4 in Algorithm 4 show this procedure.

Recall that an empty row has exactly the same row pointer information as its first non-empty right neighbor row (see the second row in the matrix A in Figure 1). Thus for the non-empty rows with an empty left neighbor, we need a specific process (which is similar to lines 12–16 in Algorithm 3) to store their partial sums to correct positions in y. To recognize whether the specific process is required, we give a hint to the other parts (i.e., tile descriptor data) of the CSR5 format and the CSR5-based SpMV algorithm. Here we set an entry in `tile_ptr` to its negative value, if its corresponding tile includes any empty rows. Lines 5–12 in Algorithm 4 show this operation.

Algorithm 4 Generating `tile_ptr`.

```
 1: for tid = 0 to p in parallel do
 2:     bnd ← tid × ω × σ
 3:     tile_ptr[tid] ← BINARY_SEARCH(*row_ptr, bnd) −1
 4: end for
 5: for tid = 0 to p − 1 do
 6:     for rid = tile_ptr[tid] to tile_ptr[tid + 1] do
 7:         if row_ptr[rid] = row_ptr[rid + 1] then
 8:             tile_ptr[tid] ← NEGATIVE(tile_ptr[tid])
 9:             break
10:         end if
11:     end for
12: end for
```

If the first tile has any empty rows, we need to store a -0 (negative zero) for it. To record -0, here we use unsigned 32- or 64-bit integer as data type of the `tile_ptr` array. Therefore, we have 1 bit for explicitly storing the sign and 31 or 63 bits for an index. For example, in our design, tile pointer -0 is represented as a binary style '1000 ... 000', and tile pointer 0 is stored as '0000 ... 000'. To the best of our knowledge, the index of 31 or 63 bits is completely compatible to most numerical libraries such as Intel MKL. Moreover, reference implementation of the recent high performance conjugate gradient (HPCG) benchmark [14] also uses 32-bit signed integer for problem dimension no more than 2^{31} and 64-bit signed integer for problem dimension larger than that. Thus it is safe to save 1 bit as the empty row hint and the other 31 or 63 bits as a 'real' row index.

3.4 Tile Descriptor Information

Only having the tile pointer is not enough for a fast SpMV operation. For each tile, we also need four extra hints: (1) `bit_flag` of size $\omega \times \sigma$, which indicates whether an entry is the first nonzero of a matrix row, (2) `y_offset` of size ω used to further let each column know where the starting point to store its local partial sums is, (3) `seg_offset` of size ω used to accelerate the local segmented sum inside a tile, and (4) `empty_offset` of unfixed size (but no longer than $\omega \times \sigma$) constructed to help the partial sums to find correct locations in y if the tile includes any empty rows. The tile descriptor `tile_desc` is defined to denote a combination of the above four groups of data.

Generating `bit_flag` is straightforward. The procedure is very similar to lines 3–5 in Algorithm 3. The main difference is that the bit flags are saved in column-major order, which matches the in-place transposed `col_idx` and `val`. Additionally, the first entry of each tile's `bit_flag` is set to `TRUE` for sealing the first segment from the top and letting 2D tiles to be independent from each other.

The array `y_offset` of size ω is used to help the columns in each tile knowing where the starting points to store their partial sums to y are. In other words, each column has one entry in the array `y_offset` as a starting point offset for all segments in the same column. We save a row index offset (i.e., relative row index) for each column in `y_offset`. Thus for the ith column in the tidth tile, by calculating `tile_ptr[`tid`]` + `y_offset[`i`]`, the column knows where its own starting position in y is. Thus the columns can work in a high degree of parallelism without waiting for a synchronization. Generating `y_offset` is simple: each column counts the number of `TRUE`s in its previous columns' `bit_flag` array. Consider Tile 1 in Figure 4 as an example: because there are 3 `TRUE`s in the 1st column, the 2nd column's corresponding value in `y_offset` is 3. In addition, since there are in total 4 `TRUE`s in the 1st, 2nd and 3rd columns' `bit_flag`, Tile 1's `y_offset[3]`= 4. Algorithm 5 lists how to generate `y_offset` for a single 2D tile in an SIMD-friendly way.

Algorithm 5 Generating `y_offset` and `seg_offset`.

```
 1: MALLOC(*tmp_bit, ω)
 2: MEMSET(*tmp_bit, FALSE)
 3: for i = 0 to ω − 1 in parallel do
 4:     y_offset[i] ← 0
 5:     for j = 0 to σ − 1 do
 6:         y_offset[i] ← y_offset[i] + bit_flag[i][j]
 7:         tmp_bit[i] ← tmp_bit[i] ∨ bit_flag[i][j]
 8:     end for
 9:     seg_offset[i] ← 1− tmp_bit[i]
10: end for
11: EXCLUSIVE_PREFIX_SUM_SCAN(*y_offset)
12: SEGMENTED_SUM(*seg_offset, *tmp_bit)
13: FREE(*tmp_bit)
```

The third array `seg_offset` of size ω is used for accelerating a local segmented sum in the workload of each tile. The local segmented sum is an essential step that synchronizes partial sums in a 2D tile (imagine multiple columns in the tile come from the same matrix row). In the previous segmented sum (or segmented scan) method [5, 9, 28,

15], the local segmented sum is complex and not efficient enough. Thus we prepare `seg_offset` as an auxiliary array to facilitate implementation of segmented sum by way of the prefix-sum scan, which is a well optimized fundamental primitive for the SIMD units.

To generate `seg_offset`, we let each column search its right neighbor columns and count the number of contiguous columns without any `TRUE`s in their `bit_flag`. Using Tile 0 in Figure 4 as an example, its 2nd column has one and only one right neighbor column (the 3rd column) without any `TRUE`s in its `bit_flag`. Thus the 2nd column's `seg_offset` value is 1. In contrast, because the other three columns (the 1st, 3rd and 4th) do not have any 'all `FALSE`' right neighbors, their values in `seg_offset` is 0. Algorithm 5 shows how to generate `seg_offset` using an SIMD-friendly method.

Algorithm 6 and Figure 5 show the fast segmented sum using `seg_offset` and an inclusive prefix-sum scan. The principle of this operation is that the prefix-sum scan is essentially an increment operation. Once a segment knows the distance (i.e., offset) between its head and its tail, its partial sum can be deduced from its prefix-sum scan results. Therefore, the more complex segmented sum operation in [5, 9, 28, 15] can be converted to a faster prefix-sum scan operation (line 5) and a few arithmetic operations (lines 6–8).

Algorithm 6 Fast segmented sum using `seg_offset`.

```
1: function FAST_SEGMENTED_SUM(*in, *seg_offset)
2:     length ← SIZEOF(*in)
3:     MALLOC(*tmp, length)
4:     MEMCPY(*tmp, *in)
5:     INCLUSIVE_PREFIX_SUM_SCAN(*in)
6:     for i = 0 to length − 1 in parallel do
7:         in[i] ← in[i+seg_offset[i]] − in[i] + tmp[i]
8:     end for
9:     FREE(*tmp)
10: end function
```

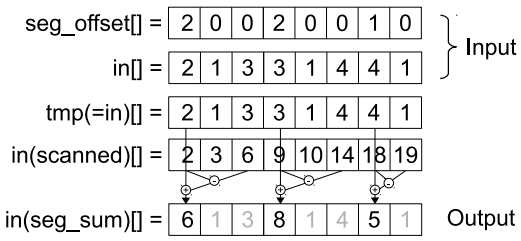


Figure 5: An example of the fast segmented sum.

The last array `empty_offset` occurs when and only when a 2D tile includes any empty rows (i.e., its tile pointer is negative). Because an empty row of a matrix has the same row pointer with its rightmost non-empty neighbor row (recall the second row in the matrix A in Figure 1), `y_offset` will record an incorrect offset for it. We correct for this by storing correct offsets for segments within a tile. Thus the length of `empty_offset` is the number of segments (i.e., the total number of `TRUE`s in `bit_flag`) in a tile. For example, Tile 0 in Figure 4 has 4 entries in its `empty_offset` since its `bit_flag` includes 4 `TRUE`s. Algorithm 7 lists the pseudocode that generates `empty_offset` for a tile that contains at least one empty row.

Algorithm 7 Generating `empty_offset` for the tidth tile.

```
1: length ← REDUCTION_SUM(*bit_flag)
2: MALLOC(*empty_offset, length)
3: eid ← 0
4: for i = 0 to ω − 1 do
5:     for j = 0 to σ − 1 do
6:         if bit_flag[i][j] = TRUE then
7:             ptr ← tid × ω × σ + i × σ + j
8:             idx ← BINARY_SEARCH(*row_ptr, ptr) −1
9:             idx ← idx − REMOVE_SIGN(tile_ptr[tid])
10:            empty_offset[eid] ← idx
11:            eid ← eid + 1
12:        end if
13:    end for
14: end for
```

3.5 Storage Details

To store the `tile_desc` arrays in a space-efficient way, we find upper bounds to the entries and utilize the bit-field pattern. First, since entries in `y_offset` store offset distances inside a 2D tile, they have an upper bound of $\omega\sigma$. So $\lceil \log_2(\omega\sigma) \rceil$ bits are enough for each entry in `y_offset`. For example, when $\omega = 32$ and $\sigma = 16$, 9 bits are enough for each entry. Second, since `seg_offset` includes offsets less than ω, $\lceil \log_2(\omega) \rceil$ bits are enough for an entry in this array. For example, when $\omega = 32$, 5 bits are enough for each entry. Third, `bit_flag` stores σ 1-bit flags for each column of a 2D tile. When $\sigma = 16$, each column needs 16 bits. So 30 (i.e., $9 + 5 + 16$) bits are enough for each column in the example. Therefore, for a tile, the three arrays can be stored in a compact bit-field composed of ω 32-bit unsigned integers. If the above example matrix has 32-bit integer row index and 64-bit double precision values, only around 2% extra space is required by the three newly added arrays.

The size of `empty_offset` depends on the number of groups of contiguous empty rows, since we only record one offset for the rightmost non-empty row with any number of empty rows as its left neighbors.

3.6 The CSR5 for Other Matrix Operations

Since we in-place transposed the CSR arrays `col_idx` and `val`, a conversion from the CSR5 to the CSR is required for doing other sparse matrix operations using the CSR format. This conversion is simply removing `tile_ptr` and `tile_desc` and transposing `col_idx` and `val` back to row-major order. Thus the conversion can be very fast. Further, since the CSR5 is a superset of the CSR, any entry accesses or slight changes can be done directly in the CSR5 format, without any need to convert it to the CSR format. Additionally, some applications such as finite element methods can directly assemble sparse matrices in the CSR5 format from data sources.

4. THE CSR5-BASED SPMV ALGORITHM

Because all computations of the information (`tile_ptr`, `tile_desc`, `col_idx` and `val`) of 2D tiles are independent of each other, they can execute concurrently. On GPUs, we assign a bunch of threads (i.e., warp in nVidia GPUs or wavefront in AMD GPUs) for each tile. On CPUs and Xeon Phi, we use OpenMP pragma for assigning the tiles to available x86 cores. Furthermore, the columns inside a tile

are independent of each other as well. So we assign a thread on GPU cores or an SIMD lane on x86 cores to each column in a tile.

While running the CSR5-based SpMV, each column in a tile can extract information from `bit_flag` and label the segments in its local data to three colors: (1) *red* means a sub-segment unsealed from its top, (2) *green* means a completely sealed segment existed in the middle, and (3) *blue* means a sub-segment unsealed from its bottom. There is an exception that if a column is unsealed both from its top and from its bottom, it is colored to *red*.

Algorithm 8 shows the pseudocode of the CSR5-based SpMV algorithm. Figure 6 plots an example of this procedure. We can see that the green segments can directly save their partial sums to y without any synchronization, since the indices can be calculated by using `tile_ptr` and `y_offset`. In contrast, the red and the blue sub-segments have to further add their partial sums together, since they are not complete segments. For example, the sub-segments B_2, R_2 and R_3 in Figure 6 have contributions to the same row, thus an addition is required. This addition operation needs the fast segmented sum shown in Algorithm 6 and Figure 5. Furthermore, if a tile has any empty rows, the `empty_offset` array is accessed to get correct global indices in y.

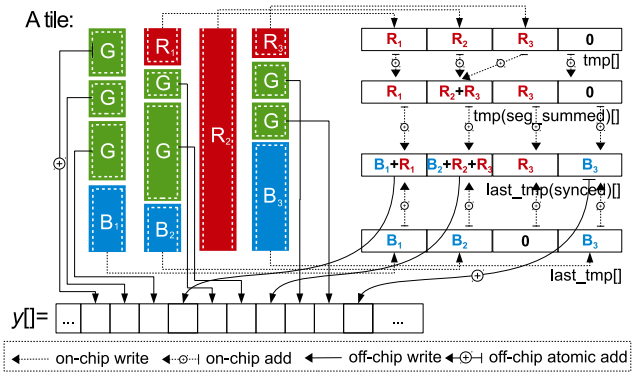


Figure 6: The CSR5-based SpMV in a tile. Partial sums of the green segments are directly stored to y. The red and the blue sub-segments require an extra segmented sum before issuing off-chip write.

Consider the synchronization among the tiles, since the same matrix row can be influenced by multiple 2D tiles running concurrently, the first and the last segments of a tile need to store to y by atomic add (or a global auxiliary array used in device-level reduction, scan or segmented scan [15, 28]). In Figure 6, the atomic add operations are highlighted by arrow lines with plus signs.

For the last entries not in a complete tile (e.g., the last two nonzero entries of the matrix in Figure 4), we execute a conventional CSR-vector method after all of the complete 2D tiles have been consumed. Note that even though the last tile (i.e., the incomplete one) does not have `tile_desc` arrays, it can extract a starting position from `tile_ptr`.

In Algorithm 8, we can see that the main computation (lines 5–21) only contains very basic arithmetic and logic operations that can be easily programmed on all mainstream processors with SIMD units. As the most complex part in our algorithm, the fast segmented sum operation (line 22) only requires a prefix-sum scan, which has been well-studied and can be efficiently implemented by using CUDA, OpenCL or x86 SIMD intrinsics.

Algorithm 8 The CSR5-based SpMV for the *tid*th tile.

```
1: MALLOC(*tmp, ω)
2: MEMSET(*tmp, 0)
3: MALLOC(*last_tmp, ω)
4: /*use empty_offset[y_offset[i]] instead of
   y_offset[i] for a tile with any empty rows*/
5: for i = 0 to ω − 1 in parallel do
6:     sum ← 0
7:     for j = 0 to σ − 1 do
8:         ptr ← tid × ω × σ + j × ω + i
9:         sum ← sum + val[ptr] × x[col_idx[ptr]]
10:        /*check bit_flag[i][j]*/
11:        if /*end of a red sub-segment*/ then
12:            tmp[i − 1] ← sum
13:            sum ← 0
14:        else if /*end of a green segment*/ then
15:            y[tile_ptr[tid] + y_offset[i]] ← sum
16:            y_offset[i] ← y_offset[i] +1
17:            sum ← 0
18:        end if
19:    end for
20:    last_tmp[i] ← sum //end of a blue sub-segment
21: end for
22: FAST_SEGMENTED_SUM(*tmp, *seg_offset)        ▷ Alg. 6
23: for i = 0 to ω − 1 in parallel do
24:    last_tmp[i] ← last_tmp[i] + tmp[i]
25:    y[tile_ptr[tid] + y_offset[i]] ← last_tmp[i]
26: end for
27: FREE(*tmp)
28: FREE(*last_tmp)
```

5. EXPERIMENTAL RESULTS

5.1 Experimental Setup

We evaluate the CSR5-based SpMV and 11 state-of-the-art formats and algorithms on four mainstream platforms: dual-socket Intel CPUs, an nVidia GPU, an AMD GPU and an Intel Xeon Phi. The platforms and participating approaches are shown in Table 1.

Host of the two GPUs is a machine with AMD A10-7850K APU, dual-channel DDR3-1600 memory and 64-bit Ubuntu Linux v14.04 installed. Host of the Xeon Phi is a machine with Intel Xeon E5-2680 v2 CPU, quad-channel DDR3-1600 memory and 64-bit Red Hat Enterprise Linux v6.5 installed. The two GPU platforms use the g++ compiler v4.8.2. The two Intel machines always set the Intel C/C++ complier 15.0.1 as default.

Here we evaluate double precision SpMV. So cuDPP library [16, 28], clSpMV [29] and yaSpMV [34] are not included since they only support single precision floating point as data type. Two recently published methods [20, 30] are not tested since the source code is not available to us yet.

We use OpenCL profiling scheme for timing SpMV on the AMD platform and record wall-clock time on the other three platforms. For all participating formats and algorithms, we evaluate SpMV 10 times (each time contains 1000 runs and records the average) and report the best observed result.

The testbeds	The participating formats and algorithms
Dual-socket Intel Xeon E5-2667 v3 (Haswell, 2×8 cores @ 3.2 GHz, 1.64 SP TFlops, 819.2 DP GFlops, 64 GB GDDR4, ECC-on, 2×68.3 GB/s bandwidth).	(1) The CSR-based SpMV in Intel MKL 11.2 Update 1. (2) BiCSB v1.2 using CSB [7] with bitmasked register block [6]. (3) pOSKI v1.0.0 [8] using OSKI v1.0.1h [31, 32] kernels. (4) The CSR5-based SpMV implemented by using OpenMP and AVX2 intrinsics.
An nVidia GeForce GTX 980 (Maxwell GM204, 2048 CUDA cores @ 1.13 GHz, 4.61 SP TFlops, 144.1 DP GFlops, 4 GB GDDR5, 224 GB/s bandwidth, driver v344.16).	(1) The best CSR-based SpMV [4] from cuSPARSE v6.5 and CUSP v0.4.0 [11]. (2) The best HYB [4] from the above two libraries. (3) BRC [2] with texture cache enabled. (4) ACSR [1] with texture cache enabled. (5) The CSR5-based SpMV implemented by using CUDA v6.5.
An AMD Radeon R9 290X (GCN Hawaii, 2816 Radeon cores @ 1.05 GHz, 5.91 SP TFlops, 739.2 DP GFlops, 4 GB GDDR5, 345.6 GB/s bandwidth, driver v14.41).	(1) The CSR-vector method [4] extracted from CUSP v0.4.0 [11]. (2) The CSR-Adaptive algorithm [17] implemented in ViennaCL v1.6.2 [26]. (3) The CSR5-based SpMV implemented by using OpenCL v1.2.
An Intel Xeon Phi 5110p (Knights Corner, 60 x86 cores @ 1.05 GHz, 2.02 SP TFlops, 1.01 DP TFlops, 8 GB GDDR5, ECC-on, 320 GB/s bandwidth, driver v3.4-1, μOS v2.6.38.8).	(1) The CSR-based SpMV in Intel MKL 11.2 Update 1. (2) The ESB [24] with dynamic scheduling enabled. (3) The CSR5-based SpMV implemented by using OpenMP and MIC-KNC intrinsics.

Table 1: The testbeds and participating formats and algorithms.

5.2 Benchmark Suite

In Table 2, we list 24 sparse matrices as our benchmark suite for all platforms. The first 20 matrices have been widely adopted in previous SpMV research [2, 4, 17, 24, 29, 33, 34]. The other 4 matrices are chosen since they have more diverse sparsity structures. All matrices except *Dense* are downloadable at the University of Florida Sparse Matrix Collection [12].

To achieve a high degree of differentiation, we categorize the 24 matrices in Table 2 into two groups: (1) *regular* group with the upper 14 matrices, (2) *irregular* group with the lower 10 matrices. This classification is mainly based on the minimum, average and maximum lengths of the rows. Matrix *dc2* is a representative of the group of irregular matrices. Its longest single row contains 114K nonzero entries, i.e., 15% nonzero entries of the whole matrix with 117K rows. This sparsity pattern challenges the design of efficient storage format and SpMV algorithm.

Id	Name	*Dimensions*	*nnz*	*nnz* per row (min, avg, max)
r1	Dense	2K×2K	4.0M	2K, 2K, 2K
r2	Protein	36K×36K	4.3M	18, 119, 204
r3	FEM/Spheres	83K×83K	6.0M	1, 72, 81
r4	FEM/Cantilever	62K×62K	4.0M	1, 64, 78
r5	Wind Tunnel	218K×218K	11.6M	2, 53, 180
r6	QCD	49K×49K	1.9M	39, 39, 39
r7	Epidemiology	526K×526K	2.1M	2, 3, 4
r8	FEM/Harbor	47K×47K	2.4M	4, 50, 145
r9	FEM/Ship	141K×141K	7.8M	24, 55, 102
r10	Economics	207K×207K	1.3M	1, 6, 44
r11	FEM/Accelerator	121K×121K	2.6M	0, 21, 81
r12	Circuit	171K×171K	959K	1, 5, 353
r13	Ga41As41H72	268K×268K	18.5M	18, 68, 702
r14	Si41Ge41H72	186K×186K	15.0M	13, 80, 662
i1	Webbase	1M×1M	3.1M	1, 3, 4.7K
i2	LP	4K×1.1M	11.3M	1, 2.6K, 56.2K
i3	Circuit5M	5.6M×5.6M	59.5M	1, 10, 1.29M
i4	eu-2005	863K×863K	19.2M	0, 22, 6.9K
i5	in-2004	1.4M×1.4M	16.9M	0, 12, 7.8K
i6	mip1	66K×66K	10.4M	4, 155, 66.4K
i7	ASIC_680k	683K×683K	3.9M	1, 6, 395K
i8	dc2	117K×117K	766K	1, 7, 114K
i9	FullChip	2.9M×2.9M	26.6M	1, 9, 2.3M
i10	ins2	309K×309K	2.8M	5, 9, 309K

Table 2: The benchmark suite.

5.3 Isolated SpMV Performance

Figure 7 shows double precision SpMV performance of the 14 regular matrices on the four platforms. We can see that, on average, all participating algorithms deliver comparable performance. *On the CPU platform*, Intel MKL obtains the best performance on average and the other 3 methods behave similar. *On the nVidia GPU*, the CSR5 delivers the highest throughput. The ACSR format is slower than the others, because its binning strategy leads to non-coalesced memory access. *On the AMD GPU*, the CSR5 achieves the best performance. Although the dynamic assigning in the CSR-Adaptive method can obtain better scalability than the CSR-vector method, it still cannot achieve near perfect load balance. *On the Xeon Phi*, the CSR5 is slower than Intel MKL and the ESB format. The main reason is that the current generation of Xeon Phi can only issue up to 4 relatively slow threads per core (i.e., up to 4×60 threads in total on the used device), and thus the latency of gathering entries from vector x becomes the main bottleneck. Then reordering or partitioning nonzero entries based on the column index for better cache locality behaves well in the ESB-based SpMV. However, in Section 5.6 we will show that this strategy leads to very high preprocessing cost.

Figure 8 shows double precision SpMV performance of the 10 irregular matrices. We can see that the irregularity can dramatically impact SpMV throughput of some approaches. *On the CPU platform*, the row block method based Intel MKL is now slower than the other methods. The CSR5 outperforms the others because of better SIMD efficiency from the AVX2 intrinsics. *On the nVidia GPU*, the CSR5 brings the best performance because of the near perfect load balance. The other two irregularity-oriented formats, HYB and ACSR, behave well but still suffer from imbalanced work decomposition. Note that the ACSR format is based on Dynamic Parallelism, a technical feature only available on recently released nVidia GPUs. *On the AMD GPU*, the CSR5 greatly outperforms the other two algorithms using the row block methods. Because the minimum work unit of the CSR-Adaptive method is one row, the method delivers degraded

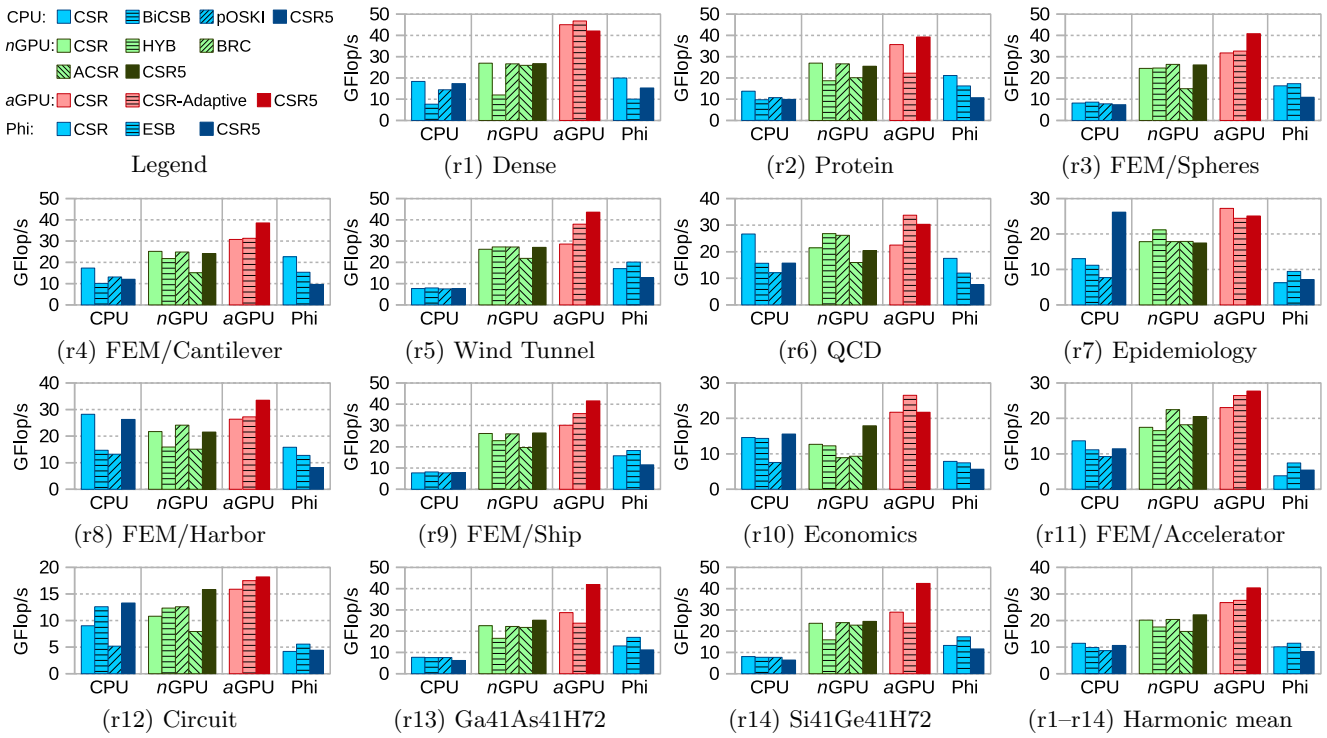


Figure 7: The SpMV performance of the 14 regular matrices. (*n*GPU=nVidia GPU, *a*GPU=AMD GPU)

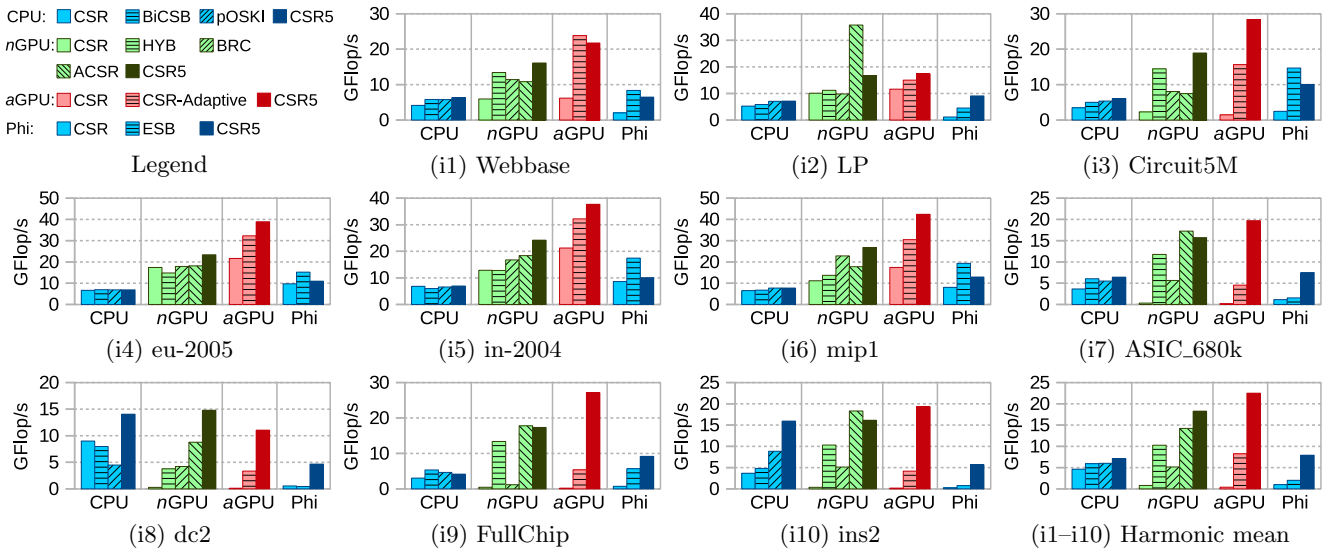


Figure 8: The SpMV performance of the 10 irregular matrices. (*n*GPU=nVidia GPU, *a*GPU=AMD GPU)

performance for matrices with very long rows[2]. *On the Xeon Phi*, the CSR5 can greatly outperform the other two methods in particular when matrices are too irregular to expose cache locality of x by the ESB format. Furthermore, since ESB is designed on top of the ELLPACK format, it cannot obtain the best performance for some irregular matrices.

Overall, the CSR5 achieves better performance (on the two GPU devices) or comparable performance (on the two x86 devices) for the 14 regular matrices. For the 10 irregular matrices, compared to pOSKI, ACSR, CSR-Adaptive and ESB as the second best methods, the CSR5 obtains average performance gain of 17.6%, 28.5%, 173.0% and 293.3% (up to 213.3%, 153.6%, 405.1% and 943.3%), respectively.

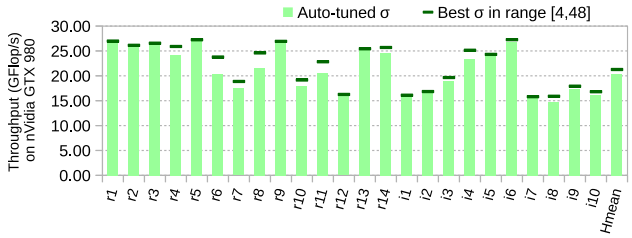
[2]Note that we use an implementation of the CSR-Adaptive from the ViennaCL Library. The AMD's version of the CSR-Adaptive, which is not available to us yet, may have slightly different performance.

Benchmark	The 14 regular matrices					The 10 irregular matrices				
Metrics	Preprocessing to SpMV ratio	Speedup of #iter.=50		Speedup of #iter.=500		Preprocessing to SpMV ratio	Speedup of #iter.=50		Speedup of #iter.=500	
		avg	best	avg	best		avg	best	avg	best
CPU-BiCSB	538.01x	0.06x	0.11x	0.35x	0.60x	331.77x	0.13x	0.24x	0.60x	1.07x
CPU-pOSKI	12.30x	0.43x	0.88x	0.57x	0.99x	10.71x	0.62x	1.66x	0.83x	2.43x
CPU-CSR5	**6.14x**	0.52x	0.74x	0.59x	0.96x	**3.69x**	0.91x	**2.37x**	**1.03x**	**2.93x**
*n*GPU-HYB	13.73x	0.73x	0.98x	0.92x	1.21x	28.59x	1.86x	13.61x	2.77x	25.57x
*n*GPU-BRC	151.21x	0.26x	0.31x	0.80x	0.98x	51.85x	1.17x	7.60x	2.49x	15.47x
*n*GPU-ACSR	**1.10x**	0.68x	0.93x	0.72x	1.03x	3.04x	5.05x	41.47x	5.41x	51.95x
*n*GPU-CSR5	3.06x	**1.04x**	**1.34x**	**1.10x**	**1.45x**	**1.99x**	**6.43x**	**48.37x**	**6.77x**	**52.31x**
*a*GPU-CSR-Adaptive	**2.68x**	1.00x	1.33x	1.07x	1.48x	**1.16x**	3.02x	27.88x	3.11x	28.22x
*a*GPU-CSR5	4.99x	**1.04x**	**1.39x**	**1.14x**	**1.51x**	3.10x	**5.72x**	**135.32x**	**6.04x**	**141.94x**
Phi-ESB	922.47x	0.05x	0.15x	0.33x	0.88x	222.19x	0.27x	1.15x	1.30x	2.96x
Phi-CSR5	**11.52x**	0.54x	**1.14x**	0.65x	**1.39x**	**9.45x**	**3.43x**	**18.48x**	**4.10x**	**21.18x**

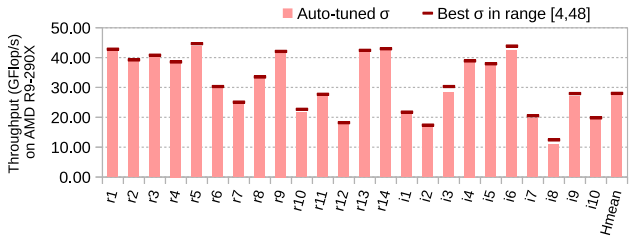
Table 3: Preprocessing cost and its impact on the iteration-based scenarios.

5.4 Effects of Auto-Tuning

In section 3.2, we discussed a simple auto-tuning scheme for the parameter σ on GPUs. Figure 9 shows its effects (the x axis is the matrix *id*s). We can see that compared to the best performance chosen from a range of $\sigma = 4$ to 48, the auto-tuned σ does not have obvious performance loss. On the nVidia GPU, the performance loss is on average -4.2%. On the AMD GPU, the value is on average -2.5%.



(a) The nVidia GTX 980 GPU.

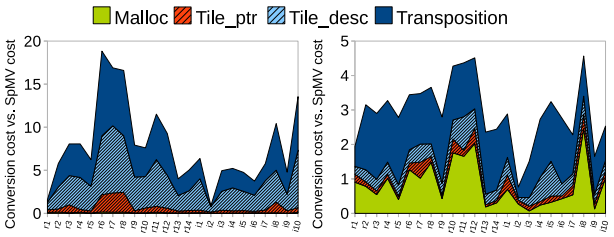


(b) The AMD R9-290X GPU.

Figure 9: Auto-tuning effects on the two GPUs.

5.5 Format Conversion Cost

The format conversion from the CSR to the CSR5 includes four steps: (1) memory allocation, (2) generating `tile_ptr`, (3) generating `tile_desc`, and (4) transposition of `col_idx` and `val` arrays. Figure 10 shows the cost of the four steps for the 24 matrices (the x axis is the matrix *id*s) on the four used platforms. Cost of one single SpMV operation is used for normalizing format conversion cost on each platform. We can see that the conversion cost can be on average as low as the overhead of a few SpMV operations on the two GPUs. On the two x86 platforms, the conversion time is longer (up to cost of around 10–20 SpMV operations). The reason is that the conversion code is manually SIMDized using CUDA or OpenCL on GPUs, but only auto-parallelized by OpenMP on x86 processors.



(a) The CPU. (b) The nVidia GPU.

(c) The AMD GPU. (d) The Xeon Phi.

Figure 10: The normalized format conversion cost.

5.6 Iteration-Based Scenarios

Since both the preprocessing (i.e., format conversion from a basic format) time and the SpMV time are important for real-world applications, we have designed an iteration-based benchmark. This benchmark measures the overall performance of a solver with n iterations. We assume the input matrix is already stored in the CSR format. So the overall cost of using the CSR format for the scenarios is nT^{csr}_{spmv}, where T^{csr}_{spmv} is execution time of one CSR-based SpMV operation. For a new format, the overall cost is $T^{new}_{pre} + nT^{new}_{spmv}$, where T^{new}_{pre} is preprocessing time and the T^{new}_{spmv} is one SpMV time using the new format. Thus we can calculate speedup of a new format over the CSR format in the scenarios, through $(nT^{csr}_{spmv})/(T^{new}_{pre} + nT^{new}_{spmv})$.

Table 3 shows the new formats' preprocessing cost (i.e., $T^{new}_{pre}/T^{new}_{spmv}$) and their speedups over the CSR format in the iteration-based scenarios when $n = 50$ and $n = 500$. The emboldened font in the table shows the highest positive speedups on each platform. The compared baseline is the fastest CSR-based SpMV implementation (i.e., Intel MKL,

nVidia cuSPARSE/CUSP, CSR-vector from CUSP, and Intel MKL, respectively) on each platform. We can see that because of the very low preprocessing overhead, the CSR5 can further outperform the previous methods when doing 50 iterations and 500 iterations. Although two GPU methods, the ACSR format and the CSR-Adaptive approach, in general have shorter preprocessing time, they suffer from lower SpMV performance and thus cannot obtain the best speedups. On all platforms, the CSR5 always achieves the highest overall speedups. Moreover, the CSR5 is the only format that obtains higher performance than the CSR format when only 50 iterations are required.

6. RELATED WORK

A great deal of work has been published on accelerating the SpMV operation. The **block-based sparse matrix construction** has received most attention [2, 6, 7, 10, 25, 31, 34] because of two main reasons: (1) sparse matrices generated by some real-world problems (e.g., finite element discretization) naturally have the block sub-structures, and (2) off-chip load operations may be decreased by using the block indices instead of the entry indices. However, for many matrices that do not exhibit a natural block structure, trying to extract the block information is time consuming and has limited effects.

On the other hand, the **hybrid formats** [4, 29], such as HYB, have been designed for irregular matrices. However, higher kernel launch overhead and invalidated cache among kernel launches tend to decrease their overall performance. Moreover, it is hard to guarantee that every sub-matrix can saturate the whole device. In addition, some relatively simple operations such as solving triangular systems become complex while the input matrix is stored in two or more separate parts.

The recent **row block methods** showed good performance either for regular matrices [17] or for irregular matrices [1], but not for both. In contrast, the CSR5 can deliver higher throughput both for regular matrices and for irregular matrices.

The **segmented sum methods** have been used in two recently published papers [30, 34] for the SpMV on either GPUs or Xeon Phi. However, both of them need to store the matrix in COO-like formats to utilize the segmented sum. In contrast, the CSR5 format saves useful row index information in a compact way, and thus can be more efficient both for the format conversion and for the SpMV operation.

Sedaghati et al. [27] constructed machine learning classifiers for **automatic selection of the best format** for a given sparse matrix on a target GPU. The CSR5 format described in this work can further simplify such a selection process because it is insensitive to the sparsity structure of the input sparse matrix.

Moreover, to the best of our knowledge, the CSR5 is the only format that supports high throughput **cross-platform SpMV** on CPUs, nVidia GPUs, AMD GPUs and Xeon Phi at the same time. This advantage may simplify the development of scientific software for processors with massive on-chip parallelism.

7. CONCLUSIONS

In this paper, we proposed the CSR5 format for efficient cross-platform SpMV on CPUs, GPUs and Xeon Phi. The format conversion from the CSR to the CSR5 was very fast because of the format's insensitivity to sparsity structure of the input matrix. The CSR5-based SpMV was implemented by a redesigned segmented sum algorithm with higher SIMD utilization compared to the classic methods. The experimental results showed that the CSR5 delivered high throughput both in the isolated SpMV tests and in the iteration-based scenarios.

8. ACKNOWLEDGMENTS

The authors would like to thank James Avery (KU), Huamin Ren (AAU), Wenliang Wang (BOC), Jianbin Fang (NUDT), Joseph L. Greathouse (AMD), Shuai Che (AMD), Ruipeng Li (UMN), Anders Logg (Chalmers and GU), and our anonymous reviewers for their insightful feedback. We thank Klaus Birkelund Jensen (KU), Hans Henrik Happe (KU) and Rune Kildetoft (KU) for access to the Intel Xeon and Xeon Phi machines. We thank Bo Shen (Inspur) for helpful discussion about Xeon Phi programming. We also thank Arash Ashari (OSU), Intel MKL team, Joseph L. Greathouse (AMD) and Mayank Daga (AMD) for sharing source code, libraries or implementation details of their SpMV algorithms with us. Finally, we thank the PPoPP '15 reviewers for their valuable suggestions and comments.

9. REFERENCES

[1] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan. Fast Sparse Matrix-Vector Multiplication on GPUs for Graph Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 781–792, 2014.

[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*, ICS '14, pages 273–282, 2014.

[3] M. M. Baskaran and R. Bordawekar. Optimizing Sparse Matrix-Vector Multiplication on GPUs. Technical Report RC24704, IBM, 2008.

[4] 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, pages 18:1–18:11, 2009.

[5] G. E. Blelloch, M. A. Heroux, and M. Zagha. Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors. Technical Report CMU-CS-93-173, Carnegie Mellon University, 1993.

[6] A. Buluç, S. Williams, L. Oliker, and J. Demmel. Reduced-Bandwidth Multithreaded Algorithms for Sparse Matrix-Vector Multiplication. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 721–733, 2011.

[7] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson. Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in*

Algorithms and Architectures, SPAA '09, pages 233–244, 2009.

[8] J.-H. Byun, R. Lin, J. W. Demmel, and K. A. Yelick. pOSKI: Parallel Optimized Sparse Kernel Interface Library. Technical report, University of California, Berkeley, 2012.

[9] S. Chatterjee, G. Blelloch, and M. Zagha. Scan Primitives for Vector Computers. In *Supercomputing '90., Proceedings of*, pages 666–675, 1990.

[10] 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, pages 115–126, 2010.

[11] S. Dalton and N. Bell. *CUSP : A C++ Templated Sparse Matrix Library*, 2014.

[12] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, 2011.

[13] Y. S. Deng, B. D. Wang, and S. Mu. Taming Irregular EDA Applications on GPUs. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, ICCAD '09, pages 539–546, 2009.

[14] J. Dongarra and M. A. Heroux. Toward a New Metric for Ranking High Performance Computing Systems. Technical Report SAND2013-4744, Sandia National Laboratories, 2013.

[15] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli. Fast Scan Algorithms on Graphics Processors. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, pages 205–213, 2008.

[16] M. Garland. Sparse Matrix Computations on Manycore GPU's. In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, pages 2–6, 2008.

[17] J. L. Greathouse and M. Daga. Efficient Sparse Matrix-Vector Multiplication on GPUs using the CSR Storage Format. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 769–780, 2014.

[18] D. Guo and W. Gropp. Adaptive Thread Distributions for SpMV on a GPU. In *Proceedings of the Extreme Scaling Workshop*, pages 2:1–2:5, 2012.

[19] K. Kourtis, G. Goumas, and N. Koziris. Exploiting Compression Opportunities to Improve SpMxV Performance on Shared Memory Systems. *ACM Trans. Archit. Code Optim.*, 7(3):16:1–16:31, 2010.

[20] 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(5):C401–C423, 2014.

[21] J. Li, G. Tan, M. Chen, and N. Sun. SMAT: An Input Adaptive Auto-Tuner for Sparse Matrix-Vector Multiplication. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 117–126, 2013.

[22] R. Li and Y. Saad. GPU-accelerated preconditioned iterative linear solvers. *The Journal of Supercomputing*, 63(2):443–466, 2013.

[23] W. Liu and B. Vinter. An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 370–381, 2014.

[24] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey. Efficient Sparse Matrix-Vector Multiplication on x86-based Many-Core Processors. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 273–282, 2013.

[25] A. Pinar and M. Heath. Improving Performance of Sparse Matrix-Vector Multiplication. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, SC '99, 1999.

[26] K. Rupp, F. Rudolf, and J. Weinbub. ViennaCL - A High Level Linear Algebra Library for GPUs and Multi-Core CPUs. In *GPUScA*, pages 51–56, 2010.

[27] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan. Automatic Selection of Sparse Matrix Representation on GPUs. In *Proceedings of the 29th ACM International Conference on Supercomputing*, ICS '15, 2015.

[28] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan Primitives for GPU Computing. In *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '07, pages 97–106, 2007.

[29] B.-Y. Su and K. Keutzer. clSpMV: A Cross-Platform OpenCL SpMV Framework on GPUs. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 353–364, 2012.

[30] W. T. Tang, R. Zhao, M. Lu, Y. Liang, H. P. Huynh, X. Li, and R. S. M. Goh. Optimizing and Auto-Tuning Scale-Free Sparse Matrix-Vector Multiplication on Intel Xeon Phi. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 136–145, 2015.

[31] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A Library of Automatically Tuned Sparse Matrix Kernels. *Journal of Physics: Conference Series*, 16(1):521–530, 2005.

[32] R. W. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, University of California, Berkeley, dec 2003.

[33] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms. *Parallel Computing*, 35(3):178–194, 2009.

[34] S. Yan, C. Li, Y. Zhang, and H. Zhou. yaSpMV: Yet Another SpMV Framework on GPUs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14,