

# Merge-based Parallel Sparse Matrix-Vector Multiplication

Duane Merrill  
NVIDIA Corporation  
Santa Clara, CA  
dumerrill@nvidia.com

Michael Garland  
NVIDIA Corporation  
Santa Clara, CA  
mgarland@nvidia.com

**Abstract**—We present a strictly balanced method for the parallel computation of sparse matrix-vector products (SpMV). Our algorithm operates directly upon the Compressed Sparse Row (CSR) sparse matrix format without preprocessing, inspection, reformatting, or supplemental encoding. Regardless of nonzero structure, our equitable 2D merge-based decomposition tightly bounds the workload assigned to each processing element. Furthermore, our technique is suitable for recursively partitioning CSR datasets themselves into multi-scale, distributed, NUMA, and GPU environments that are constrained by fixed-size local memories.

We evaluate our method on both CPU and GPU microarchitectures across a very large corpus of diverse sparse matrix datasets. We show that traditional CsrMV methods are inconsistent performers, often subject to order-of-magnitude performance variation across similarly-sized datasets. In comparison, our method provides predictable performance that is substantially uncorrelated to the distribution of nonzeros among rows and broadly improves upon that of current CsrMV methods.

**Keywords**— *SpMV, sparse matrix, parallel merge, merge-path, many-core, GPU, linear algebra*

## I. INTRODUCTION

High performance algorithms for sparse linear algebra are important within many application domains, including computational science, graph analytics, and machine learning. The sparse matrix-vector product (SpMV) is of particular importance for solving sparse linear systems, eigenvalue systems, Krylov subspace methods, and other similar problems. When generalized to other algebraic semi-rings, it is also an important building block for large-scale combinatorial graph algorithms [1]–[3]. In its simplest form, a SpMV operation computes  $y = Ax$  where the matrix  $A$  is sparse and vectors  $x$  and  $y$  are dense.

Patterns of SpMV usage are often highly repetitive within iterative solvers and graph computations, making it a performance-critical component whose overhead can dominate overall application running time. Achieving good performance on today's parallel architectures requires complementary strategies for workload decomposition and matrix storage formatting that provide both uniform processor utilization and efficient use of memory bandwidth, regardless of matrix nonzero structure [4], [5]. These design objectives have

inspired many custom matrix formats and encodings that exploit both the structural properties of a given matrix and the organization of the underlying machine architecture. In fact, more than sixty specialized SpMV algorithms and sparse matrix formats have been proposed for GPU processors alone, which exemplify the current trend of increased parallelism in high performance computer architecture [6].

However, improving SpMV performance with innovative matrix formatting comes at a significant practical cost. Applications rarely maintain sparse matrices in custom encodings, instead preferring general-purpose encodings such as the Compressed Sparse Row (CSR) format for in-memory representation (Fig. 1). The CSR encoding is free of architecture-specific blocking, reordering, annotations, etc. that would otherwise hinder portability. Consequently, specialized or supplementary formats ultimately burden the application with both (1) preprocessing time for inspection and formatting, which may be tens of thousands of times greater than the SpMV operation itself; and (2) excess storage overhead because the original CSR matrix is likely required by other routines and cannot be discarded.

An ideal CsrMV would deliver consistently high performance without format conversion or augmentation. As shown in Algorithm 1, CsrMV is principally a row-wise summation of partial matrix-vector dot-products. The independence of rows and the associativity of addition provide ample opportunities for parallelism. However, contemporary CsrMV strategies that attempt to parallelize these loops independently are subject to performance degradation arising from irregular row lengths and/or wide aspect ratios [7]–[11]. Despite various heuristics for constraining imbalance, they often underperform for small-world or scale-free datasets having a minority of rows that are many orders of magnitude longer than average. Furthermore, such underutilization often scales with increased processor parallelism. This is particularly true of GPUs, where the negative impact of a few long-running threads can be drastic.

To illustrate the effects of row-based workload imbalance, consider the three similarly-sized sparse matrices in Table 1. All three comprise approximately 3M nonzeros, yet have quite different row-length variations. As variation increases, the row-based CsrMV implementations within Intel MKL [10] and NVIDIA cuSPARSE [11] are progressively unable to map their workloads equitably across parallel threads. This nonlinear

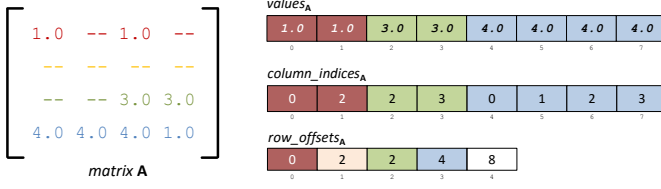


Fig. 1. Example three-array CSR sparse matrix encoding

#### ALGORITHM 1: The sequential CsrMV algorithm

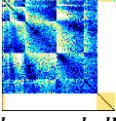
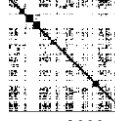
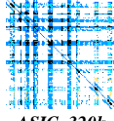
**Input:** CSR matrix  $A$ , dense vector  $x$

**Output:** Dense vector  $y$  such that  $y \leftarrow Ax$

```

1  for (int row = 0; row < A.n; ++row)
2  {
3      y[row] = 0.0;
4      for (int nz = A.row_offsets[row];
5           nz < A.row_offsets[row + 1];
6           ++nz)
7      {
8          y[row] += A.values[nz] * x[A.column_indices[nz]];
9      }
10 }
```

**TABLE 1:** Relative consistency of double-precision CsrMV performance among similarly-sized matrices from the University of Florida Sparse Matrix Collection [7], evaluated using two Intel Xeon E5-2690 CPU processors (24-core each) and one NVIDIA Tesla K40 GPU

		 <i>thermomech_dK</i> (temperature deformation)	 <i>cnnr-2000</i> (Web connectivity)	 <i>ASIC_320k</i> (circuit simulation)
<b>Number of nonzeros (nnz)</b>		2,846,228	3,216,152	2,635,364
<b>Row-length coefficient of variation</b>		0.10	2.1	61.4
<b>CPU x2</b>	<i>MKL</i> (GFLOPs)	17.9	13.4	11.8
	<i>Merge-based</i> (GFLOPs)	<b>21.2</b>	<b>22.8</b>	<b>23.2</b>
<b>GPU x1</b>	<i>cuSPARSE</i> (GFLOPs)	12.4	5.9	0.12
	<i>Merge-based</i> (GFLOPs)	<b>15.5</b>	<b>16.7</b>	<b>14.1</b>

performance response makes it difficult to establish reasonable performance expectations.

Our CsrMV does not suffer from such imbalance because it equitably splits the aggregate work performed by the loop nest as a whole. To do so, we adapt a simple parallelization strategy originally developed for efficient merging of sorted sequences [12]–[14]. The central idea is to frame the parallel CsrMV decomposition as a logical merger of two lists:

- The row descriptors (e.g., the CSR row-offsets)
- The natural numbers  $\mathbb{N}$  (e.g., the indices of the CSR nonzero values).

Individual processing elements are assigned equal-sized shares of this logical merger, with each processing element performing a two-dimensional search to isolate the corresponding region within each list that would comprise its share. The regions can then be treated as independent CsrMV subproblems and consumed directly from the CSR data structures using the sequential method of Algorithm 1. This equitable multi-partitioning ensures that no single processing element can be overwhelmed by assignment to (a) arbitrarily-long rows or (b) an arbitrarily-large number of zero-length rows. Furthermore, this method requires no preprocessing overhead, reordering, or extensions of the CSR matrix format with additional data structures. Matrices can be presented to our implementation directly as constructed by the application.

Our merge-based decomposition is also useful for recursively partitioning CSR datasets themselves within multi-scale and/or distributed memories (e.g., NUMA CPU architecture, hierarchical GPU architecture, and multi-node HPC systems).

Local storage allocation is simplified because our method guarantees equitable storage partitioning.

Whereas prior SpMV investigations have studied performance on a few dozen select datasets, we perform our evaluation using the *entire* University of Florida Sparse Matrix Collection (currently 4,201 non-trivial datasets). This experimental corpus covers a wide gamut of sparsity patterns, ranging from well-structured matrices (typical of discretized geometric manifolds) to power-law matrices (typical of network graphs). For shared-memory CPU and GPU architectures, we show MKL and cuSPARSE CsrMV to commonly exhibit performance discrepancies of one or more orders of magnitude among similarly-sized matrices. We also show that our performance is substantially uncorrelated to irregular row-lengths and highly correlated to problem size. We typically match or exceed the performance of MKL and cuSPARSE, achieving 1.6x and 1.1x respective average speedup for medium and large-sized datasets, and up to 198x for highly irregular datasets. Our method also performs favorably with respect to specialized formats (CSB [15], HYB [7]), even those that leverage expensive per-processor and per-matrix auto-tuning (pOSKI [16], yaSpMV [17]).

## II. BACKGROUND

### A. General-purpose sparse matrix formats

For a given  $n \times m$  matrix  $A$ , sparse formats aim to store only the nonzero entries of  $A$ . This can result in substantial savings, as the number of nonzero entries  $nnz$  in many datasets is  $O(m+n)$ .

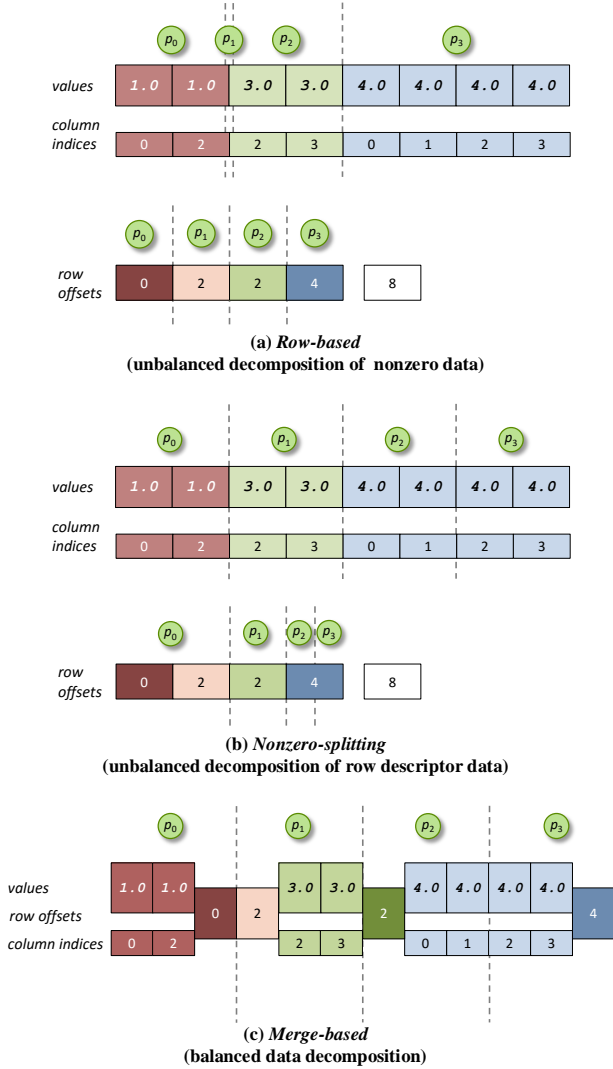


Fig. 2. Basic CsrMV parallel decomposition strategies across four threads

The simplest general-purpose representation is the Coordinate (COO) format, which records each nonzero as an index-value triple  $(i, j, A_{ij})$ . COO lends itself to SpMV parallelizations based upon segmented scan primitives [18]–[20] which provide strong guarantees of workload balance and stable performance across all nonzero distributions [7]. However, pure CooMV implementations do not often achieve high levels of performance, in part because the format has relatively high storage overhead.

The CSR format eliminates COO row-index repetition by storing the nonzero values and column indices in row-major order, and building a separate *row-offsets* array such that the entries for row  $i$  in the other arrays occupy the half-open interval  $[\text{row-offsets}_i, \text{row-offsets}_{i+1})$ . CSR is perhaps the most commonplace general-purpose format for in-memory storage.

### B. Specialized sparse matrix formats

Due to its importance as an algorithmic building block, there is a long history of SpMV performance optimization under varying assumptions of matrix sparsity structure [4]. A

primary goal of many specialized formats is to regularize computation and memory accesses, often through padding and/or matrix reorganization. For example, the ELL format pads all rows to be the same length [7], but can degenerate into dense storage in the presence of singularly large rows.

Reorganization strategies attempt to map similarly-sized rows onto co-scheduled thread groups. Bell and Garland investigated a packetized (PKT) format that partitions the matrix and then lays out the entries of each piece in a way that balances work across threads [21]. Ashari *et al.* suggested binning the rows by length so that rows of similar length can be processed together [9]. The Sliced ELL format [22] can provide similar benefits, as it also bins rows by length. Although these heuristics improve balance for many scenarios, their coarseness can still lead to processor underutilization.

Many specialized formats also aim to reduce memory I/O via index compression. *Blocking* is a common extension of the storage formats above, where only a single index is used to locate a small, dense block of matrix entries. (Our merge-based CsrMV is compatible with blocking, as the CSR structures are used in the same way to refer to sparse blocks instead of individual sparse nonzeros.) Other sophisticated compression schemes attempt to optimize the bit-encoding of the matrix, often at the expense of significant formatting overhead [23], [24]. The yaSpMV BCCOO format is a variation of block-compressed COO that uses bit flags to store the row indices in line with column indices [17].

Hybrid and multi-level formats are also commonplace. The matrix can be partitioned into separate regions, each of which may be stored in a different format. The pOSKI autotuning framework explores a wide gamut of multi-level blocking schemes [16]. The compressed sparse block (CSB) format is a nested COO-of-COO representation where tuples at both levels are kept in a Morton Z-order [15]. The HYB format combines an ELL portion with a COO portion [7]. Su *et al.* demonstrated an auto-tuning framework based on the Cocktail format that would explore arbitrary hybridizations [25]. For extended reading, Williams *et al.* explore a range of techniques for multicore CPUs [5], [26], and Filippone *et al.* provide a comprehensive survey of SpMV methods for GPUs [6].

### C. Contemporary CsrMV parallelization strategies

CsrMV implementations typically adhere to one of two general parallelization strategies: (a) a *row splitting* variation of row-based work distribution; or (b) *nonzero splitting*, an equitable parallel partitioning of nonzero data.

**Row splitting.** This practice assigns regularized sections of long rows to multiple processors in order to limit the number of number data items assigned to each processor. The partial sums from related sub-rows can be aggregated in a subsequent pass. The differences in length between rows that are smaller than the splitting size can still contribute load imbalance between threads. The splitting of long rows can be done statically via a preprocessing step that encodes the dataset into an alternative format, or dynamically using a work-sharing runtime. The dynamic variant requires runtime task distribution, a behavior likely to incur processor contention and limited scalability on massively parallel systems.

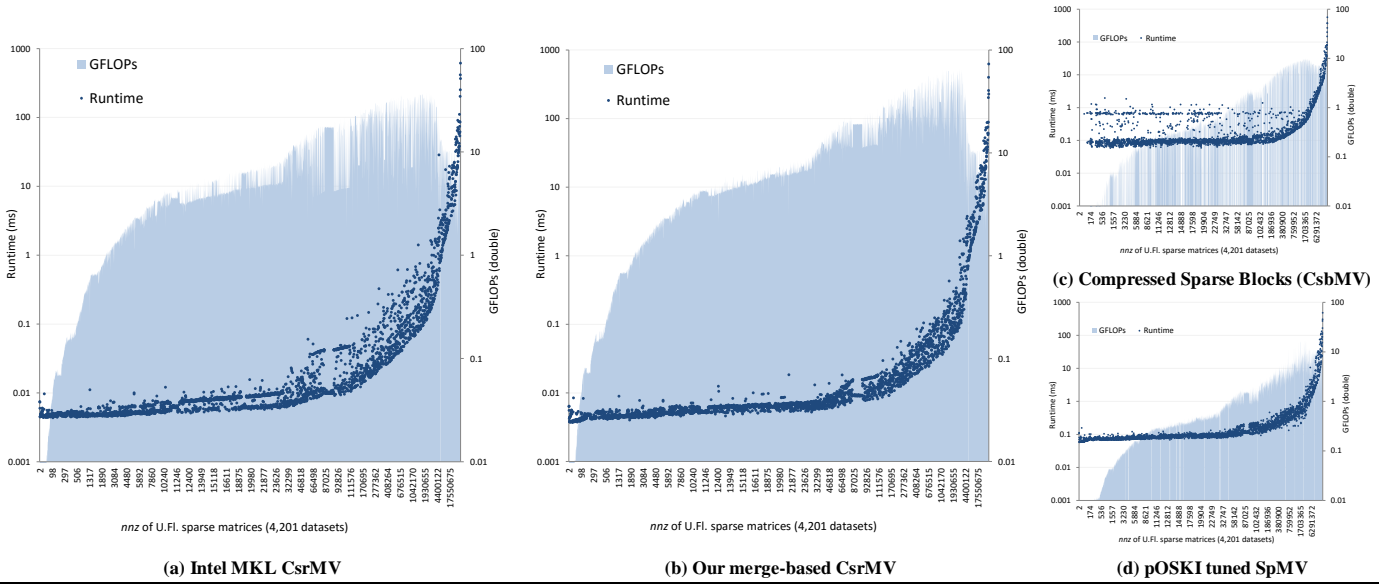


Fig. 3. CPU SpMV performance-consistency across the U.Fl. Sparse Matrix Collection, arranged by number of nonzero elements (Intel Xeon E5-2690v2 x 2)

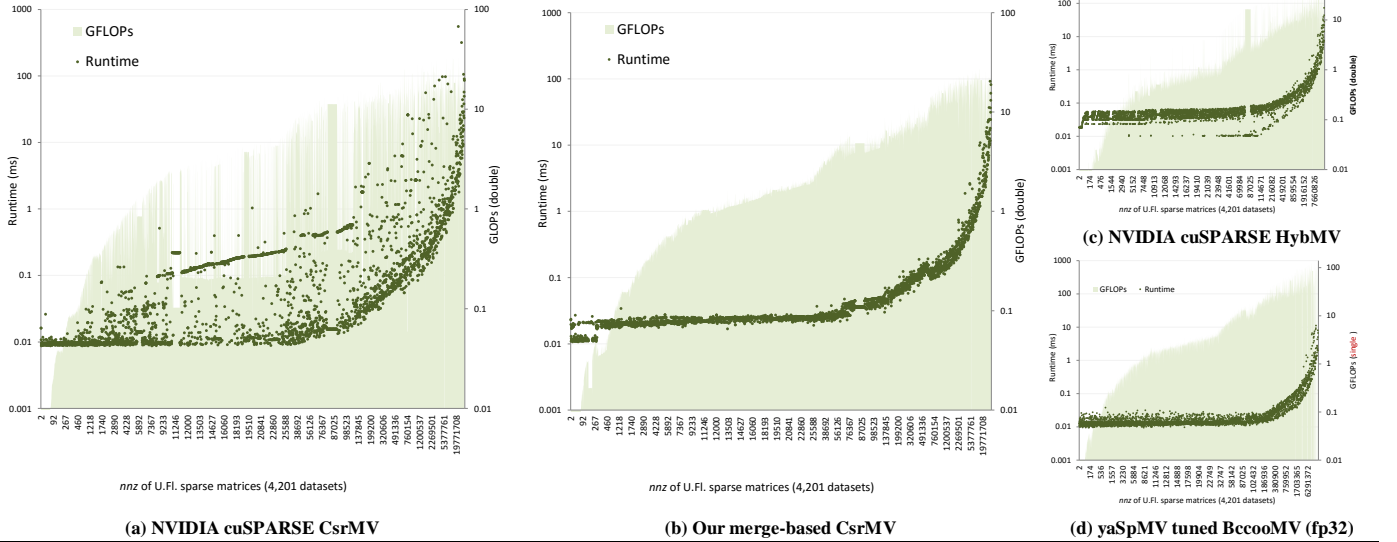


Fig. 4. GPU SpMV performance-consistency across U.Fl. Sparse Matrix Collection, arranged by number of nonzero elements (NVIDIA K40)

Vectorization is a common variant of row-splitting in which a group of threads is assigned to process each row. Vectorization is typical of GPU-based implementations such as cuSPARSE CsrMV [11]. Although bandwidth coalescing is improved by strip-mined access patterns, there is potential for significant underutilization in the form of inactive SIMD lanes when row-lengths do not align with SIMD widths [7]. Recent work by Greathouse and Daga [8] and Ashari *et al.* [9] adaptively vectorize based on row length, thus avoiding the problem that vectorized CSR kernels perform poorly on short rows. However, these methods require additional storage and preprocessing time to augment CSR matrices with supplemental data structures that capture thread assignment.

**Nonzero splitting.** An alternative to row-based parallelization is to assign an equal share of the nonzero data (i.e., the *values* and *column-indices* arrays) to each processor.

Processors then determine to which row(s) their data items belong by searching within the *row-offsets* array. As each processor consumes its section of nonzeros, it must track its progress through the *row-offsets* array. The CsrMV parallelizations from Dalton *et al.* [27] and Baxter [28] perform this searching as an offline preprocessing step, storing processor-specific waypoints within supplemental data structures. (Despite their nomenclature, these implementations are not merge-based, but rather examples of nonzero-splitting.)

As illustrated in Fig. 2b, imbalance is still possible because some processors may consume arbitrarily more row offsets than others in the presence of empty rows. Although such *hypersparse* [29] matrices are fairly rare in numerical algebra (nonsingular square matrices must have  $nnz \geq n$ ), they occur frequently in computations on graphs. For example, more than 20% of web-crawl vertices (*webbase-2001*), 12% of Amazon

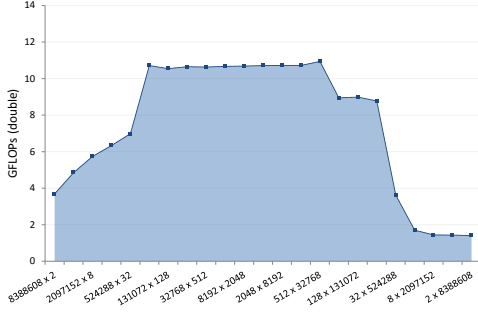


Fig. 5. Intel MKL CsrMV performance as a function of aspect ratio (Xeon E5-2690v2 x2). As row-count falls below the width of the machine (40 threads), performance falls by up to 8x.

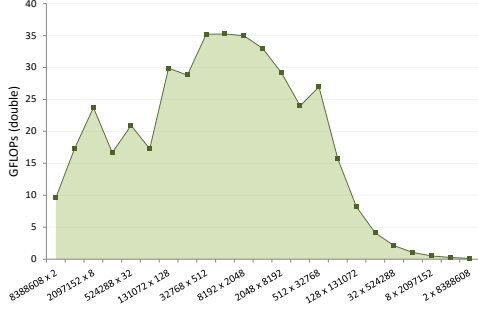


Fig. 6. NVIDIA cuSPARSE CsrMV performance as a function of aspect ratio (Tesla K40). As row-count falls below the width of the machine (960 warps), performance falls by up to 270x.

vertices (*amazon-2008*), and 50% of vertices within the synthetic RMAT datasets generated for the Graph500 benchmark have zero out-degree [30], [31].

#### D. Illustrations of CsrMV workload imbalance

Fig. 3a and Fig. 4a illustrate underutilization from irregular sparsity structure, presenting MKL and cuSPARSE CsrMV elapsed running time performance across the entire Florida Sparse Matrix Collection. Ideally we would observe a smooth, continuous performance response that is highly correlated to data set size ( $nnz$ ). However, many performance outliers are readily visible (despite being plotted on a log-scale), often running one or two orders of magnitude slower than similarly-sized datasets. This is especially pronounced on the GPU, where parallel efficiency is particularly hindered by workload imbalance.

Furthermore, Fig. 5 and Fig. 6 highlight the performance cliffs that are common to row-based CsrMV when computing on short, wide matrices having many columns and few rows. We use MKL and cuSPARSE to compute CsrMV across different aspect ratios of dense matrices (stored in sparse CSR format), all comprising the same number of nonzero values. Performance drops severely as the number of rows falls below the number of available hardware thread contexts. Although the CSR format is general-purpose, these implementations discriminate by aspect ratio against entire genres of short, wide sparse matrices.

### III. MERGE-BASED CSR MV

Our parallel CsrMV decomposition is similar to the fine-grained *merge-path* method for efficiently merging two sorted lists  $A$  and  $B$  [12]–[14]. The fundamental property of this method is that each processor is provided with an equal share of  $|A|+|B|$  steps, regardless of list sizing and value content.

#### A. Parallel “merge-path”

In general, a merge computation can be viewed as a decision path of length  $|A|+|B|$  in which progressively larger elements are consumed from  $A$  and  $B$ . Fig. 7 illustrates this as a two-dimensional grid in which the elements of  $A$  are arranged along the  $x$ -axis and the elements of  $B$  are arranged along the  $y$ -axis. The decision path begins in the top-left corner and ends in the bottom-right. When traced sequentially, the merge path moves right when consuming the elements from  $A$  and down when consuming from  $B$ . As a consequence, the path coordinates describe a complete schedule of element comparison and consumption across both input sequences. Furthermore, each path coordinate can be linearly indexed by its grid *diagonal*, where diagonals are enumerated from top-left to bottom-right. Per convention, the semantics of merge always prefer items from  $A$  over those from  $B$  when comparing same-valued items. This results in exactly one decision path.

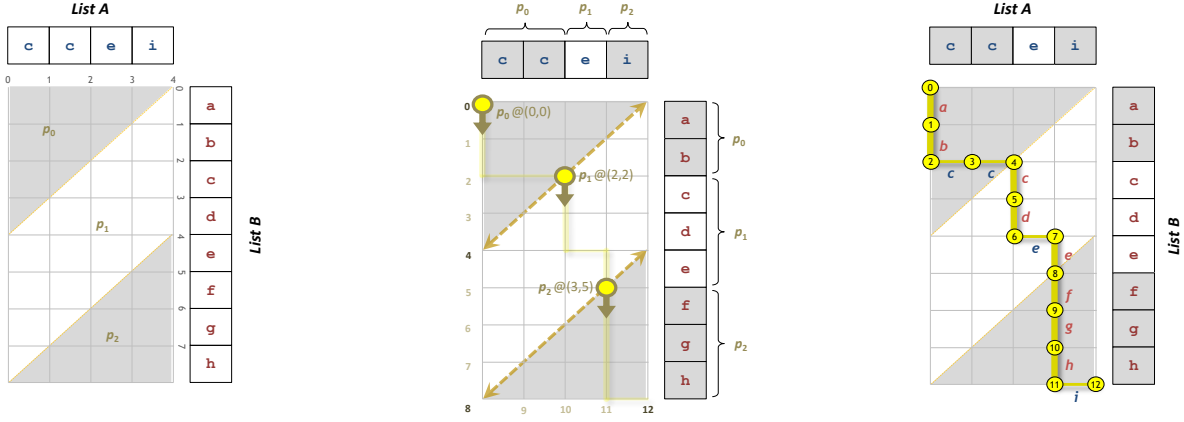
To parallelize across  $p$  threads, the grid is sliced diagonally into  $p$  swaths of equal width, and it is the job of each thread is to establish the route of the decision-path through its swath. The fundamental insight is that any given path coordinate  $(i,j)$  can be found independently using the two-dimensional search procedure presented in Algorithm 3. More specifically, the two elements  $A_i$  and  $B_j$  scheduled to be compared at diagonal  $k$  can be found via constrained binary search along that diagonal: find the first  $(i,j)$  where  $A_i$  is greater than all of the items consumed before  $B_j$ , given that  $i+j=k$ . Each thread need only search the first diagonal in its swath; the remainder of its path segment can be trivially established via sequential comparisons seeded from that starting coordinate.

An important property of this parallelization strategy is that the decision path can be partitioned hierarchically, trivially enabling parallelization across large, multi-scale systems. Assuming the number of processors  $p$  is a finite, constant property of any given level of the underlying machine (unrelated to  $N=|A|+|B|$ ), the total work remains  $O(N)$ .

#### B. Adaptation for CSR SpMV

As illustrated in Fig. 8, we can compute CsrMV using the merge-path decomposition by logically merging the *row-offsets* vector with the sequence of natural numbers  $\mathbb{N}$  used to index the *values* and *column-indices* vectors. We emphasize that this merger is never physically realized, but rather serves to guide the equitable consumption of the CSR matrix. By design, each contiguous vertical section of the decision path corresponds to a row of nonzeros in the CSR sparse matrix. As threads follow the merge-path, they accumulate matrix-vector dot-products when moving downwards. When moving rightwards, threads then flush these accumulated values to the corresponding row output in  $y$  and reset their accumulator. The partial sums from rows that span multiple threads can be aggregated in a subsequent reduce-value-by-key “fix-up” pass. The result is



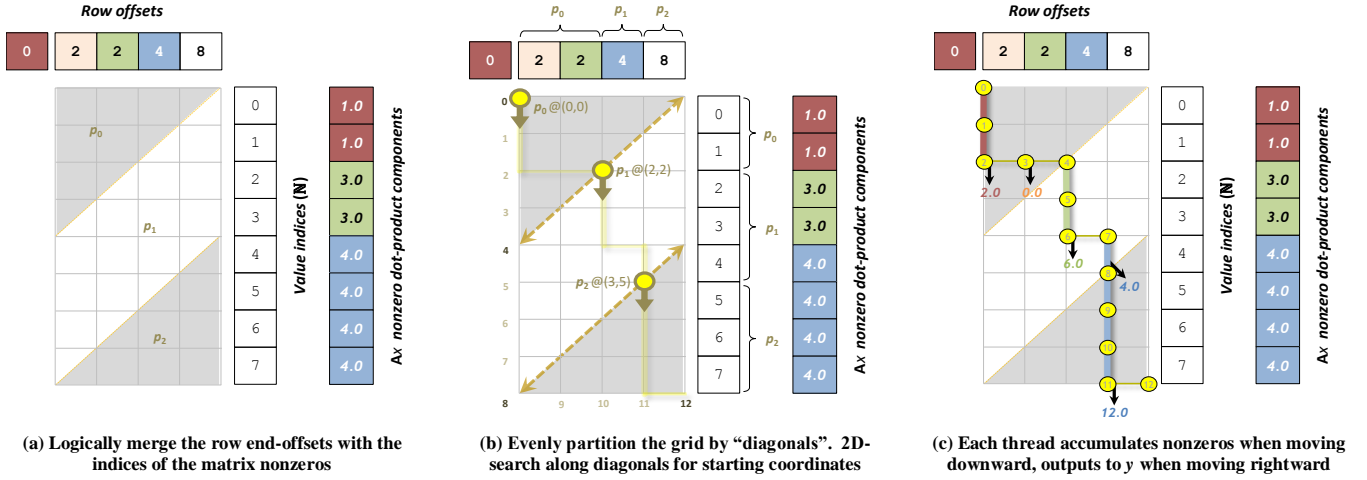


(a) The “merge path” will traverse a 2D grid from top-left to bottom-right

(b) Evenly partition the grid by “diagonals”. 2D-search along diagonals for starting coordinates

(c) Each thread runs the sequential merge algorithm from starting coordinates

Fig. 7. An example *merge-path* visualization for the parallel merger of two sorted lists by three threads. Regardless of array content, the path is twelve items long and threads consume exactly four items each.



(a) Logically merge the row end-offsets with the indices of the matrix nonzeros

(b) Evenly partition the grid by “diagonals”. 2D-search along diagonals for starting coordinates

(c) Each thread accumulates nonzeros when moving downward, outputs to y when moving rightward

Fig. 8. An example merge-based CsrMV visualization for the sparse matrix presented in Fig. 1. Threads consume exactly four CSR items each, where an item is either a nonzero or a row-offset.

that we *always* partition equal amounts of CsrMV work across parallel threads, regardless of matrix structure.

### C. CPU implementation

To illustrate the simplicity of this method, we present our parallel OpenMP C++ merge-based CsrMV implementation in its entirety in Algorithm 2. Lines 3-6 establish the merger lists and merge-path lengths (total and per-thread). Entering the parallel section, lines 16-21 identify the thread’s starting and ending diagonals and then search for the corresponding 2D starting and ending coordinates within the merge grid. (We use a counting iterator to supply the elements of  $\mathbb{N}$ .) In lines 24-32, each thread effectively executes the sequential CsrMV Algorithm 1 along its share of the merge path. Lines 35-36 accumulate any nonzeros for a partial row shared with the next thread. Lines 39-40 save the thread’s running total and row-id for subsequent fix-up. Back in the sequential phase, lines 44-46 update the values in  $y$  for rows that span multiple threads.

To underscore the importance of workload balance, our implementation performs no architecture-specific optimizations other than affixing thread affinity to prevent migration across cores and sockets. We do not explicitly incorporate software pipelining, branch elimination, SIMD intrinsics, advanced pointer arithmetic, prefetching, register-blocking, cache-blocking, TLB-blocking, matrix-blocking, or index compression.

### D. CPU NUMA Optimization

We did, however, explore NUMA opportunities for speedup when consuming datasets too large to fit in aggregate last-level cache. As is typical with most multi-socket CPU systems, each socket on our Xeon e5-2690 platform provisions its own channels of off-chip physical memory into the shared virtual address space. For applications willing to engage in a two-phase “inspector/executor” relationship with our CsrMV implementation, our NUMA-aware variant will perform a one-time merge-path search to identify the sections of the *values*

---

**ALGORITHM 2.** The parallel merge-based CsrMV algorithm (C++ OpenMP)

---

**Input:** Number of parallel threads, CSR matrix  $A$ , dense vector  $x$ **Output:** Dense vector  $y$  such that  $y \leftarrow Ax$ 

---

```
1 void OmpMergeCsrmv(int num_threads, const CsrMatrix& A, double* x, double* y)
2 {
3     int* row_end_offsets = A.row_offsets + 1; // Merge list A: row end-offsets
4     CountingInputIterator<int> nz_indices(0); // Merge list B: Natural numbers (NZ indices)
5     int num_merge_items = A.num_rows + A.num_nonzeros; // Merge path total length
6     int items_per_thread = (num_merge_items + num_threads - 1) / num_threads; // Merge items per thread
7
8     int row_carry_out[num_threads];
9     double value_carry_out[num_threads];
10
11     // Spawn parallel threads
12     #pragma omp parallel for schedule(static) num_threads(num_threads)
13     for (int tid = 0; tid < num_threads; tid++)
14     {
15         // Find starting and ending MergePath coordinates (row-idx, nonzero-idx) for each thread
16         int diagonal = min(items_per_thread * tid, num_merge_items);
17         int diagonal_end = min(diagonal + items_per_thread, num_merge_items);
18         CoordinateT thread_coord = MergePathSearch(diagonal, row_end_offsets, nz_indices,
19             A.num_rows, A.num_nonzeros);
20         CoordinateT thread_coord_end = MergePathSearch(diagonal_end, row_end_offsets, nz_indices,
21             A.num_rows, A.num_nonzeros);
22
23         // Consume merge items, whole rows first
24         double running_total = 0.0;
25         for (; thread_coord.x < thread_coord_end.x; ++thread_coord.x)
26         {
27             for (; thread_coord.y < row_end_offsets[thread_coord.x]; ++thread_coord.y)
28                 running_total += A.values[thread_coord.y] * x[A.column_indices[thread_coord.y]];
29
30             y[thread_coord.x] = running_total;
31             running_total = 0.0;
32         }
33
34         // Consume partial portion of thread's last row
35         for (; thread_coord.y < thread_coord_end.y; ++thread_coord.y)
36             running_total += A.values[thread_coord.y] * x[A.column_indices[thread_coord.y]];
37
38         // Save carry-outs
39         row_carry_out[tid] = thread_coord_end.x;
40         value_carry_out[tid] = running_total;
41     }
42
43     // Carry-out fix-up (rows spanning multiple threads)
44     for (int tid = 0; tid < num_threads - 1; ++tid)
45         if (row_carry_out[tid] < A.num_rows)
46             y[row_carry_out[tid]] += value_carry_out[tid];
47 }
```

---

and *column-indices* arrays that will be consumed by each thread, and then instruct the operating system to migrate those pages onto the NUMA node where those threads will run. The algorithm and CSR data structures remain unchanged through this preprocessing step.

### E. GPU Implementation

We implemented our GPU version in CUDA C++ using a two-level merge-path parallelization to accommodate the hierarchical GPU processor organization [32]. At the coarsest level, the merge path is equitably divided among thread blocks, of which we invoke only as many as needed to fully occupy the GPU’s multiprocessors (a finite constant uncorrelated to problem size). Each thread block then proceeds to consume its share of the merge path in fixed size path-chunks. Path-chunk length is determined by the amount of local storage resources available to each thread block. For example, a path-chunk of 896 items = 128 *threads-per-block* \* 7 *items-per-thread*.

Threads can then locally search for their 2D-path coordinates relative to the thread block’s current path coordinate. The search range of this second-level of path processing is restricted to a fixed-size tile of *chunk-items* x *chunk-items* of the merge grid, so overall work complexity is unaffected. The thread block then copies the corresponding regions of *row-offsets* (list A) and matrix-value dot-products (list B) into local

shared memory with efficient strip-mined, coalesced loads, forming *local-row-offsets* and *local-nonzeros*.

After the sublists are copied to local shared memory, threads independently perform sequential CsrMV, each consuming exactly *items-per-thread*. However, the doubly-nested loop in lines 24-32 of Algorithm 2 is inefficient for SIMD predication. As shown in Algorithm 4, we can restructure the nest as a single loop that runs for *items-per-thread* iterations for all threads.

The two levels of merge-path parallelization require two levels of “fix-up”. At the thread level, we use block-wide reduce-value-by-key primitives from the CUB Library [33]. At the thread block level, we use device-wide reduce-value-by-key primitives, also from the CUB library.

We emphasize that our merge-based method allows us to make maximal utilization of the GPU’s fixed-size shared memory resources, regardless of the ratio between row-offsets and nonzeros consumed during each path-chunk. This is not possible for row-based or nonzero-splitting strategies that are unable to provide tight bounds on local workloads

## IV. EVALUATION

We primarily compare against the CsrMV implementations provided by Intel’s Math Kernel Library v11.3 [10] and

**ALGORITHM 3: 2D merge-path search**

**Input:** Diagonal index, lengths of lists A and B, iterators (pointers) to lists A and B  
**Output:** The 2D coordinate (x,y) of the intersection of the merge decision path with the specified grid diagonal

```

1  CoordinateT MergePathSearch(
2      int diagonal, int a_len, int b_len,
3      AltoratorT a, BIteratorT b)
4  {
5      // Diagonal search range (in x coordinate space)
6      int x_min = max(diagonal - b_len, 0);
7      int x_max = min(diagonal, a_len);
8
9      // 2D binary-search along the diagonal search range
10     while (x_min < x_max) {
11         OffsetT pivot = (x_min + x_max) >> 1;
12         if (a[pivot] <= b[diagonal - pivot - 1]) {
13             // Keep top-right half of diagonal range
14             x_min = pivot + 1;
15         } else {
16             // Keep bottom-left half of diagonal range
17             x_max = pivot;
18         }
19     }
20     return CoordinateT(
21         min(x_min, a_len), // x coordinate in A
22         diagonal - x_min); // y coordinate in B
23 }

```

**ALGORITHM 4: Re-writing the inner loops of Algorithm 2 (lines 24-33) as a single loop**

```

1  ...
2  // Consume exactly items-per-thread merge items
3  double running_total = 0.0;
4  for (int i = 0; i < items_per_thread; ++i)
5  {
6      if (nz_indices[thread_coord.y] <
7          row_end_offsets[thread_coord.x])
8      {
9          // Move down (accumulate)
10         running_total += A.values[thread_coord.y] *
11             x[A.column_indices[thread_coord.y]];
12         ++thread_coord.y;
13     }
14     else
15     {
16         // Move right (output row-total and reset)
17         y[thread_coord.x] = running_total;
18         running_total = 0.0;
19         ++thread_coord.x;
20     }
21 }
22 ...

```

**TABLE 2: SpMV implementations under evaluation**

	<i>Merge-based CsrMV</i>	<i>NUMA merge- based CsrMV</i>	<i>MKL CsrMV</i>	<i>cuSPARSE CsrMV</i>	<i>Compressed Sparse Blocks CsbMV [15], [35]</i>	<i>NVIDIA cuSPARSE HybMV [21]</i>	<i>pOSKI SpMV [16]</i>	<i>yaSpMV BccooMV [17]</i>
<i>Format (decomposition)</i>	CSR (merge)	CSR (merge)	CSR (row-based)	CSR (vectorized row- based)	COO-of-COO (block-row splitting)	ELL + COO (row-based + segmented reduction)	multi-level block- compressed CSR (opaque)	block-compressed COO (segmented reduction)
<i>Supported platforms</i>	CPU, GPU	CPU	CPU	GPU	CPU	GPU	CPU	GPU
<i>Evaluated precision</i>	double, single	double	double	double	double	double	double	single
<i>Autotuning (per matrix)</i>	no	no	no	no	no	no	yes	yes
<i>Avg. preprocessing overhead (per matrix)</i>	-	44x	-	-	-	19x	484x	155,000x

NVIDIA’s cuSPARSE library v7.5 [11]. To highlight the competitiveness of our merge-based CsrMV method in the absolute, we also compare against several non-CSR formats expressly designed to tolerate row-length variation: CSB [15], HYB [7], pOSKI [16], and yaSpMV [17]. The properties of these implementations are further detailed in Table 2. We define *preprocessing overhead* as a one-time inspection, encoding, or tuning activity whose running time is counted separately from any subsequent SpMV computations. We normalize it as the ratio between preprocessing time versus a single SpMV running time. We do not count our merge-based binary searching and fix-up as preprocessing, but rather as part of SpMV running time. However, we do report our NUMA page migration as preprocessing overhead.

Our corpus of sparse test matrices is comprised of the 4,201 non-vector, non-complex matrices currently catalogued by the University of Florida Sparse Matrix Collection [31].

Our evaluation hardware is comprised of a dual-socket NUMA CPU platform with two Intel Xeon E5-2690v2 CPUs and one NVIDIA Tesla K40 GPU. Each CPU is comprised of 10 cores with two-way hyper-threading (40 threads total) and 25MB L3 cache (50MB total). Together, they achieve a Stream Triad [34] score of 77.9 GB/s. The K40 is comprised of 15 SMs capable of concurrently executing 960 warps of 32 threads each (30k threads total), provides 1.5MB of L2 cache

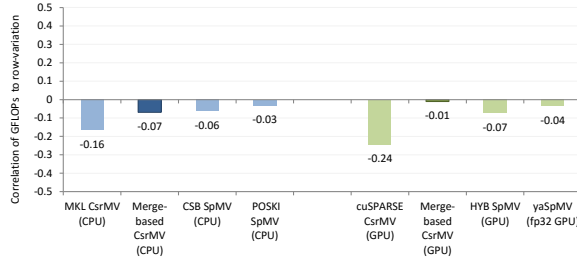
and 720KB of texture cache (2.3MB aggregate cache), and a Stream Triad score of 249 GB/s (ECC off).

**A. Performance consistency**

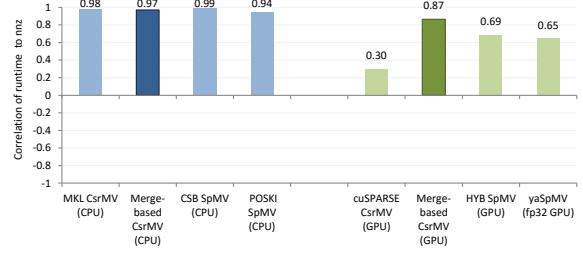
Relative performance consistency is highlighted in the performance landscapes of Fig. 3 and Fig. 4, which plot running time as a function of matrix size measured by nonzero count (*nnz*). The presence of significant outliers is readily apparent for both MKL and cuSPARSE CsrMV parallelizations. Our merge-based strategy visibly conveys a more consistent performance response. These observations are also validated statistically. Fig. 9a presents the degree to which each implementation is able to decouple SpMV throughput from row-length irregularity. Relative to MKL and cuSPARSE CsrMV, our merge-based performance is much less associated with row-length irregularity, and even improves upon the specialized GPU formats.

Furthermore, Fig. 9b presents the degree to which each implementation conforms to the general expectation of a linear correspondence between SpMV running time and matrix size. This general metric incorporates all aspects of the SpMV computation that might affect performance predictability, such as cache hierarchy, nonzero locality, etc. Our GPU performance is significantly more predictable than that of the other



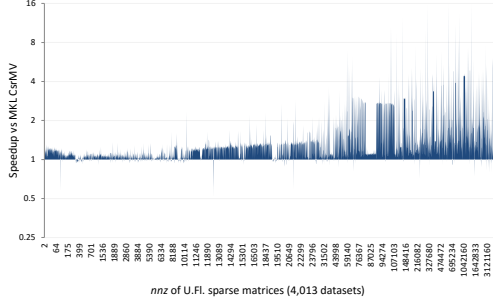


(a) Row-length imperviousness: correlation between GFLOPs throughput vs. row-length variation. (Closer to 0.0 is better.)

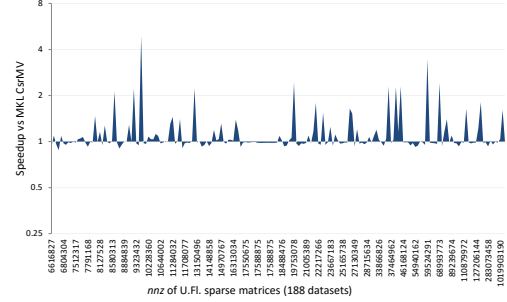


(b) Performance predictability: the correlation between elapsed running time and matrix size  $nnz$  (Closer to 1.0 is better.)

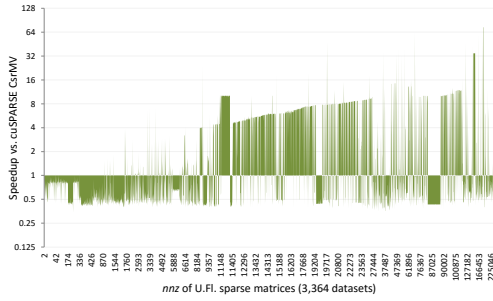
Fig. 9. Metrics of SpMV performance consistency



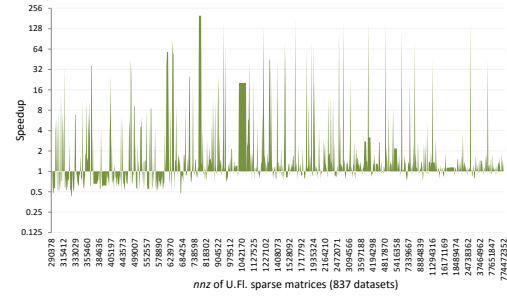
(a) vs. MKL (small matrices < 6M  $nnz$ )



(b) vs. MKL (large matrices > 6M  $nnz$ )



(c) vs. cuSPARSE (small matrices < 300K  $nnz$ )



(d) vs. cuSPARSE (large matrices > 300K  $nnz$ )

		Max speedup	Min speedup	Small-matrix harmonic mean speedup	Large-matrix harmonic mean speedup	Harmonic mean speedup
CPU merge-based CsrMV	vs. MKL CsrMV	15.8	0.51	1.22	1.06	1.21
	vs CSB SpMV	445	0.65	9.21	1.09	6.58
	vs POSKI SpMV	24.4	0.59	11.0	1.10	7.86
CPU NUMA merge-based CsrMV	vs. MKL CsrMV	15.7	0.50	1.25	1.58	1.26
	vs CSB SpMV	464	0.87	9.63	1.66	7.65
	vs POSKI SpMV	24.0	1.07	11.5	1.72	9.14
GPU merge-based CsrMV	vs. cuSPARSE CsrMV	198	0.34	0.79	1.13	0.84
	vs cuSPARSE HybMV	5.96	0.24	1.41	0.96	1.29
	vs yaSpmv BccooMV (fp32)	2.43	0.39	0.78	0.75	0.78

Fig. 10. Relative speedup of merge-based CsrMV

GPU implementations, and we roughly match or exceed the predictability of the other CPU implementations.

We note that our merge-based CsrMV is not entirely free of performance variation among similarly-sized matrices. These irregularities are primarily the result of differing cache responses to different access patterns within the dense vector  $x$ .

### B. Performance throughput

Fig. 10 presents relative SpMV speedup. Our CPU merge-based methods achieve up to 15.8x speedup versus MKL CsrMV on highly irregular matrices. We generally match or

outperform MKL, particularly smaller datasets that fit within aggregate last-level cache ( $nnz < 6M$ ). Our NUMA-aware variant is better able to extend this advantage to large datasets that cannot be captured on-chip, and is 26% faster than MKL on average. Furthermore, our CsrMV is capable of approximating the consistency of the CSB and pOSKI implementations while broadly improving upon their performance. This is particularly true for small problems where the added complexity of these specialized formats is not hidden by the latencies of off-chip memory accesses.

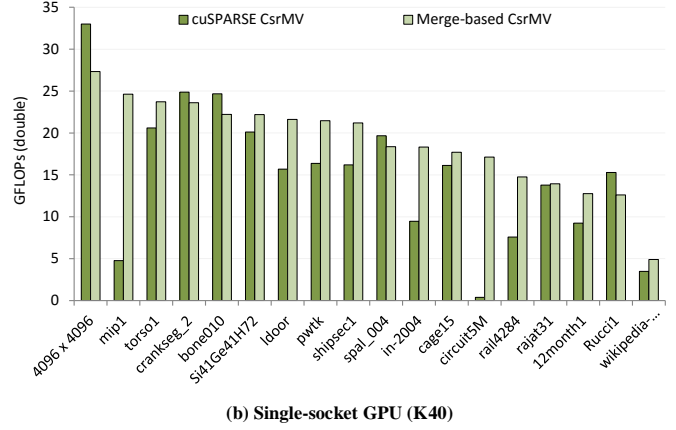
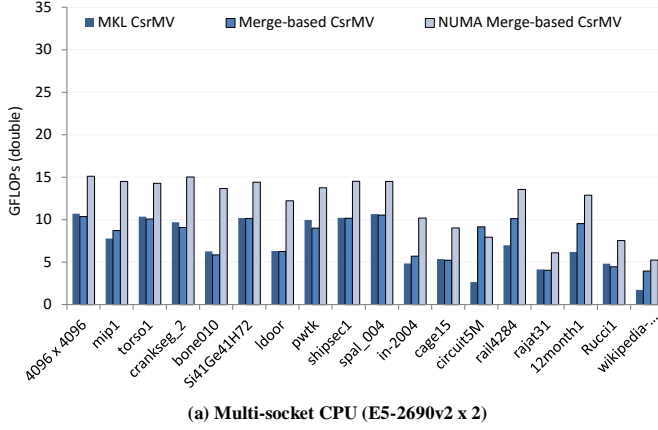


Fig. 11. CsrMV throughput for commonly-evaluated large-matrix datasets

The wider parallelism of the GPU amplifies the detrimental effects of workload imbalance, where our CsrMV achieves up to 198x speedup versus cuSPARSE. For the K40 GPU, the threshold for being captured within aggregate cache is much smaller ( $nnz < 300K$ ). In general, we are 13% faster than cuSPARSE for medium and large-sized size datasets that do not fit in cache.

However, our CsrMV can be up to 50% slower than cuSPARSE for small datasets that align well with its vectorized, row-based method. The high latencies of the GPU can be problematic for our CsrMV on small matrices (towards which the Florida repository is biased) where the basic SpMV workload is not sufficient to overcome our method-specific overheads of (1) merge-path coordinate search and (2) a second kernel for inter-thread fix-up. For comparison, the cuSPARSE CsrMV is able to process more than half of the datasets in under 15 $\mu$ s, whereas the latency of an empty kernel invocation alone is approximately 5 $\mu$ s.

For small problems, the HybMV implementation performs worse than our merge-based CsrMV because its fix-up stage incurs even higher latencies than ours. For larger, non-cacheable datasets, our performance is comparable with the HybMV implementation.

In comparison to yaSpMV BCCOO, our fp32 version of merge-based CsrMV is slower by 22% on average. This exceeds our expectations, given that BCCOO is able to compress away nearly 50% of all indexing data.

### C. Curated datasets

Fig. 11 compares CsrMV performance across commonly-evaluated, large-sized matrices [26], [35]. For the CPU, these matrices are large enough to exceed aggregate last-level cache and comprise substantially more rows than hardware threads. Although the row-based MKL CsrMV can benefit from the inherent workload balancing aspects of oversubscription, our merge-based CsrMV still demonstrates substantial speedup for ultra-irregular matrices (*Wikipedia* and *Circuit5M*). Furthermore, our NUMA-aware CsrMV performs well in this regime because matrix data is not being transferred across CPU sockets. Compared to MKL and our standard merge-based

CsrMV, the NUMA benefit provides average speedups of 1.7x and 1.4x, respectively.

For the GPU, the performance benefits of balanced computation typically outweigh the merge-specific overheads for searching and fix-up, and we achieve speedups of up to 46x (for *Circuit5M*) and a harmonic mean speedup of 1.26x. However, cuSPARSE CsrMV performs marginally better for select matrices having row-lengths that align well with the architecture’s SIMD width (*dense4k*, *bone010*, *Rucci1*).

Finally, in comparing our GPU CsrMV versus our regular and NUMA-optimized CPU implementations, the single K40 outperforms our two-socket Xeon system on average by 2.2x and 1.5x, respectively. (These speedups are lower than the 3.2x bandwidth advantage of the K40 GPU because the larger CPU cache hierarchy is better suited for capturing accesses to the dense vector  $x$ .)

## V. CONCLUSION

As modern processors continue to exhibit wider parallelism, workload imbalance can quickly become the high-order performance limiter for segmented computations such as CSR SpMV. To make matters worse, data-dependent performance degradations often pose significant challenges for applications that require predictable performance response.

In this work, we have adapted merge-based parallel decomposition for computing a well-balanced SpMV directly on CSR matrices without offline analysis, format conversion, or the construction of side-band data. This decomposition is particularly useful for bounding workloads across multi-scale processors and systems with fixed-size local memories. To the best of our knowledge, no prior work achieves these goals.

Furthermore, we have conducted the broadest SpMV evaluation to date to demonstrate the practical shortcomings of exiting CsrMV implementations on real-world data. Our study reveals that contemporary CsrMV methods are inconsistent performers, whereas the performance response of our method is substantially impervious to row-length irregularity. The importance of workload balance is further underscored by the simplicity of our implementation. Even in the absence of architecture-specific optimization strategies, our method is

generally capable of matching or exceeding the performance of contemporary CsrMV implementations.

Finally, our merge-based decomposition is orthogonal to (and yet supportive of) bandwidth optimization strategies such as index compression, blocking, and relabeling. In fact, the elimination of workload imbalance from our method is likely to amplify the benefits of these techniques.

## REFERENCES

- [1] J. Kepner, D. A. Bader, A. Buluç, J. R. Gilbert, T. G. Mattson, and H. Meyerhenke, “Graphs, Matrices, and the GraphBLAS: Seven Good Reasons,” *CoRR*, vol. abs/1504.01039, 2015.
- [2] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, 2011.
- [3] M. Mohri, “Semiring Frameworks and Algorithms for Shortest-distance Problems,” *J. Autom. Lang. Comb.*, vol. 7, no. 3, pp. 321–350, Jan. 2002.
- [4] R. W. Vuduc, “Automatic Performance Tuning of Sparse Matrix Kernels,” Ph.D. Dissertation, University of California, Berkeley, 2003.
- [5] S. Williams, N. Bell, J. W. Choi, M. Garland, L. Oliker, and R. Vuduc, “Sparse Matrix-Vector Multiplication on Multicore and Accelerators,” in *Scientific Computing with Multicore and Accelerators*, J. Kurzak, D. A. Bader, and J. Dongarra, Eds. Taylor & Francis, 2011, pp. 83–109.
- [6] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo, “Sparse Matrix-Vector Multiplication on GPGPUs,” *ACM Trans. Math. Softw.*, To Appear.
- [7] 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*, New York, NY, USA, 2009, pp. 18:1–18:11.
- [8] 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*, Piscataway, NJ, USA, 2014, pp. 769–780.
- [9] 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*, Piscataway, NJ, USA, 2014, pp. 781–792.
- [10] *Intel Math Kernel Library (MKL) v11.3*. Intel Corporation, 2015.
- [11] *NVIDIA cuSPARSE v7.5*. NVIDIA Corporation, 2013.
- [12] S. Odeh, O. Green, Z. Mwassi, O. Shmueli, and Y. Birk, “Merge Path - Parallel Merging Made Simple,” in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, Washington, DC, USA, 2012, pp. 1611–1618.
- [13] S. Baxter and D. Merrill, “Efficient Merge, Search, and Set Operations on GPUs,” Mar-2013. [Online]. Available: <http://on-demand.gputechconf.com/gtc/2013/presentations/S3414-Efficient-Merge-Search-Set-Operations.pdf>.
- [14] N. Deo, A. Jain, and M. Medidi, “An optimal parallel algorithm for merging using multiselection,” *Inf. Process. Lett.*, vol. 50, no. 2, pp. 81–87, Apr. 1994.
- [15] 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 *Proc. SPAA*, Calgary, Canada, 2009.
- [16] A. Jain, “poSKI: An Extensible Autotuning Framework to Perform Optimized SpMVs on Multicore Architectures,” Master’s Thesis, University of California at Berkeley, 2008.
- [17] 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*, New York, NY, USA, 2014, pp. 107–118.
- [18] G. E. Blelloch, M. A. Heroux, and M. Zagha, “Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors,” School of Computer Science, Carnegie Mellon University, CMU-CS-93-173, Aug. 1993.
- [19] D. Merrill, “Allocation-oriented Algorithm Design with Application to GPU Computing,” Ph.D. Dissertation, University of Virginia, 2011.
- [20] 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*, Aire-la-Ville, Switzerland, Switzerland, 2007, pp. 97–106.
- [21] N. Bell and M. Garland, “Efficient Sparse Matrix-Vector Multiplication on CUDA,” NVIDIA Corporation, NVIDIA Technical Report NVR-2008-004, Dec. 2008.
- [22] A. Monakov, A. Lokhmotov, and A. Avetisyan, “Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures,” in *High Performance Embedded Architectures and Compilers*, vol. 5952, Y. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, and X. Martorell, Eds. Springer Berlin Heidelberg, 2010, pp. 111–125.
- [23] W. Tang, W. Tan, R. Goh, S. Turner, and W. Wong, “A Family of Bit-Representation-Optimized Formats for Fast Sparse Matrix-Vector Multiplication on the GPU,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2014.
- [24] W. T. Tang, W. J. Tan, R. Ray, Y. W. Wong, W. Chen, S. Kuo, R. S. M. Goh, S. J. Turner, and W.-F. Wong, “Accelerating Sparse Matrix-vector Multiplication on GPUs Using Bit-representation-optimized Schemes,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, New York, NY, USA, 2013, pp. 26:1–26:12.
- [25] 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*, New York, NY, USA, 2012, pp. 353–364.
- [26] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, “Optimization of Sparse Matrix-vector Multiplication on Emerging Multicore Platforms,” in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, New York, NY, USA, 2007, pp. 38:1–38:12.
- [27] S. Dalton, S. Baxter, D. Merrill, L. Olson, and M. Garland, “Optimizing Sparse Matrix Operations on GPUs Using Merge Path,” in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, 2015, pp. 407–416.
- [28] S. Baxter, *Modern GPU*. NVIDIA Research, 2013.
- [29] A. Buluc and J. R. Gilbert, “On the representation and multiplication of hypersparse matrices,” in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, 2008, pp. 1–11.
- [30] D. Chakrabarti, Y. Zhan, and C. Faloutsos, “R-MAT: A Recursive Model for Graph Mining,” in *SIAM International Conference on Data Mining*, 2004.
- [31] T. Davis and Y. Hu, “University of Florida Sparse Matrix Collection.” [Online]. Available: <http://www.cise.ufl.edu/research/sparse/matrices/>. [Accessed: 11-Jul-2011].
- [32] NVIDIA, “CUDA.” [Online]. Available: [http://www.nvidia.com/object/cuda\\_home\\_new.html](http://www.nvidia.com/object/cuda_home_new.html). [Accessed: 25-Aug-2011].
- [33] D. Merrill, *CUB v1.5.3: CUDA Unbound, a library of warp-wide, block-wide, and device-wide GPU parallel primitives*. NVIDIA Research, 2015.
- [34] J. D. McCalpin, “Memory Bandwidth and Machine Balance in Current High Performance Computers,” *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.
- [35] 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*, New York, NY, USA, 2013, pp. 273–282.

### A. Abstract

This artifact comprises the source code, datasets, and build instructions on GitHub that can be used to reproduce our merge-based CsrMV results (alongside those of Intel MKL and NVIDIA cuSPARSE CsrMV results) presented in our SC’2016 paper *Merge-based Parallel Sparse Matrix-Vector Multiplication*.

### B. Description

#### 1) Check-list (artifact meta information):

- **Algorithm:** sparse matrix-vector multiplication
- **Program:** CUDA and C/C++ OpenMP code
- **Compilation:** CPU OpenMP code: Intel icc (v16.0.1 as tested); GPU CUDA code: NVIDIA nvcc and GNU gcc (v7.5.17 and v4.4.7 as tested, respectively)
- **Binary:** CUDA and OpenMP executables
- **Data set:** Publicly available matrix market files
- **Run-time environment:** CentOS with CUDA toolkit/driver and Intel Parallel Studio XE 2016 (v6.4, v7.5, and v2016.0.109 as tested, respectively)
- **Hardware:** Any CUDA GPU with compute capability at least 3.0 (NVIDIA K40 as tested); any Intel CPU (Xeon CPU E5-2695 v2 @ 2.40GHz as tested)
- **Output:** Matrix dataset statistics, elapsed running time, GFLOPs throughput
- **Experiment workflow:** `git clone` projects; download the datasets; run the test scripts; observe the results
- **Publicly available?:** Yes

#### 2) How delivered

The CPU-based and GPU-based CsrMV artifacts can be found in the `merge-spmv` open source project hosted on GitHub. The software comprises code, build, and evaluation instructions, and is provided under BSD license.

#### 3) Hardware dependences

For adequate reproducibility, we suggest an NVIDIA GPU with compute capability at least 3.5 and Intel CPU with AVX or wider vector extensions.

#### 4) Software dependences

The CPU-based CsrMV evaluation requires the Intel C++ compiler and Math Kernel Library, both of which are included with Intel Parallel Studio. The GPU-based CsrMV evaluation requires the CUDA GPU driver, nvcc CUDA compiler, cuSPARSE library, all of which are included with the CUDA Toolkit. Both artifacts have been tested on CentOS 6.4 and Ubuntu 12.04/14.04, and are expected to run correctly under other Linux distributions.

#### 5) Datasets

At this time of writing, our matrix parsers currently only support input files encoded using *matrix market* format. All matrix market datasets used in this evaluation are publicly

available from the Florida Sparse Matrix Repository. Datasets can be downloaded individually from the UF website:

<https://www.cise.ufl.edu/research/sparse/matrices/>

Additionally, the `merge-spmv` project provides users with the script `get_uf_datasets.sh` that will download and unpack the entire corpus used in this evaluation. (Warning, this may be several hundred gigabytes.)

### C. Installation

First you must clone `merge-spmv` code to the local machine:

```
$ git clone https://github.com/dumerrill/merge-spmv.git
```

Then you must use GNU make to build CPU and GPU test drivers:

```
$ cd merge-spmv
$ make cpu_spmv
$ make gpu_spmv sm=<cuda-arch, e.g., 350>
```

Finally, you can (optionally) download and unpack the entire UF repository:

```
$ ./get_uf_datasets.sh <path/to/dataset-dir>
```

### D. Experimental workflow

Before running any experiments, users will need to export the Intel OpenMP environment variable that establishes thread-affinity (one-time):

```
$ export KMP_AFFINITY=granularity=core,scatter
```

You can run the program CPU program on the specified dataset:

```
$ ./cpu_spmv --mtx=<path/to/dataset.mtx>
```

Or run the program GPU program on the specified dataset:

```
$ ./gpu_spmv --mtx=<path/to/dataset.mtx>
```

(Note: use the `--help` commandline option to see extended commandline usage, including options for specifying the number of timing iterations and indicating which GPU to use in multi-GPU systems.)

Finally you can run a side-by-side evaluation on the entire corpus:

```
$ ./eval_csrmv.sh <path/to/dataset-dir> <driver-  
prog> > output.csv
```

### E. Evaluation and expected result

The expected results include matrix statistics (dimensions, nonzero count, coefficient of row-length variation), elapsed running time, GFLOPs throughput

### F. Notes

For up-to-date instruction, please visit the `merge-spmv` project’s GitHub page (<https://github.com/dumerrill/merge-spmv>).