Globally Homogeneous, Locally Adaptive Sparse Matrix-Vector Multiplication on the GPU

Markus Steinberger
Max Planck Institute for Informatics
Saarland Informatics Campus
msteinbe@mpi-inf.mpg.de

Rhaleb Zayer
Max Planck Institute for Informatics
Saarland Informatics Campus
rzayer@mpi-inf.mpg.de

Hans-Peter Seidel
Max Planck Institute for Informatics
Saarland Informatics Campus
hpseidel@mpi-inf.mpg.de

ABSTRACT

The rising popularity of the graphics processing unit (GPU) across various numerical computing applications triggered a breakneck race to optimize key numerical kernels and in particular, the sparse matrix-vector product (SpMV). Despite great strides, most existing GPU-SpMV approaches trade off one aspect of performance against another. They either require preprocessing, exhibit inconsistent behavior, lead to execution divergence, suffer load imbalance or induce detrimental memory access patterns. In this paper, we present an uncompromising approach for SpMV on the GPU. Our approach requires no separate preprocessing or knowledge of the matrix structure and works directly on the standard compressed sparse rows (CSR) data format. From a global perspective, it exhibits a homogeneous behavior reflected in efficient memory access patterns and steady per-thread workload. From a local perspective, it avoids heterogeneous execution paths by adapting its behavior to the work load at hand, it uses an efficient encoding to keep temporary data requirements for on-chip memory low, and leads to divergence-free execution. We evaluate our approach on more than 2500 matrices comparing to vendor provided, and state-of-the-art SpMV implementations. Our approach not only significantly outperforms approaches directly operating on the CSR format (20% average performance increase), but also outperforms approaches that preprocess the matrix even when preprocessing time is discarded. Additionally, the same strategies lead to significant performance increase when adapted for transpose SpMV.

CCS CONCEPTS

•Mathematics of computing → Mathematical software performance;

KEYWORDS

SpMV, sparse matrix, GPU, linear algebra

ACM Reference format:
Markus Steinberger, Rhaleb Zayer, and Hans-Peter Seidel. 2017. Globally Homogeneous, Locally Adaptive Sparse Matrix-Vector Multiplication on the GPU. In *Proceedings of ICS '17, Chicago, IL, USA, June 14-16, 2017,* 11 pages.
DOI: http://dx.doi.org/10.1145/3079079.3079086

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICS '17, Chicago, IL, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
978-1-4503-5020-4/17/06…$15.00
DOI: http://dx.doi.org/10.1145/3079079.3079086

$$\mathbf{A} = \begin{bmatrix} \cdot & 5 & \cdot & 9 & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ 2 & 3 & 6 & \cdot & 3 & 6 & \cdot & 3 & 6 & 3 \\ \cdot & \cdot & 7 & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 4 & 8 & 1 & \cdot & \cdot & \cdot & 5 & 7 \end{bmatrix}$$

$$val = \{5, 9, 1, 2, 3, 6, 3, 6, 3, 6, 3, 7, 1, 1, 4, 8, 1, 5, 7\}$$
$$col\_id = \{1, 3, 6, 0, 1, 2, 4, 5, 7, 8, 9, 2, 6, 0, 2, 3, 4, 8, 9\}$$
$$row\_ptr = \{0, 3, 11, 13, 14, 19\}$$

Figure 1: A CSR matrix with (color coded) non-zero splitting.

1 INTRODUCTION

Conceptually, the sparse matrix-vector product (SpMV): $y = \mathbf{A}x$, where $\mathbf{A}$ is a sparse matrix and x and y are dense vectors, is a rather straightforward operation in linear algebra. Nonetheless, it is of utter importance, as it is often evaluated multiple times, be it in solving linear systems, performing eigenanalysis, or querying graph structures. Being part of the critical path of many applications, every performance improvement in SpMV translates directly into gains for the entire application. Traditionally, the standard *compressed sparse rows* (CSR) representation of sparse matrices lends itself to a simple, yet efficient sequential SpMV implementation, as all non-zeros that contribute to one output element are placed next to another in memory, see Algorithm 1. However, the advent of affordable, parallel architectures, poses new challenges for SpMV reflected in recent research efforts [1–4, 6, 7, 9–17, 19, 21–28].

Algorithm 1: Sequential SpMV $y = A \cdot x$

```
1 for i ← 0 to A.num_rows do
2     temp ← 0
3     for k ← A.row_ptr[i] to A.row_ptr[i + 1] do
4         temp ← temp + A.val[k] · x[A.col_id[k]]
5     y[i] ← temp
```

There are usually two options when optimizing SpMV for modern parallel architectures: First, one can adapt the core SpMV algorithm to the available hardware, by, *e.g.*, tuning the execution strategy, adjusting memory access patterns, or generating a better load balance. Second, one can alter the way the matrix is stored with the characteristics of the hardware in mind, such that an SpMV implementation uses the hardware more efficiently. While the second approach potentially offers more space for improvements, it comes with the "hidden charges" of preprocessing and additional storage requirements. Thus, an alternative format only makes sense, if there is sufficient memory available and the preprocessing overhead is

amortized over multiple SpMV iterations. In practice, numerical computing applications encompass other algorithmic steps which may require the matrix in a standard data format. Thus, non-general purpose alternative formats may have limited practical value [9].

Working along the first option, one of the earliest SpMV implementations on the graphics processing unit (GPU), *CSR-Scalar* [10], explored parallelization over the matrix rows. Visibly, as rows may have unequal lengths, each thread ends up with a different nnz, which will reduce performance. Moreover, as rows grow larger, the memory access of threads drifts further apart, rendering the overall approach inefficient [21]. To address these issues, one can assign an entire group of threads to work on a row [6], often referred to as *CSR-Vector*. However, this strategy does not work well for small rows, extremely long rows still cause slowdowns, and the output memory access is inefficient, as only one thread within a group is used. To counter these issues, one can assign an equal number of non-zeros (nnz) to every thread [13], dynamically switch between CSR-Scalar and CSR-Vector [7, 11], or balance the number of elements and rows processed by each thread [15]. However, as we will discuss in section 2, all of the aforementioned approaches still suffer from at least one of the following issues: execution divergence, detrimental output memory access pattern, unnecessary reductions, or are oblivious to the way the input vector x is accessed.

We propose an SpMV method that considers all levels of the GPU memory hierarchy, creating coherent memory access for global device memory while reading non-zeros, column indices, and x (when possible), and when writing y. Our method performs load balancing on a global and on a local level, using a strict non-zero splitting (see Figure 1) to avoid thread divergence when possible. When divergence cannot be avoided, we make sure that it effects efficient operations, like on-chip shared memory access, rather than global memory. We encode data efficiently as we store it in registers or on-chip shared memory. Finally, we dynamically switch between different data combination strategies, avoiding unnecessary computations. To some extent these ideas are also applicable when multiplying with a transpose matrix (SpMVT). We devise the details of our approach in section 3. We evaluate our method on the entire *University of Florida Sparse Matrix Collection* [8], comparing against vendor provided implementations and state-of-the-art approaches. Detailed performance comparisons are shown in section 4, as well as in the supplemental material.

2 BACKGROUND

Arguably, the two most common sparse matrix formats are *coordinate list* (COO) and *compressed sparse rows* (CSR). Although CSR SpMV usually performs favorably in comparison to COO SpMV as it reduces memory bandwidth, COO SpMV implementations [4] can be built around efficient GPU scan primitives [20]. Trying to achieve more uniform workloads and better memory access patterns, a multitude of different formats have been proposed. A simple way of achieving uniform workload is to pad all rows to the same length (ELL format) [4]. More advanced methods partition the matrix to balance the work between threads (PKT format) [3] or bin rows according to their length [1, 16]. While the above-mentioned approaches are competitive for certain matrices, they do not achieve state-of-the-art performance in the general case.

Alternatively, matrices can be organized in blocks. For example using a bit encoding [24], or as a block compressed COO format using bit flags to store row indices (yaSpMV) [27]. yaSpMV even chooses the estimated best block setup for a given matrix. Formats can be combined in a bock-based manner. For example, HYB combines the ELL and COO format [4]. The most suitable format combination depends on the sparsity pattern and requires comparing a multitude of formats [22]. While block-based or hybrid formats can increase performance, their preprocessing time can be in the range of seconds.

2.1 Analysis of State-of-the-art GPU CSR SpMV

To the best of our knowledge, some of the best performers among publicly available GPU CSR SpMV implementations for general matrices are CSR5 [13], CSR-adaptive [7, 11], and merge-based SpMV [15]. The common theme among all three methods is that each thread is assigned consecutive non-zeros from the same row, mirroring parts of the core sequential SpMV (line 4 in Alg. 1). However, they differ in terms of work distribution among threads, memory access, and preprocessing requirements. In the following we will analyze their behavior. Our findings are summarized in Table 1 and flow charts for all methods are provided in Figure 2.

The ensuing discussions relies on a basic understanding of GPU hardware and execution, which we quickly summarize here. Functions executed on a GPU in parallel are called *kernels*. Every kernel is split into *blocks* of threads that run on the same multi-processor and can communicate via on-chip *shared memory*. Blocks are transparently split into *warps* or *wavefronts*, *e.g.*, 32 or 64 threads, which execute on one single instruction, multiple data (SIMD) unit. If threads within the same warp execute different instructions their execution is serialized, leading to *thread divergence*. Similarly, when they access global GPU memory, their access is most efficient if it is *coalesced*, *i.e.*, if they access data within a 128 bytes region. When accessing shared memory, the distance between accessed words does affect performance. However, access is most efficient, if addresses do not fall on the same *banks*, *i.e.*, if $\bmod(address, 32)$ is different for all threads. Also, threads within a warp can access the registers of one another using *shuffle instructions*. The availability of both shared memory and registers is limited on each multi-processor.

CSR5. The core algorithm of CSR5 proceeds by assigning a predefined number n of consecutive non-zeros to every thread, regardless of how they are distributed across rows. Each thread loops over its non-zeros, fetches them from global memory alongside the column id, and loads its entry from x and adds the values up. If one was operating on the original CSR format, threads would access matrix and column id values which are n memory locations apart, resulting in a poor access pattern. However, the preprocessing step of CSR5 rearranges data to achieve coalesced memory access. To correctly handle the end of rows, CSR5 precomputes and stores a bit pattern indicating whether an entry forms the end of a row. Once the end of a row is reached, the temporary sum is written to global memory. As threads are potentially responsible for multiple rows, the resulting memory accesses can be far apart. Also, threads will only concurrently write to global memory, when their bit pattern matches, thus, in general, only few threads will access memory concurrently, leading to thread divergence.

	CSR5	CSR-adaptive	merge-based	ours
global load balancing	equal nnz	**#nnz rounded down to row**	**equal nnz + rows**	equal nnz
local load balancing	equal nnz	**row after row / reduction**	**equal sums + writes**	equal nnz
matrix access	coalesced (**preprocessing**)	pre-loaded to shared	pre-loaded to shared	vector load + shfl
input vector access	**arbitrary**	**arbitrary**	**arbitrary**	thread sorted
local row overlap	**warp scan + global atomics**	**block scan**	**block scan**	warp red. + shared atomics
output access	**arbitrary**	coalesced / **arbitrary**	buffered, coalesced	buffered, coalesced
global row overlap	atomics	**locks**	fix-up pass	atomics
empty row handling	skip (**special handling**)	**not considered**	balanced	skip (**special handling**)
additional memory	**full matrix**	offset buffer	block starts	block starts
separate preprocessing	**data layout + flag buffers**	**offset buffer**	none	none

Table 1: Comparison between different CSR SpMV methods. Characteristics that can be detrimental for performance in comparison to other approaches in bold. For details see Section 2.1.

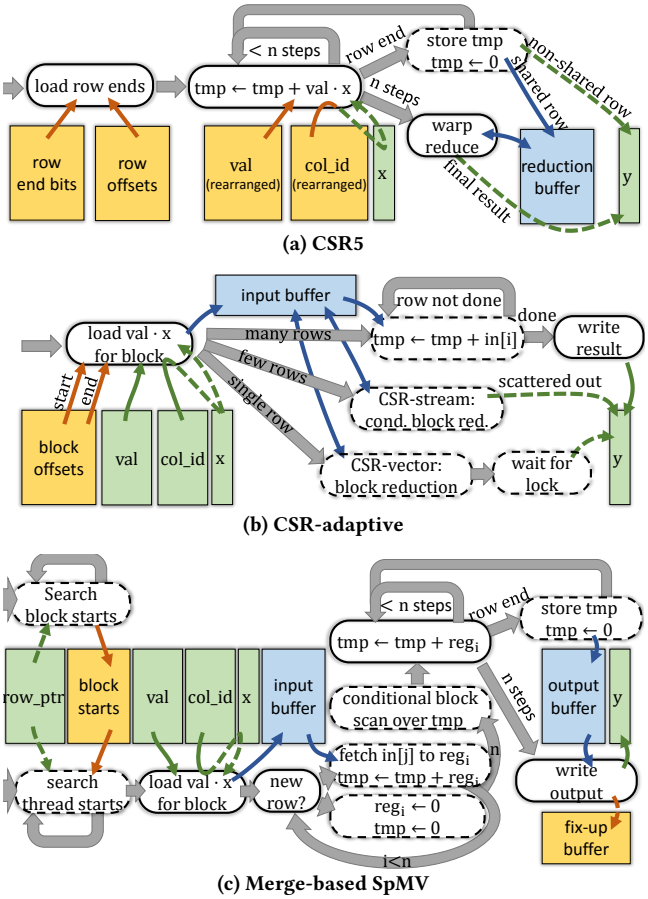


Figure 2: Flow chart for the different CSR SpMV algorithm. Yellow buffers come from preprocessing, green buffers are global inputs to SpMV, blue buffers are placed in shared memory. Thin arrows represent memory accesses, dashed arrows might show detrimental memory access patterns and dashed steps might lead to thread divergence. Note that merge-based SpMV is split into three kernels: (top) generating the block starts buffer; (bottom) performing SpMV; and (not shown) merges the fix-up buffer int y.

If rows are shared across different threads, the result is not written to y directly, but a reduction is performed in shared memory after all threads completed the work on their data. This reduction is even performed when no threads have leftover values and relies on information from preprocessing about which threads work on the same row. As there might be overlaps with other blocks, the result of the first and last row of each block is written using atomic operations to global memory. Thus, the output vector must be zeroed before running SpMV. CSR5 uses an optimization for long rows, *i.e.*, the ones spanning an entire block, for which a direct reduction is executed. As empty rows cannot be represented by their bit pattern, a separate offset array is created during preprocessing, and used in case a block contains empty rows. Although the necessary preprocessing to convert from CSR to CSR5 can be performed on the GPU, it takes between $10 - 30\times$ longer than SpMV.

CSR-adaptive. The core CSR-adaptive algorithm cuts the nonzeros of the matrix into approximately equal chunks and rounds them down, such that small rows do not span across chunks. Chunk sizes are chosen such that they fit in shared memory. As chunk sizes vary, an optimal global load balancing is not achieved. However, they can be loaded in a coalesced manner when a block starts processing its chunk. At the same time, the column indices are loaded and the corresponding input vector entry is fetched. The access pattern for the input vector solely depends on the column indices. Depending on the number of rows a block processes, CSR-adaptive switches between three strategies: CSR-vector (for a single row), CSR-stream (if threads per row >= 2), and CSR-scalar. The CSR-scalar approach assigns one thread to each row, possibly resulting in shared memory bank conflicts and thread divergence as rows have different lengths. However, the output access pattern will be coalesced. Clearly, one long row with many short or empty rows (such that the threshold for CSR-stream is not reached), can lead to significant load imbalance and slowdowns. The CSR-stream approach immediately performs a conditional reduction in shared memory. In comparison to CSR-scalar the reduction is not cost efficient. As the result for each row lies with one of the threads that was assigned to the row, only a subset of threads write an output and thus the memory access pattern is not coalesced.

The approach splits long rows into multiple chunks and uses busy wait locks to resolve access conflicts to output values. The

exact impact of this strategy on performance depends on the low-level GPU scheduler. Certainly, a significant overhead should be expected if there are many long rows. According to the description of the approach, it seems that empty rows are not considered for load balancing. Closely located empty rows can affect global load balancing negatively as they may end up in the same block. Preprocessing for CSR-adaptive is inherently serial and cannot be readily performed on the GPU. It takes $1 - 10\times$ longer than the SpMV on the same matrix. When CPU and GPU do not share the same memory, data transfer further complicates the use of CSR-adaptive for non-static matrices.

Merge-based SpMV. The core merge-based SpMV implementation assigns work to threads in such a way that the sum of handled non-zeros plus writes to the output vector is equal among all threads. To this end, in an initial kernel, an additional buffer is populated with the position of each block's first and last row/non-zero. It is assumed that this buffer has been allocated beforehand and thus no allocation costs arise. During the SpMV step, threads search for their own starting row/non-zero from their block's starting position. Then, all threads cooperatively load the block's non-zeros, column indices and x entries, multiply the values and store them in shared memory. The access to x completely depends on the input data. After a synchronization, every thread starts working on its rows/non-zeros, loading the data from shared memory to registers or filling its registers with zeros when a row end is encountered. At the same time, it computes the SpMV running sums. However, all but the last row's result are discarded, which is used in a block-wide conditional prefix sum, computing the carryover for shared rows. With the carryover as a starting point, every thread again computes the core SpMV summation and stores its results in a shared memory buffer, which is used the smooth out the access pattern to y. As rows can span multiple blocks, the last row result in each block is not written to y, but stored in another buffer, which is subject to a conditional reduction in a separate kernel.

While weighing the number of processed rows against summation operations is not very intuitive per se for a SIMD architecture, where different instructions result in thread divergence, this approach nonetheless comes with a set of advantages: First, empty rows are implicitly handled and do not affect load balancing negatively. Second, the number of output rows is bounded per block and thus the access to y can be buffered in shared memory. However, there are also downsides to the approach. First, each thread has to search for its starting row/non-zero. Second, the number of reads (val, col_id, x) may vary significantly between blocks and thus lead to load imbalances. Third, comparing against approaches considering non-zeros as workload, more work needs to be carried out: Consider rows with two non-zeros each and threads carry out six steps. Merge-based SpMV would assign two rows to each thread ($2 \cdot 2nnz$ and $2 \cot 1$ row ends); CSR5 would assign three rows to each thread. Thus, merge-based SpMV overall needs to execute more threads to complete the work on the same matrix.

3 HOLA-SPMV

Our approach, globally **ho**mogeneous, **l**ocally **a**daptive SpMV, (Hola-SpMV), attempts to achieve an equally good behavior on all performance critical steps discussed earlier (see Table 1). Our approach consists of two kernels. The first deals with global load balancing and work assignment (subsection 3.1). The second describes the usual path through our implementation, considering local load balancing, memory access and data compression (subsection 3.2). The treatment of long rows and empty rows is detailed in subsection 3.3, and the extension to transpose SpMV is given in subsection 3.4.

3.1 Global Load Balancing

We understand global load balancing as how work is distributed between blocks running on the GPU. A homogeneous load among blocks usually leads to similar run times and is essential for good performance. Considering that SpMV is in general bounded by loading input data (non-zeros, col_ids, x) and the inner loop of SpMV runs over the non-zeros, assigning the same number of non-zeros to every block of threads seems to be the intuitive choice. CSR-adaptive's approach of adjusting these boundaries to rows leads to slight imbalances and requires a sequential preprocessing step. The merge-based strategy of considering non-zeros and rows as equal loads leads to blocks potentially performing significantly different number of reads and multiply add instructions. Thus, we choose a strict non-zero splitting, assigning a predefined number of non-zeros to each block. While a static assignment does not require a search, each block still needs to know to which rows its data belongs. As this information is not directly available in the CSR format, we require an initial step to prepare this information in a global buffer. For the example shown in Figure 1, this additional buffer would look as follows:

$$rowStarts = \{0, 1, 1, 2, 4, 4\}$$

We use a scatter approach, as outlined in Algorithm 2. Note that in comparison to CSR5, which also follows a strict non-zero splitting approach, we avoid costly preprocessing and only run this slim kernel before the main SpMV step. Please note that we assume the memory for this buffer has been allocated in advance.

Algorithm 2: AssignRowsGlobal

```
1 a ← row_ptr[tid]
2 b ← row_ptr[tid + 1]
3 blocka ← divup(a, NNZ_PER_BLOCK)
4 blockb ← (b − 1)/NNZ_PER_BLOCK
5 while blocka ≤ blockb do
6 |   blockRowStarts[blocka] ← tid
7 |   blocka ← blocka + 1
8 y[tid] ← 0
```

While a scatter approach requires all row_ptr entries to be read, it is guaranteed that each thread essentially loads data only once from global memory (row_ptr is cached in L1 cache). Each block of the main SpMV step typically handles between two and four thousand non-zeros (depending on the setup). Thus, in most cases multiple rows will be processed by one block and only few threads write to the $blockRowStarts$ buffer. Also the loop (line 5-7), which iterates over the blocks that start with the same row, is usually executed only once. In case a matrix is quite dense (many non-zeros per row) it could be beneficial to use multiple threads to write the output in parallel, reducing the number of loop iterations and

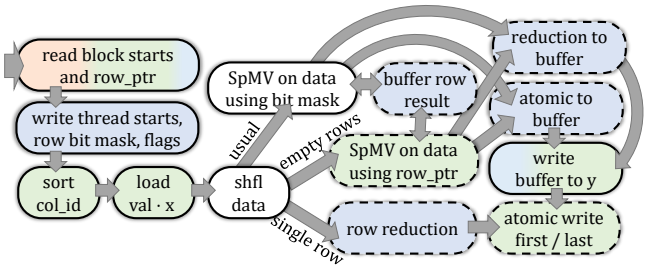


Figure 3: Flow chart for our core Hola-SpMV. Green blocks involve global memory, orange a temporary buffer, blue shared memory. Dashed steps can cause thread divergence.

achieving a better output memory access pattern. This change in execution setup could be determined from the average number of non-zeros per row and thus switched dynamically. However, in our experiments we saw only slight gains when switching the strategy for matrices with very long rows. Thus, for all tests we used the simple one thread per row setup.

Note, that our main SpMV algorithms requires the output to be set to zero beforehand. Starting one thread per row allows us to execute this step in the same kernel (line 8). This would not have been possible with a search-based approach (similar to merge-based SpMV). The runtime would depend on the matrix structure as each thread loads row data until the right offset is found.

3.2 Core Hola-SpMV

The usual path through our SpMV approach is outlined in Algorithm 3 and can be broken down into five major steps:

(1) assign row offsets for local load balancing (ln 1)
(2) input data loading and distribution (ln 2)
(3) summation and buffering in shared memory (ln 6-15)
(4) resolving the output for shared rows (ln 17-20)
(5) transferring data from shared memory to y (ln 22-23)

The algorithm is carried out by each thread block, working on the predefined number of non-zeros assigned to it. The role of threads switches multiple times during the algorithm, making sure that each step is executed in an efficient way.

Local Load Balancing. Our local load balancing approach follows the same modus operandi of the global approach: we assign a predefined number of non-zeros to each thread. In this case, every thread needs to know to which rows its entries belong, such that it can combine them accordingly and write to the right output location. To assign starting rows, we essentially perform the same step as in the global case, just within a block, storing the result in shared memory, as shown in Algorithm 4. However, in addition to the starting row, we also generate information about which entries are to be combined, which we encode in a bit mask (line 13-15). If the ith bit of the mask is set, it signals that the thread's ith non-zero is the last entry in a row. Using a bit mask, saves shared memory as well as registers when performing the following steps of the algorithm. Note that in this way, we only read the *row_ptr* data once in a coalesced manner and keep all information around in a compressed form.

Algorithm 3: CoreHola SpMV

```
1  (rowStarts, rowBits, flags) ←AssignRowsLocal ()
2  (values) ← LoadAndPremultiply ()
3  syncThreads ()
4  if flags[tid]/WARP_SIZE then
5      use row_ptr instead of rowBits
6  row ← rowStarts[tid]
7  bits ← rowBits[tid]
8  syncThreads ()
9  temp ← 0
10 for i ← 0 to NNZ_PER_THREAD do
11     temp ← temp + values[i]
12     if ith bit of bits is set then
13         outputBuffer[row] ← temp
14         temp ← 0
15         row ← row + 1
16 syncThreads ()
17 if bits = 0 for > WARP_SIZE/2 threads then
18     ReduceValues (temp,row,bits)
19 else if temp ≠ 0 then
20     outputBuffer[row] <-atomicAdd- temp
21 syncThreads ()
22 WriteBufferInnerCoalesced ()
23 WriteBufferFirstAndLastAtomic ()
```

Algorithm 4: AssignRowsLocal

```
1  block_start ← blockId · NNZ_PER_BLOCK
2  for r ← tid to block_rows step THREADS do
3      a ← row_ptr[r + block_row_start] − block_start
4      b ← row_ptr[r + block_row_start + 1] − block_start
5      b ← min(b, NNZ_PER_BLOCK)
         threada ← divup(max(a, 0), NNZ_PER_THREAD)
6      threadb ← (b − 1)/NNZ_PER_BLOCK
7      if a equals b then
8          flags[a/(WARP_SIZE · NNZ_PER_THREAD)] = 1
9      else
10         while threada ≤ threadb do
11             rowStarts[blocka] ← r
12             threada ← threada + 1
13         row_end_bit ← (b − threadb · NNZ_PER_THREAD)
14         row_end_bitmask ← 1 << row_end_bit
15         rowBits[threadb] <-atomicOr- row_end_bitmask
```

However, empty rows prohibit efficient use of bit mask approaches (*cf.* CSR5). Thus, we store an additional flag, if a warp faces empty rows (line 7-8). This flag signals all threads of the warp to use the original *row_ptr* instead of the bit mask. Making this decision per warp avoids thread divergence. Details about empty rows are discussed in the special cases subsection (3.3). Again, we could use multiple threads to perform the assignment of *rowStarts* as rows might span multiple threads. Interestingly, using a single thread again turned out to be faster than a more elaborate approach which switches to using an entire warp for the assignment on demand.

Data Loading. As SpMV is mainly bounded by memory access, an efficient way of loading data is of paramount importance. CSR-adaptive and merge-based SpMV use shared memory to load data in an coalesced manner and distribute it from there. The use of shared memory is also necessary in their cases as they do not have a static non-zero assignment, but assign a variable number of non-zeros to each thread. CSR5 uses a static mapping, but changes the data layout in a preprocessing step to achieve coalesced access. Working directly with the CSR data, we need a sophisticated way of loading data. As we already use share memory for *rowStarts* and *rowBits*, we want to avoid using shared memory for loading data. We implement an approach around shuffle instructions, which are in general faster than shared memory [18].

In any case, the underlying problem of data loading comes from the fact that each thread requires multiple consecutive elements from the *val* and *col_id* arrays. Accessing this data directly leads to bad memory access patterns and essentially a transpose of the data is required. Such a transpose can be completed in a cost efficient manner in parallel [5]. We slightly adjust their algorithm, replacing shuffle instructions by more efficient register move, see Algorithm 5.

Algorithm 5: LoadAndPremultiply

```
1  Vecs ← VEC_SIZE/NNZ_PER_THREAD
2  warp_offset ← blockId · NNZ_PER_BLOCK +
     tid/WARP_SIZE · WARP_SIZE · NNZ_PER_THREAD
3  for i ← 0 to Vecs do
4      el ← warp_offset + VEC_SIZE · (laneId + iẆARP_SIZE)
5      vec_val[i] ← LoadVector (val + el)
6      vec_col_id[i] ← LoadVector (col_id + el)
7  (vec_col_id, ids) ← Sort (vec_col_id, 0:NNZ_PER_THREAD)
8  for i ← 0 to VEC_SIZE do
9      vec_v[i] ← vec_val[i] · x[vec_col_id[i]]
10 (·, vec_v) ← Sort (ids, vec_v)
11 for k ← 0 to  mod (laneId, Vecs) do
12     tmp ← vec_v[Vecs − 1]
13     for j ← Vecs − 1 to 1 do
14         vec_v[j] ← vec_v[j − 1]
15     vec_v[0] ← tmp
16 padding ← Vecs − laneId · Vecs/WARP_SIZE
17 section ←  mod (laneId · Vecs, WARP_SIZE)
18 for j ← 0 to VEC_SIZE do
19     source ← section +  mod (padding + j, Vecs)
20     vec_v[j] ← Shfl(vec_v[j], source)
21 for k ← 0 to laneId · Vecs/WARP_SIZE) do
22     tmp ← vec_v[0]
23     for j ← 0 to Vecs − 1 do
24         vec_v[j] ← vec_v[j + 1]
25     vec_v[Vecs − 1] ← tmp
```

Because memory access patterns only matter within a warp, every warp can load its data in a coalesced manner (line 3-6). Then, sequence of move instructions generates a layout that follows diagonals (line 11-15). This allows shuffle instructions to move data to the target thread (line 18-20). Again using move instructions, each thread can restore the original data order (line 21-25). Note that our adapted approach only works when the number of elements held by each thread is a power of two. Loading single elements with every thread would lead to many move instructions. Using vector loads (line 4-6), we can fetch multiple elements in their right order at once (four in case of single precision float and integer, two for double) and treat them as single items in the outlined algorithm. In this way, a single move instruction is sufficient for single precision.

The access pattern to x depends on the row indices of the matrix and ignoring this fact has the potential to slow down SpMV considerably. There are multiple options to smooth out this access pattern: (a) sort the fetched *col_id* within the block, (b) sort within each warp, or (c) sort the elements of each thread. According to our experiments, the overheads of (a) and (b) outweigh the gains seen in typical matrices (across the test data set [8]). However, using a simple odd-even-merge sort for each thread's *col_id*s shows hardly any overhead and increased performance by 2% on average.

Summation and Buffering. After the entire data has been loaded and the bit masks as well as the first row offsets are readily available, the basic SpMV step is straight forward and translates into simple fused-multiply-add instructions on the GPU. If the *rowBit* is set, a conditional move writes the current temporary value to shared memory. As every thread only writes a value if it is the last entry of a row, every row will be written exactly once and we do not need to clear the shared memory buffer beforehand. Note, that the last thread writes the block's last row due to the way we compute the bit mask. As the memory access pattern depends on the input data, bank conflicts can occur. However, bank conflicts in shared memory are preferable to threads writing arbitrarily to global memory.

Algorithm 6: ReduceValues(temp,row,bits)

```
1 end ← bits ≠ 0
2 end_in ← laneId = WARP_SIZE − 1
3 for o ← 1 to WARP_SIZE/2 step o ← 2 · o do
4     temp_in ← ShflDown (if end then 0 else temp, o)
5     if ¬end_in then
6         temp ← temp + temp_in
7     end_in ← end_in or ShflDown (end or end_in, o)
8 if pivot or laneId = 0 then
9     outputBuffer[row] ←(atomicAdd)— temp
```

Shared Rows. To handle rows shared between threads, CSR-adaptive and merge-based SpMV execute a block-wide conditional scan, which requires an up and down pass with barriers in between. We propose an adaptive approach instead: If every thread in a warp holds data for a different row, we directly write the data to shared memory using atomic operations, avoiding the reduction entirely. Even if some threads hold data for identical rows, it is more efficient to use atomic instructions directly instead of performing a reduction. However, if there are more shared rows, we perform a conditional reduction within each warp, as shown in Algorithm 6.

Output. Because the output buffer resides in shared memory, it must be of predefined size. We choose its size to be equal to the number of non-zeros handled by the block, which is a conservative

estimate as long as there are no empty rows. After all threads have added their contribution to the buffer, we transfer it to y in a coalesced manner and make sure that the accesses of all warps are aligned with cache lines, leading to the lowest possible bandwidth for the transfer. As the first and last row might overlap with other blocks, we use atomic operations to add them to y. This is one reason for initializing y to zero.

3.3 Special Case Handling

There are three cases for which we use special code branches. The simplest concerns a single row being assigned to a block. It is directly detected from the starting and end row of the block. If they are identical, we directly execute a block-wide reduction and use one thread to write the result using an atomic add.

The second case arises when empty rows are assigned to a block. In this case, the bit mask cannot be used to increase the current counter, as it needs to skip rows. To avoid thread divergence, we use a per warp flag (see Algorithm 4) to switch entire warps into an alternative mode. When in this mode, threads encountering a bit that signals the end of a row, they look up the correct row from the original *row_ptr* array instead. Additionally, empty rows also require the output buffer to be cleared. To this end, we use the entire warp to clear the rows it is assigned to when switching into this mode. Then, the summation can ignore empty rows. Note that often the *row_ptr* data will reside in L1 cache and thus the overhead for again looking up the row pointers is low.

The above considerations are only valid when the number of overall rows assigned to a block allows them to be cached in shared memory, *i.e.*, the sum of populated rows and empty rows is smaller than nnz assigned to a block. If the number of rows exceeds this threshold, we switch the entire block into an alternative mode. Instead of compressing row starts in bit masks, we use the entire shared memory to store the actual row for each non-zero, which corresponds to decoding a COO description from CSR. Each thread then compares the row ids for its premultiplied values instead of referring to the bitmask. If a new row is encountered, the temporary sum is written to global memory directly. To consider row overlaps, we apply the same heuristics as in the standard case, and use atomic instructions to global memory. With this approach, non-zeros are completely skipped by the algorithm and not written to y, which is possible as y is initialized to zero. The only detriment here is that the output memory pattern is not smoothed out. However, as this special treatment usually only arises, when many empty rows fall between populated rows, memory access would be scattered independently of which thread writes which row. Thus, the impact on performance can be expected to be small.

3.4 Transpose SpMV

A multiplication on a transpose CSR matrix is not likely to achieve the same performance as the direct case, as elements that are consecutive in memory reference arbitrary output locations and thus it is not possible to simply sum elements up. However, we can apply some of the ideas from SpMV to SpMVT, as shown in Algorithm 7. We apply the same global and local load balancing strategies. Instead of premultiplying the input data, we load both values and column ids (line 2). The common trait among the elements in the same row, is that they are multiplied by the same x-value. Using the information from the local load balancing step, we load this common value only once and perform the multiplication (line 9-14). In accordance with our SpMV implementation, we would now combine the values in a buffer in shared memory. Experiments with buffering the values turned out to be less efficient than writing the values directly using atomic operations (line 16-17). This is not surprising, as there is no guarantee that column indices overlap in any way. In case of long rows, it is even guaranteed that they are all unique and thus, no combination is possible. Note that even if we combined data in shared memory, we would have to write the results using atomic operations, as other blocks can reference the same column. However, we apply one more step from Hola-SpMV, namely, sorting the column access within threads (line 15).

Algorithm 7: Hola-SpMVT

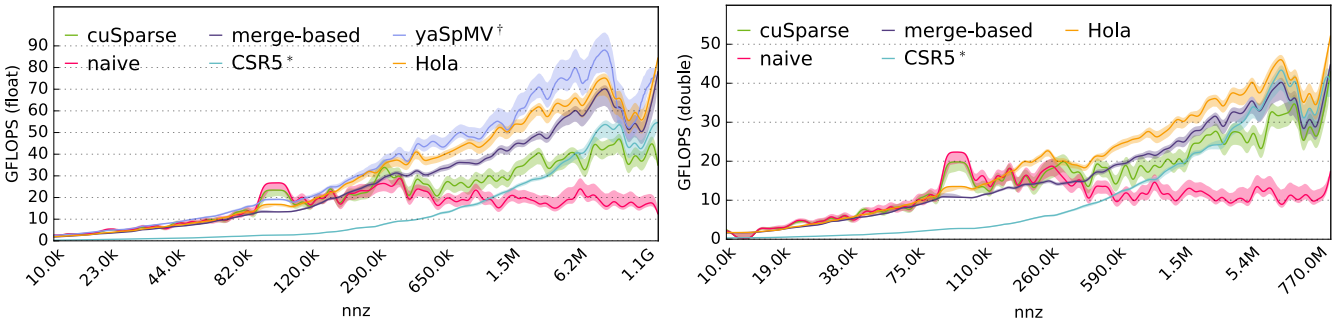
```
1  (rowStarts, rowBits, flags) ←AssignRowsLocal ()
2  (values, col_ids) ← Load ()
3  syncThreads ()
4  if flags[tid]/WARP_SIZE then
5      use row_ptr instead of rowBits
6  row ← rowStarts[tid]
7  bits ← rowBits[tid]
8  syncThreads ()
9  row_factor ← x[row]
10 for i ← 0 to NNZ_PER_THREAD do
11     values[i] ← row_factor · values[i]
12     if ith bit of bits is set then
13         row ← row + 1
14         row_factor ← x[row]
15 (col_ids, values) ← Sort (col_ids, values)
16 for i ← 0 to NNZ_PER_THREAD do
17     y[col_id[i]] ←atomicAdd— values[i]
```

4 EVALUATION

To evaluate our approach, we benchmarked the entire *University of Florida Sparse Matrix Collection* [8], which contains more than 2500 unique matrices of non-trivial size from various application domains with different matrix characteristics. We compare our approach to the vendor provided cuSparse [17], the most recent merge-based SpMV [15], and a naive SpMV implementation [21] which all work directly on CSR, as well as yaSpMV [27] and CSR5 [13], which both require preprocessing and involve a format change. To evaluate our SpMVT performance working directly on CSR, we compare to cuSparse and a naive SpMVT implementation using atomic operations [21]. As test system we use an Intel Xeon CPU×2 @3.40GHz with 32GB of RAM and an NVIDIA Titan X (Pascal) (compute capability 6.1) and CUDA Toolkit 8.0.61.

4.1 SpMV Performance

Summary plots for SpMV on the entire test body are shown in Figure 4, detailed plots for all matrices are available with our source code. Relative speed ups against the evaluated methods are shown in Table 2, commonly compared matrices in Figure 5. For small



* CSR5 requires 10 – 30× of additional preprocessing, † yaSpMV requires up to 150 000× of additional preprocessing time

Figure 4: Trend line of the SpMV performance for all tested methods over the entire University of Florida Sparse Matrix collection. Additional line thickness indicates variance.

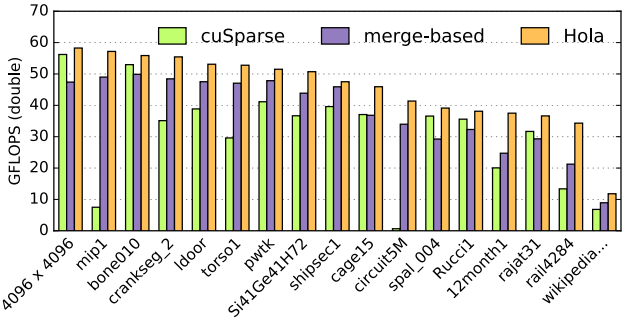


Figure 5: SpMV results for commonly tested matrices.

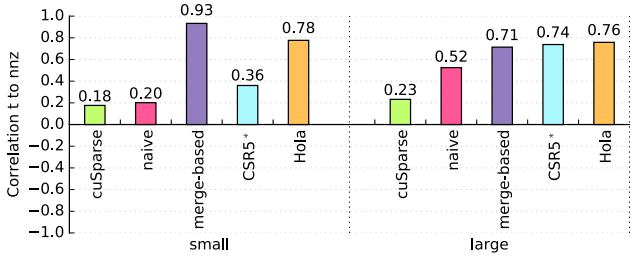


Figure 6: Performance predictability for small and large matrices in double precision (best: 1.0).

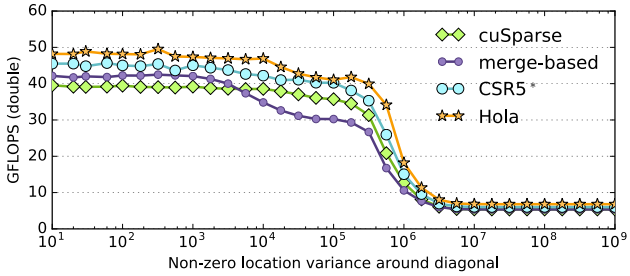


Figure 7: SpMV performance on a 1M square matrix with 22M non-zeros normal distributed around the diagonal with increasing distribution variance.

matrices, the difference between the approaches is small, as performance is dominated by launch overhead and sub-par GPU utilization, *e.g.*, our method starts to fully fill the test GPU for matrices with 400 000 or more nnz. Thus, it is also not surprising that cuSparse and naive, which fill the GPU faster and only start a single kernel, dominate the tests for tiny matrices. However, considering the harmonic mean speedup, our approach performs on average as well on matrices between 10 000 - 300 000 nnz. On larger matrices, which show more variance in row length, the naive approach clearly loses ground. cuSparse's performance also deteriorates, while our approach increases its speedup to 55% and 36%. Hola-SpMV is faster than cuSparse in 94% / 82% of large matrices.

Ignoring approaches that do not work on the original CSR data, merge-based SpMV achieves the closest performance to Hola-SpMV. Overall, the relative behavior of the two approaches is widely consistent across the entire data set, with our approach consistently achieving a better performance. For single precision floating point data, we achieve a speedup of about 13%, for double precision approximately 19%, with our approach being better for 93% / 98% of all matrices. Figure 5 shows that out approach also works well for those matrices that favor cuSparse over merge-based SpMV.

yaSpMV, which only supports single precision float—as expected—achieves the best result in many test cases. The extremely long preprocessing (150 000× time) spent disassembling matrices into perfectly aligned and bit compressed sub-matrices is rewarded by an immense performance increase for matrices which contain many strongly connected regions. However, for more general matrix structures, its performance often drops below Hola-SpMV. Also, note that yaSpMV failed during preprocessing for many matrices and did not yield the correct result for some larger matrices, pointing towards race conditions in the implementation.

Although CSR5 performs significant preprocessing, it did not on average achieve very competitive performance, being 4 – 6× slower than our approach for small matrices and 1.4 – 1.90× slower on large matrices. However, for very large matrices it outperformed cuSparse for single precision and cuSparse and merge-based for double precision matrices. We note that CSR5 does not clear the output vector, although that would be required for correctness. Arguably, this gives it an advantage for matrices with low non-zero count per row and empty rows, where the clearing effort becomes noticeable. Nonetheless, Hola-SpMV is consistently faster.

		small matrices ($10k < nnz < 300k$)					large matrices ($300k < nnz < 800M$)				
		speed up of Hola			better than	best	speed up of Hola			better than	best
		min	max	h. mean	Hola		min	max	h. mean	Hola	
float	cuSparse	0.20	119.36	1.01	60%	17%	0.26	432.74	1.55	6%	3%
	naive	0.14	138.61	0.98	55%	43%	0.20	413.87	2.30	8%	3%
	merge-based	0.27	1.42	1.17	6%	0%	0.21	1.47	1.13	7%	2%
	CSR5*	1.04	13.72	5.83	0%	0%	0.34	8.29	1.90	1%	0%
	yaSpMV†	0.20	1.28	0.88	80%	31%	0.18	1.51	0.87	52%	47%
	Hola	-	-	-	-	9%	-	-	-	-	45%
double	cuSparse	0.33	124.91	1.06	55%	20%	0.36	320.09	1.36	18%	16%
	naive	0.15	169.86	1.03	53%	41%	0.32	309.32	2.34	6%	5%
	merge-based	0.30	1.77	1.21	3%	2%	0.30	1.80	1.19	2%	2%
	CSR5*	0.86	13.35	4.77	0%	0%	0.34	5.21	1.43	7%	6%
	Hola	-	-	-	-	37%	-	-	-	-	71%

* CSR5 requires $10 - 30\times$ of additional preprocessing time, † yaSpMV requires up to $150\,000\times$ of additional preprocessing time

Table 2: Relative speedup of Hola-SpMV over competing approaches and number of cases the respective approaches achieved a better performance than Hola-SpMV / achieved the best performance. Excluding YaSpMV (due to its preprocessing times), naive dominates small matrices, followed by cuSparse and Hola-SpMV. For large matrices, Hola-SpMV significantly outperforms the other approaches leading to 20% mean performance increase for large double precision matrices.

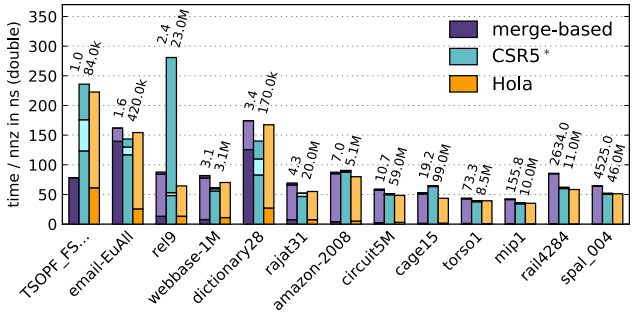
4.2 Performance Predictability

The more balanced an approach is, the more likely it is to achieve a consistent performance across different data sets. However, for CSR SpMV scattered column entries may always lead to slowdowns due to cache-inefficient accesses to x. Thus, a variance free performance cannot be expected, if the distribution of row entries becomes more unpredictable, as shown in Figure 7. The figure also shows that Hola-SpMV consistently performs best over all distribution variances. Interestingly, it seems that merge-based SpMV does not handle variances above 40 000 well, dropping below cuSparse. However, all approaches significantly drop in performance when there is no correlation between the entries of different rows.

However, as can be seen in Figure 6, there is a high correlation between performance and the number of non-zeros for various approaches for the tested matrices. Merge-based SpMV shows the highest correlation for small matrices and Hola-SpMV for large matrices. We attribute the higher correlation of merge-based SpMV for small matrices partially to their optimization, which avoids the initial search kernel for small matrices (see matrices wtih $nnz < 200k$ in Figure 8). Using a scattering approach, we cannot skip our first step and our time always includes a constant launch overhead, increasing the non-linearity of the execution time.

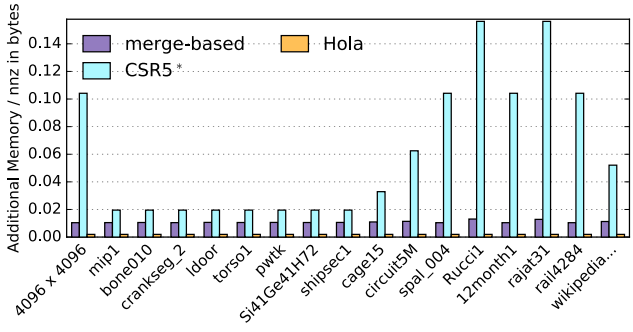
4.3 Row Starts Buffer

Of the tested methods, Hola-SpMV, merge-based SpMV and CSR5 require additional buffers which are filled before running the core SpMV kernel. While Hola-SpMV and merge-based SpMV create these buffers as a first step during SpMV; for CSR5 it is a full preprocessing step. Nevertheless, all three approaches require additional memory and potentially face overheads due to memory allocation of these buffers. In our tests, we have ignored the allocation overhead as memory for these buffers can be allocated once at application start and reused for different matrices during SpMV. The required



* CSR5 timings do not include clearing the output vector

Figure 8: Times of the individual kernels (different shades) per non-zero in ns with average number of non-zeros per row and overall nnz on top. The overhead of the additional kernels reduces with increasing non-zero count per row.



* assuming CSR5 overwrites the original CSR data

Figure 9: Additional memory requirements per non-zero for commonly tested matrices.

additional memory is shown in Figure 9. CSR5 requires significant additional memory and reuses the original CSR storage. Merge-based SpMV treats rows as work and uses smaller and thus more blocks than Hola-SpMV which leads to a multiple of the memory requirement of Hola-SpMV. Our approach requires approximately 0.002 bytes of additional memory per non-zero.

The time for generating the additional buffer for Hola-SpMV is shown in Figure 8. The timings for the fixup kernel of merge-based SpMV as well as the fixup kernels apparently executed by the publicly available version of CSR5 are shown. Note that merge-based SpMV only runs one kernel for the very small *TSOPF...* and two kernels for the small *email-EuAll* and *dictionary28*. The cost of the additional kernels in all cases is larger for matrices with low non-zero count per row, as all approaches execute additional steps proportional to the row count. For more than 7 non-zeros per row, the overhead seems negligible.

4.4 Empty Rows

While merge-based SpMV implicitly handles empty rows, we require special code branches for empty rows which may reduce performance. Surprisingly, as can be seen in Figure 10, our approach also works well when matrices contain many empty rows, achieving the best performance in all but two cases. For matrices where the empty rows are clustered (marked with *), like *relat*, or *rel*, our approach achieves an even better relative performance, as it can skip these empty rows efficiently. The plot also shows that there is no correlation between relative performance and number of switched warp (percentage over the bar). For example, for *web-Google* all warps follow the slow path and we still achieve the best performance. *cnr-2000* is an interesting case, as our approach performs poorly. Deeper investigation showed that the scheduling of blocks to multiprocessors is suboptimal for our approach as warps following the slow branch are launched late and end up on the same multiprocessor. Thus one multiprocessor stays active for nearly twice as long as all others, essentially doubling the runtime. A similar issue arises for *in-2004* but with less severe consequences.

4.5 Transpose SpMV Performance

The performance for transpose SpMV is outlined in Figure 11 as well as Table 3 and selected matrices are shown in Figure 12. We only compare to cuSparse and naive, as they provide SpMVT implementations. Naive follows a one-thread-per-row strategy and uses atomic operations to merge the output. For small matrices, naive and Hola-SpMVT achieve approximately the same performance. For large matrices our approach is on average 15% and 40% faster, and more than 200× faster in particular cases. Overall, we achieve a better performance in 80% to 90% of matrices. In some cases, however, naive can be 100× faster, *e.g.*, for `Rucci1` with 300 GFLOPS. This variance is not surprising, as the performance is dominated by the distribution of column ids and cache behavior as a result of different load balancing strategies. However, for non-pathologic cases, Hola-SpMVT is the best performing approach across various non-zero distributions, as shown in Figure 13. It shares similar characteristics to our direct SpMV, albeit slower, and yields a steady runtime.

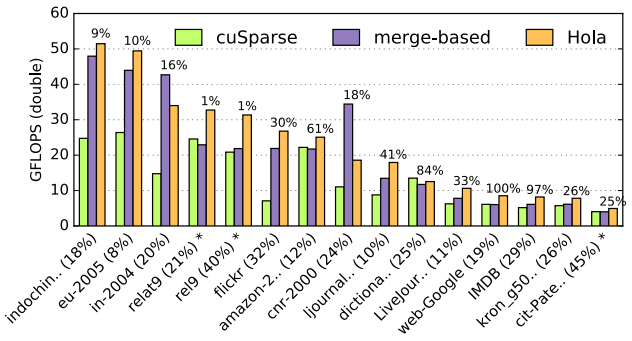


Figure 10: SpMV results for matrices with many empty rows (% with the matrix). The percentage over the bar is the number of warps that altered their behavior in our approach.

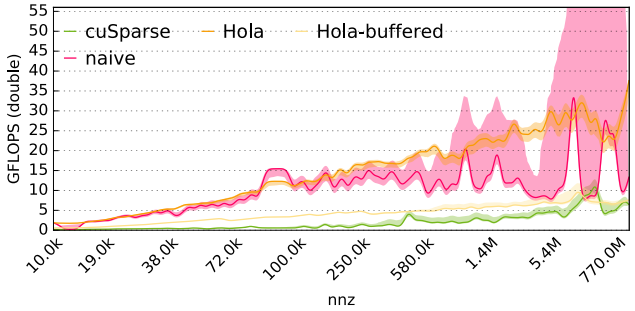


Figure 11: Trend line of the performance for SpMVT. Additional line thickness indicates variance.

		small matrices ($10k < nnz < 300k$)				large matrices ($300k < nnz < 800M$)			
		min	max	h. mean	best	min	max	h. mean	best
float	cuSparse	1.73	168	12.61	0%	0.56	324	6.40	1%
	naive	0.03	73.9	0.98	50%	0.01	217	1.15	16%
	Hola	-	-	-	50%	-	-	-	83%
double	cuSparse	1.84	159	13.1	0%	0.75	226.70	6.55	0%
	naive	0.04	108	1.10	45%	0.01	181.18	1.37	8%
	Hola	-	-	-	55%	-	-	-	92%

Table 3: Relative speedup of Hola-SpMVT

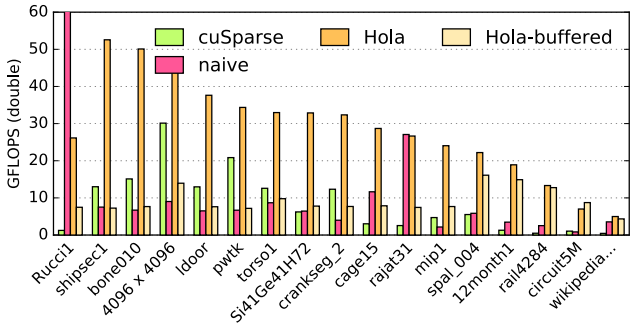


Figure 12: SpMVT results for commonly tested matrices.

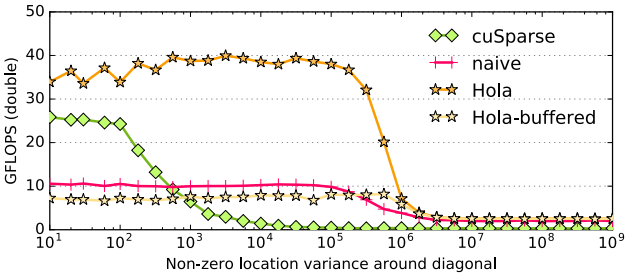


Figure 13: SpMVT performance on a 1M square matrix with 22M non-zeros normal distributed around the diagonal with increasing distribution variance.

As cuSparse most likely computes a full transpose before multiplication, its performance falls short. Certainly, the cost of an explicit transpose can be amortized over many SpMVT operations however, it significantly increases memory requirements. Figure 11 additionally shows the performance obtained when buffering the output (Hola-buffered) in shared memory. The results suggest this strategy does not work well in general for SpMVT and confirm the prediction in Section 3.4. The buffered approach only works well for highly varying non-zero distributions, *e.g.*, `circuit5M`, where it can sometimes outperform the other approaches.

5 CONCLUSION

We have presented a GPU sparse matrix-vector multiplication approach that advances the state-of-the-art in multiple aspects. Starting from a diagnosis of the shortcomings of existing CSR SpMV approaches, we have designed Hola-SpMV to perform global and local load balancing based on non-zero splitting, load input data directly to registers, smooth out memory access patterns, use a conditional warp-wide reduction to handle long rows, and rely on atomic operations when they show the best performance. Most importantly, our approach does not require any preprocessing or characterization of the input matrix and thus it is also perfectly suited for dynamic problems with changing matrix structure.

Performance analysis over the entire University of Florida Sparse Matrix collection placed Hola-SpMV as the most efficient CSR SpMV approach, increasing performance of more than 20% over previous state-of-the-art on average for large matrices, showing the best performance in about three quarters of all cases. For small matrices, it is on par with approaches that show less overhead and achieve better GPU occupancy, highlighting that the core of Hola-SpMV is more efficient than previous approaches. A more detailed performance analysis for matrices with many empty rows revealed that Hola-SpMV also achieves the overall best performance in those cases, when many warps switch into the slower mode. However, occasionally unfortunate scheduling choices can reduce the performance of our approach.

Adopting similar strategies to the transpose SpMV resulted in the fastest approach working directly on the CSR format, to the best of our knowledge. Hola-SpMVT is on par with the naive approach on small matrices, and achieves the best performance in 90% of all large matrices in the test set.

The source code of our approach and supplemental plots are available at https://bitbucket.org/gpusmack/holaspmv.

Acknowledgements

This research was supported by the Max Planck Center for Visual Computing and Communication.

REFERENCES

[1] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan. 2014. Fast Sparse Matrix-vector Multiplication on GPUs for Graph Applications. In *Proc. SC14*. IEEE Press, 781–792.
[2] M. M. Baskaran and R. Bordawekar. 2008. Optimizing sparse matrix-vector multiplication on GPUs using compile-time and run-time strategies. *IBM Reserach Report, RC24704 (W0812-047)* (2008).
[3] N. Bell and M. Garland. 2008. *Efficient Sparse Matrix-Vector Multiplication on CUDA*. Technical Report NVR-2008-004. NVIDIA.
[4] N. Bell and M. Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proc. SC09*. ACM, 1–11.
[5] B. Catanzaro, A. Keller, and M. Garland. 2014. A Decomposition for In-place Matrix Transposition. *SIGPLAN Not.* 49, 8 (Feb. 2014), 193–206.
[6] J. W. Choi, A. Singh, and R. W. Vuduc. 2010. Model-driven Autotuning of Sparse Matrix-vector Multiply on GPUs. *SIGPLAN Not.* 45, 5 (Jan. 2010), 115–126.
[7] M. Daga and J. L. Greathouse. 2015. Structural Agnostic SpMV: Adapting CSR-Adaptive for Irregular Matrices. In *Proc. HiPC 2015*. 64–74.
[8] T. A. Davis and Y. Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1 (Dec. 2011), 1:1–1:25.
[9] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo. to appear. Sparse matrix-vector multiplication on GPGPUs. *ACM Trans. Math. Softw.* (to appear).
[10] M. Garland. 2008. Sparse Matrix Computations on Manycore GPU's. In *Proceedings of the 45th Annual Design Automation Conference (DAC '08)*. ACM, 2–6.
[11] J. L. Greathouse and M. Daga. 2014. Efficient Sparse Matrix-vector Multiplication on GPUs Using the CSR Storage Format. In *Proc. SC '14*. IEEE Press, 769–780.
[12] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. 2014. 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 (2014), C401–C423.
[13] W. Liu and B. Vinter. 2015. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *Proc. ICS '15*. ACM, 339–350.
[14] Y. Liu and B. Schmidt. 2015. LightSpMV: Faster CSR-based sparse matrix-vector multiplication on CUDA-enabled GPUs. In *ASAP 2015*. 82–89.
[15] D. Merrill and M. Garland. 2016. Merge-based Parallel Sparse Matrix-vector Multiplication. In *Proc. SC '16*. IEEE Press, 58:1–58:12.
[16] A. Monakov, A. Lokhmotov, and A. Avetisyan. 2010. Automatically Tuning Sparse Matrix-vector Multiplication for GPU Architectures. In *Proc. HiPEAC'10*. 111–125.
[17] NVIDIA. 2016. *The API reference guide for cuSPARSE, the CUDA sparse matrix library.* (v8.0 ed.). NVIDIA.
[18] Nvidia. 2016. CUDA Programming guide 8.0. (2016).
[19] J. C. Pichel, F. F. Rivera, M. Fernández, and A. Rodríguez. 2012. Optimization of Sparse Matrix-vector Multiplication Using Reordering Techniques on GPUs. *Microprocess. Microsyst.* 36, 2 (March 2012), 65–77.
[20] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. 2007. Scan Primitives for GPU Computing. In *Proc. GH '07*. 97–106.
[21] M. Steinberger, A. Derler, R. Zayer, and H. P. Seidel. 2016. How naive is naive SpMV on the GPU?. In *Proc. HPEC 2016*. 1–8.
[22] B.-Y. Su and K. Keutzer. 2012. clSpMV: A Cross-Platform OpenCL SpMV Framework on GPUs. In *Proc. ICS '12*. ACM, 353–364.
[23] X. Sun, Y. Zhang, T. Wang, X. Zhang, L. Yuan, and L. Rao. 2011. Optimizing SpMV for Diagonal Sparse Matrices on GPU. In *proc. ICPP 2011*. 492–501.
[24] W. T. Tang, W. J. Tan, R. Ray, Y. W. Wong, W. Chen, S.-H. Kuo, R. S. M. Goh, S. J. Turner, and W.-F. Wong. 2013. Accelerating Sparse Matrix-vector Multiplication on GPUs Using Bit-representation-optimized Schemes. In *Proc. SC '13*. ACM, 26:1–26:12.
[25] F. Vázquez, J. J. Fernández, and E. M. Garzón. 2011. A New Approach for Sparse Matrix Vector Product on NVIDIA GPUs. *Concurr. Comput. : Pract. Exper.* 23, 8 (June 2011), 815–826.
[26] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. 2009. Optimization of Sparse Matrix-vector Multiplication on Emerging Multicore Platforms. *Parallel Comput.* 35, 3 (March 2009), 178–194.
[27] S. Yan, C. Li, Y. Zhang, and H. Zhou. 2014. yaSpMV: Yet Another SpMV Framework on GPUs. In *Proc. PPoPP '14*. ACM, 107–118.
[28] H. Yoshizawa and D. Takahashi. 2012. Automatic Tuning of Sparse Matrix-Vector Multiplication for CRS Format on GPUs. In *IEEE CSE 2012*. 130–136.