How naive is naive SpMV on the GPU?

Markus Steinberger*, Andreas Derler†, Rhaleb Zayer* and Hans-Peter Seidel*
*Max Planck Institute for Informatics
Saarbrücken, Germany
Email: {msteinbe,rzayer,hpseidel}@mpi-inf.mpg.de
†Graz University of Technology
Graz, Austria
Email: andreas.derler@icg.tugraz.at

Abstract—**Sparse matrix vector multiplication (SpMV) is the workhorse for a wide range of linear algebra computations. In a serial setting, naive implementations for direct multiplication and transposed multiplication achieve very competitive performance. In parallel settings, especially on graphics hardware, it is widely believed that naive implementations cannot reach the performance of highly tuned parallel implementations and complex data formats. Most often, the cost for data conversion to these specialized formats as well as the cost for transpose operations are neglected, as they do not arise in all algorithms. In this paper, we revisit the naive implementation of SpMV for the GPU. Relying on recent advances in GPU hardware, such as fast hardware supported atomic operations and better cache performance, we show that a naive implementation can reach the performance of state-of-the-art SpMV implementations. In case the cost of format conversion and transposition cannot be amortized over many SpMV operations a naive implementation can even outperform state-of-the-art implementations significantly. Experimental results over a variety of data sets suggest that the adoption of the naive serial implementation to the GPU is not as inferior as it used to be on previous hardware generations. The integration of some naive strategies can potentially speed up state-of-the-art GPU SpMV implementations, especially in the transpose case.**

I. Introduction

Sparse matrix vector multiplication (SpMV) is a key linear-algebra primitive in many scientific algorithms. In scientific computing, there has always been a strive to increase performance in order to target larger problems and reduce the response time in critical applications. A broadly adopted solution to fulfill this ever growing demand for more compute power amounts to the use of parallel architectures. One such parallel architecture is the graphics processing unit (GPU), which attracts increasing interest in high performance computing applications due to its potential high raw compute power and high memory bandwidth. Unsurprisingly, SpMV on the GPU quickly became the focus in a variety of research domains, which is reflected in the large body of work aimed at increasing its performance on multicore architectures like the GPU [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15].

In general, SpMV is usually limited by memory bandwidth rather than compute power, which, in the early days of sparse matrix algebra, led to the development of compressed formats, such as compressed sparse rows (CSR). CSR is still the quasi standard in many application fields [16]. While the CSR format lends itself to efficient serial implementations for direct SpMV and achieves equal performance in the transpose case (SpMVT), a parallel implementation is a non trivial problem. Targeting the GPU, a parallel implementation must consider additional factors, like load balancing between the vast number of cores, instruction divergence on the single instruction multiple data (SIMD) compute units, memory access conflicts between threads, cache performance among thousands of threads, and local memory access patterns. Many authors identified the matrix format as a major hurdle for performance and a myriad of intermediate formats have been proposed to increase SpMV performance [8], [12], [17]. However, the overhead of format conversion as well the integration with existing codes hinders the mainstream adoption of these intermediate formats. Most recently, awareness of these issues prompted a slightly different direction where the modification of the CSR format is restricted to a minimum [13], [14], [15]. Nevertheless, format conversion is usually still more costly than a single SpMV in the altered format.

Despite great strides in performance after format conversion, most existing implementations lack generality in the sense that only direct multiplication is handled and the transpose—which is also often needed in practice—is overlooked. Thus, to perform SpMVT with an intermediate format, not only a transformation to that format is required, but also an explicit transpose operation on the matrix itself. Due to these issues, vendor based implementations such as the standard implementation of cuSparse [18] and CUSP [19] are commonly used. They either provide a direct implementation of SpMVT or at least means to explicitly compute the transposed matrix. However, the provided SpMVT as well as the explicit transpose computation are usually up to an order of magnitude slower than a direct SpMV. While algorithms that repeatedly perform SpMVT can transpose the matrix once and achieve good performance on the GPU, algorithms that require both SpMV and SpMVT are left with slow performance.

The entire body of research on SpMV on the GPU is motivated by the assumption that a naive implementation that follows the serial implementation greatly lacks performance. However, this notion is mostly based on the performance achieved on early GPU architectures and codes compiled with one of the first compilers available for GPUs. In recent years, the hardware has significantly evolved. Register files [20] and cache sizes [21] have been increased, global

$$\mathbf{A} = \begin{bmatrix} 0 & 5 & 0 & 1 \\ 2 & 3 & 6 & 0 \\ 0 & 0 & 7 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \qquad \begin{aligned} val &= \{5,1,2,3,6,7,1\} \\ col\_id &= \{1,3,0,1,2,2,0\} \\ row\_ptr &= \{0,2,5,6,7\} \end{aligned}$$

Fig. 1: A simple matrix in CSR format stores the values and column ids explicitly and offsets to the non-zeros of each row.

memory accesses can directly be cached in fast L1-cache [22], hardware supported floating point atomic operations have been introduced [20], [21], and the compiler nowadays cares for instruction level parallel code. The combination of these advances can lead to a significant boost of naive code performance on current hardware when compared to specialized code that was designed with fewer hardware features in mind.

In this paper, we take a look at a naive implementation of SpMV and SpMVT in the light of recent advances in GPU architecture and compare the performance to commonly used implementations. We are especially interested in the performance of often neglected scenarios like SpMVT, which is usually troublesome in other implementations. In the following, we will first review the CSR format and its transpose counterpart the compressed sparse columns (CSC) format. We then briefly discuss the straightforward serial SpMV and SpMVT implementations and translate them into a naive GPU implementation. We compare the performance of these naive implementations to cuSparse, CUSP, and bhSparse [15], and show that the straightforward adaption of the classical serial formulation to the GPU can achieve similar performance. discuss the implication found throughout the experiments. Finally, with the evaluation in mind, we answer the question how naive a naive SpMV on the GPU really is.

II. CSR and CSC Format

Arguably the most straightforward formats for representing a sparse matrix is to store all of its non-zero entries alongside their row and column coordinates. This format is widely known as coordinate list (COO). While offering complete flexibility in terms of the order in which entries are stored, it also requires a large amount of memory per non-zero entry (row id, column id, and value). The CSR and CSC format reduce the memory requirements at the cost of less flexibility in terms of storage order. Instead of storing row ids for each entry, CSR enforces an ordering of the values according to the rows, such that all entries of a given row lie consecutively in memory. Thus, the row ids can be replaced by pointers to the first element in each row.

Usually, the pointers are stored as simple indices. In this way, a matrix A of size $n \times m$ can be represented by three arrays: The floating point values array val of size nnz (number of non-zeros), an integer array col_id of size nnz that stores the column id of each entry, and an integer array row_ptr of size $n+1$ which stores the offset to the beginning of each row. Thus, an entry $a_{i,j} \in A$ is stored as the combinations of its value $val[k]$ and its column id $col\_id[k] = j$, while $row\_ptr[i] \leq k < row\_ptr[i+1]$ holds, as shown for a simple matrix in Fig. 1. Storing a matrix in CSR format thus reduces the memory requirement by $nnz - n$ integers compared to the COO format. Moreover, reducing the storage cost goes hand in hand with reducing the amount of memory that needs to be read when performing SpMV. Additionally, the CSR format enforces elements of the same row to lie consecutively in memory. Both factors are important for efficient computation, as SpMV is usually memory bandwidth limited and data locality is important for effective cache usage. Note that the CSR format has been used since the early days of sparse matrix formulations [23], [24], and continues to be one of the prevalent matrix formats.

The CSC format is defined analogously to the CSR format, but sorts the entries along the columns instead of the rows and removes the column id array from the COO format rather than the row id. Thus, storing a matrix in CSC format is equivalent to storing the transposed matrix in CSR format.

III. Serial SpMV

Using the CSR format, SpMV can be easily implemented in a serial algorithm, as shown in Algorithm 1. For every output element (ln 2), the algorithm iterates over all entries in the corresponding row of the matrix (line 4). The col_id is used to look up the required entry from the input vector and the result is simply summed up. This simple algorithm loads sequential entries from the val array and col_id array. Thus, good cache behavior can be expected for the core of the algorithm (line 5). Similarly, consecutive values in the row_ptr array are required. If the entries within the same row are close by, even the data loads from x will show good cache behavior.

```
1 SpMV $y = A \cdot x$
2 for $i \leftarrow 0$ to $A.num\_rows$ do
3     $y_i \leftarrow 0$
4     for $k \leftarrow A.row\_ptr[i]$ to $A.row\_ptr[i+1]$ do
5         $y_i \leftarrow y_i + A.val[k] \cdot x[A.col\_id[k]]$
6     end
7     $y[i] \leftarrow y_i$
8 end
```

Algorithm 1: A simple serial SpMV algorithm usually achieves very good performance on the CPU as well as a good cache behavior.

In the serial setup, the implementation of the SpMVT algorithm is as simple as the SpMV implementation, as shown in Algorithm 2. Again, the algorithm can iterate over the rows of the matrix (line 5). However, as the matrix is transposed, the column id determines the output vector entry that needs to be accessed (line 7). Thus, it is also necessary to zero the output vector (line 2-4) before looping over the matrix elements. While the access pattern for val and col_id is again cache friendly, the way the output vector is accessed depends on the matrix data itself.

```
SpMVT $y = A^\top \cdot x$
for $i \leftarrow 0$ to $A.num\_cols$ do
    $y[i] \leftarrow 0$
end
for $i \leftarrow 0$ to $A.num\_rows$ do
    for $k \leftarrow A.row\_ptr[i]$ to $A.row\_ptr[i+1]$ do
        $y[A.col\_id[k]] \leftarrow y[A.col\_id[k]] + A.val[k] \cdot x[i]$
    end
end
```

Algorithm 2: The serial SpMVT algorithm is very similar to the SpMV implementation. Thus, performance similar to SpMV can be expected from a serial implementation.

IV. Naive parallel SpMV

Implementing a naive SpMV and SpMVT algorithm on the GPU is nearly as straightforward as the serial CPU algorithm. Starting from the serial algorithm one can simply parallelize the loop over the matrix rows (line 2 in SpMV and line 5 in SpMVT). Obviously, this has been done by many authors before. However, such a naive parallelization is usually considered very inefficient. In the following we discuss the characteristics of the naive SpMV implementation and highlight under which circumstances it can actually achieve good performance.

A. Caching

The considerations about cache behavior on the CPU are also true for the cache behavior on the GPU. However, when simply storing values in global GPU memory, they do not necessary go through the entire cache hierarchy, but might only be cached in L2-cache, which can be about one order of magnitude slower than the L1-cache found on the GPU multiprocessors. Thus, implementations usually store the matrix as well as the input vector in a texture. Textures are cached directly on the multiprocessors (L1) and no cache coherency algorithm is necessary as texture data is assumed to be immutable. While this somewhat tedious texture setup and texture loading was necessary for previous hardware generations, current Nvidia GPUs (since GK110) support the so called ldg instruction, which generate loads from global GPU memory that use the L1 cache with a non-coherent caching algorithm, *i.e.*, achieving the same behavior and performance as textures. Also, current compilers for the GPU will usually detect memory loads that can be handled as ldg, *e.g.*, when an array is marked as constant. Thus, a naive implementation will achieve the same caching performance as an implementation that uses textures.

B. Load Balancing

A simple parallelization over the rows offers parallel work equal to the number of rows in the matrix. For large matrices this is sufficient to fully fill current GPUs: on the latest GPU architectures about 60 000 threads are required to provide a sufficient number of ready threads for latency hiding of memory transfers. Thus, typical sparse matrices offer sufficient parallel workloads for at least one GPU. However, parallelism alone is only one of the factors that influence GPU performance. Another is load balancing. By parallelizing over the matrix rows, one might assign vastly different work loads to individual threads, *i.e.*, when the number of entries varies between matrix rows. Thus, a single dense row can arbitrarily delay the execution, *i.e.*, one needs to wait for this one thread to finish, while all other GPU cores are idle. This fact has been identified as an issue before and is one of the major motivation for recent alternative matrix data formats [14], [15].

However, load balancing among the entirety of rows is a too coarse way of looking at the issue. The GPU hardware scheduler tries to fill available execution cores whenever possible. To this aim, it operates on thread groups that at least match the SIMD width of the GPU. These groups are often called *warps* (32 threads, Nvidia) or *wavefronts* (64 threads, AMD). Henceforth, we refer to them as warps. As soon as all threads within a warp finish their execution, the hardware scheduler can free their resources and start the execution of new warps (if there are sufficient resources available). Consider a large matrix, whose first 32 rows are dense and all other rows are sparse. In the naive implementation, the warp working on the first 32 rows will take very long in comparison to the threads working on the sparse rows. However, if the matrix is large enough, the hardware scheduler can iterate over all other rows while the first warp operates on its rows. In this case, the overall performance might not suffer at all. Even, if the number of non-zeros per row vary vastly throughout the matrix, performance still can be good, as long as rows of approximately equal size end up within the same warp.

C. Divergence

When rows with different numbers of non-zeros end up within the same warp, performance will certainly deteriorate in the naive approach. As the hardware scheduler only works on the notion of warps, a single thread within one warp can slowdown all other 31 threads, effectively yielding a worst case performance reduction of up to a factor of 32 in comparison to a perfectly load balanced algorithm that parallelizes only over the rows. This issue is usually referred to as a form of thread divergence. While divergence, as a major reason for slowdowns, is limited to a factor of 32, it must be considered that in practice the matrices might not be large enough to hide long running warps and that dense rows might not be launched immediately, which might result in an inability to balance their running time with a higher number of sparser rows. In this cases, the overall slowdown can be significantly larger. Thus, more advanced algorithms try to not only parallelize over the rows, but also within the rows [13], [14], [15].

D. Memory Access Pattern

SpMV in a serial implementation is usually considered memory bandwidth limited. While, on the GPU, the previously described load balancing issues can be a more pressing issue than memory bandwidth, memory access patterns still influence performance. Parallelizing over the rows of the matrix will lead to all threads accessing matrix elements that are as far apart as there are entries in each row, *i.e.*, if there is only a single element in each row, all threads load data that is right next to each other in memory. In this case, the memory requests of all thread within one warp can be handled with a single transaction (for single precision data in val and integer data in col_id). For every additional non-zero entry, the number of required memory transactions increases by one as the number of fetched cache lines increases accordingly. For example, if there are three elements in each row, three transactions are needed to serve an entire warp. The limit is reached when there are 32 or more elements per row. In this case, the memory requests of threads within a warp do not overlap at all and one transaction for each thread is needed. However, after each thread's first load, the following entries are expected to be resident in L1 cache. While this is the ideal case, it must be noted that up to about 2000 threads can run on the same multiprocessor, sharing the L1 cache and possibly replacing each others entries. Thus, it should be expected that fewer elements per row will still increase the performance of the naive algorithm. A more complex algorithm could load the matrix content to shared memory on the multiprocessor and use it as a manual cache [13].

V. Naive parallel SpMVT

While the parallelization of SpMV in the CSR format is straightforward, the parallelization over the rows introduces issues in SpMVT. The output element that needs to be updated in the inner loop (line 7) depends on col_id and not solely on the thread id as before. Thus, different threads may update the same entry of y, introducing race conditions. This problem not only arises in the naive algorithm, but is inherent to the CSR format as entries are sorted according to the rows. This issue also led to alternative data formats developed for multi-threaded CPU SpMV [25]. One obvious way of resolving the race conditions on the GPU is using atomic operations. However, they are widely considered as too slow for scenarios where every single step of the algorithm creates one atomic operation. While that is certainly true for CPU architectures which do not provide hardware support for floating point atomic additions, current GPUs might show higher performance than expected. GPU architectures as shipped today provide full support for single-precision floating point atomic operations in hardware. Double-precision atomic operations will be supported in the upcoming Nvidia architecture [26]. Thus, our naive implementation of SpMVT uses atomic operations to resolve the access conflicts that arise in the serial implementation (line 7).

The loop used to zero the output vector (line 2) can also be fully parallelized and will show perfect memory access patterns. Ignoring the use of atomic operations, all performance characteristics discussed for the naive SpMV implementation also hold for the naive SpMVT. However, the way the input vector is accessed changes: each thread accesses the same entry over and over. Current compilers recognize this fact and load the value only a single time and reuse it during computations. Also note that threads will access consecutive elements of x and a perfect memory access pattern will be achieved.

A. Atomic Operations

The atomic operations that arise in SpMVT are a special case. As the algorithm does not require the previous value stored in y for later computation, the compiler will generate a so-called atomic reduction (red) instruction for single precision floating point data. In this case, the instruction together with its operands can be handed off to the memory controller and the executing thread does not have to wait for the result of the atomic operation. Thus, a red can be significantly faster than a full atomic operation. Although the red instruction can be efficiently implemented as a tree reduction in hardware, its performance will ultimately depend on the number of collisions that arise between different threads. Thus, a matrix with dense columns will generate more collisions and show degenerated performance, whereas matrices with entries spread over different columns will show a higher performance. In an ideal setting, the performance of the naive SpMVT implementation will be similar to the naive SpMV.

While atomic-add instructions for double precision floating point data are not available on current GPUs, they can be emulated using atomic-compare-and-swap instructions which are readily available for 64 bit integer data types. Unfortunately, this precludes the use of red operations and requires a sequential resolution of collisions by the calling threads, as shown in Algorithm 3. Thus, double precision SpMVT can be significantly worse than single precision, as every collision has the potential to lead to another iteration of the loop.

```
1 AtomicAddDouble ($address$, $value$)
2 $old \leftarrow$ read address
3 do
4     $assumed \leftarrow old$
5     $old \leftarrow$ atomicCAS ($address$, $assumed$,
        $value + assumed$)
6 while $assumed \neq old$
```

Algorithm 3: Double precision atomic addition can be implemented using atomic-compare-and-swap if no hardware support for double precision atomic-add is available.

VI. Evaluation

To evaluate the performance of the naive approach, we compare it to the vendor provided cuSparse [18] and CUSP [19], as well as to the most recent bhSparse [15]. In all cases, we assume that a matrix in CSR format is present in GPU main memory and we want to perform a single SpMV/SpMVT operation, highlighting all costs of format conversion and transposition. cuSparse allows to directly use the CSR matrix for SpMV and SpMVT, whereas it is noted that SpMVT can take significantly longer and requires additional memory, *i.e.*, internal conversion takes place. CUSP only provides SpMV, thus, an explicit transpose is required before SpMVT. bhSparse always requires a conversion to their CSR5 format. It also does not provide SpMVT. Thus, for SpMVT we use cuSparse to compute the transpose, then convert to CSR5, and call the provided SpMV. As the CSC format is conceptually the same as the CSR format, we ran all experiments in both CSR and CSC format, revealing non-symmetric behaviors of the used algorithms. Note that using CSC SpMVT equals SpMV in CSR and thus SpMVT is efficient for CSC data while SpMV is not.

As a test system, we used an Intel Xeon E5-2637 v3 CPU running at 3.50GHz, 32GB of memory and an NVIDIA Geforce 980Ti with 2816 compute cores and 6GB of memory running at 1GHz. As test cases we used a variety of matrices from the University of Florida Sparse Matrix Collection [27], including the ones used for the evaluation of compressed sparse blocks on the CPU [25]. The used matrices and performance results for single precision floating point data are shown in Fig. 2. A detailed list of each algorithm's runtime as well as multiplication time ignoring conversion are given in Fig. 3.

Looking at the performance results in Fig. 2, it becomes apparent that cuSparse, CUSP and bhSparse achieve clearly better performance for CSR SpMV (CSC SpMVT) than for CSR SpMVT (CSC SpMV). The data conversion step to CSR5 performed as part of bhSparse test, clearly reduces the overall achieved performance if only a single multiplication is carried out. However, the detailed timings in Fig. 3 show that the pure multiplication performance of bhSparse (gray row) is on average better than CUSP and cuSparse. Thus, as expected, in a scenario where multiple SpMV operations are carried out sequentially, formats that are specifically designed for GPU SpMV achieve better performance.

Comparing the performance of the naive implementation to the other approaches for CSR SpMV does not yield a clear winner. The naive implementation, as well as CUSP achieve the best performance in 5 cases, cuSparse in 3 cases and bhSparse in 1. Ignoring data conversion, bhSparse steals the best performance from CUSP in one more case. The naive approach shows the best performance for matrices that are large, have few nnz with a low standard deviation per row, and show a uniform distribution of entries. Note that a single row with many elements (1.3k) as in rajat31 does not reduce performance significantly as predicted before. The naive approach suffers due to load imbalances and divergence, if the matrix is small and has a high variance among nnz per row, as, *e.g.*, in sme3Dc or poisson3Da. As predicted, if the matrix is slightly larger and at least has rows with similar nnz close by, like in ASIC_320k or webbase, the performance of the naive approach is equal to or slightly better than cuSparse. However, CUSP working with a global sorting approach, generates more uniform workloads in these cases, achieving an overall better performance. Interestingly, bhSparse significantly outperforms all other approaches for ASIC_320k, with its high variance between rows/columns. However, the performance of all approaches is very low in this case. The performance of CSC T is, as expected, most often very similar to CSR. Exceptions arise, when a transpose operations significantly changes the structure of the matrix, like for webbase, cont11, or Rucci, where either significant changes in the distribution of nnz arise or the matrix dimensions change.

Looking at the performance of SpMVT (CSR T and CSC), the naive approach takes the lead in all but one test case, being on average about 6× faster than cuSparse, 17× faster than CUSP, and 26× faster than bhSparse. The naive implementation using atomics is most often similar in performance to the corresponding SpMV case, showing that hardware supported atomic operations can be very competitive to standard memory writes. The most interesting case is Rucci1, where the naive implementation for CSR T outperforms the other approaches by 200-300×. The structure of Rucci1 seems well suited for the naive implementation, as close by rows have many entries in the same columns. Thus, the atomic red can at first locally reduce values within a warp, and only a smaller number of transactions goes through to the global reduction. Note that the naive implementation performs worse for the CSC format, as it offers less parallelism (20 times fewer columns than rows) and the local structure of the matrix is non symmetric. Again note that the large performance gains are only due to the data conversion required by the other approaches. In situation where a transpose can be computed once and reused multiple times, the other techniques produce on average better results, as shown in Fig. 3 (comparing the slightly grayed out values). However, the naive implementation using atomics is still faster in three cases than the other approaches even when they are given the transposed matrix (cont11_l, Rucci1, rajat31)!

The relative performance of the naive SpMV is slightly worse for double precision than single precision. This behavior can be attributed to the increase in cache load (by a factor of almost two due to double precision) and the strong reliance of the naive algorithm on cache performance. The double precision naive SpMVT, with the emulated atomic-add operations—as expected—shows significantly worse performance. Especially, in cases with a high number of collision (ASIC_320k and webbase), the performance is reduced by more than 10× over the SpMV implementation.

Overall, we summarize the results as follows:

- For a subset of matrices (large, low variance of nnz per row), the naive SpMV can outperform other approaches, even when ignoring conversion costs.
- For less regular matrices, the performance of the naive approach is significantly below the other approaches. These cases also reduce its average performance below CUSP and cuSparse.
- For SpMVT, the naive implementation using atomic operations on average achieves similar FLOPS as SpMV, outperforming all other approaches significantly, if the transpose cannot be precomputed.
- The emulated double precision atomic-add operation is too slow on current GPU hardware to achieve competitive results. Upcoming architectures will, however, support double precision atomic-add in hardware.

VII. Conclusion

In this paper, we revisited the naive implementation of SpMV and SpMVT on the GPU. While the presented algorithms are not new, we show that their performance on current GPU hardware is significantly better than what is usually believed.

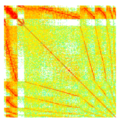
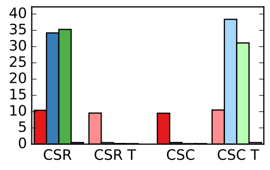
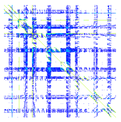
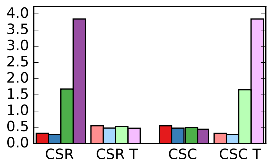
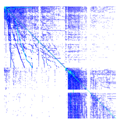
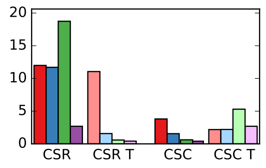
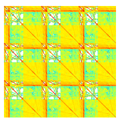
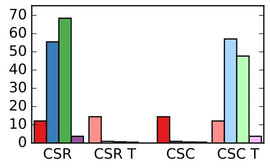
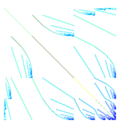
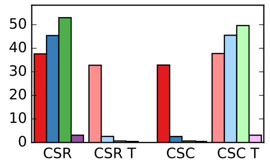
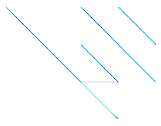
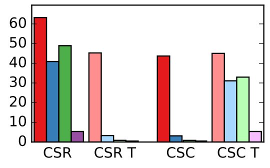
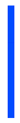
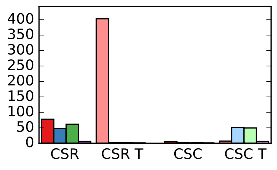
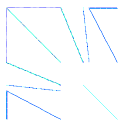
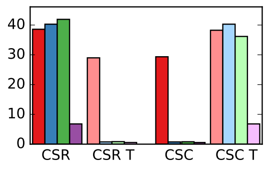
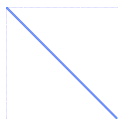
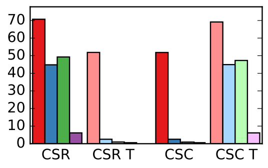
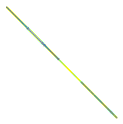
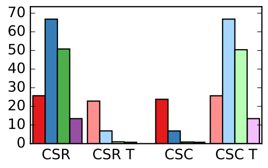
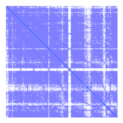
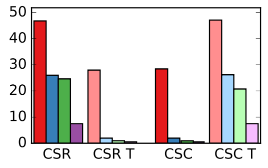
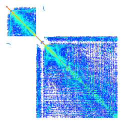
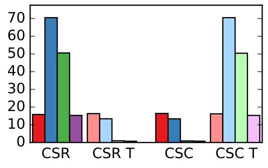
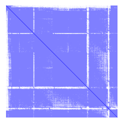
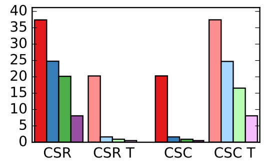
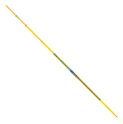
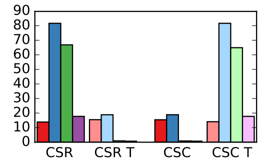
Name (ShortName) Rows×Columns No-Zeros CSR: mean, max CSC: mean, max	Structure	Performance GFLOPS 🟥 Naive 🟩 CUSP 🟦 cuSparse 🟪 bhSparse
poisson3Da (poiss) 13.5k×13.5k 352.8k 26.1(13.8), 110.0 26.1(13.8), 110.0		
ASIC_320k (asic) 321.8k×321.8k 2.6M 8.2(503.0), 203.8k 8.2(503.0), 203.8k		
webbase (web) 1.0M×1.0M 3.1M 3.1(25.3), 4.7k 3.1(36.9), 28.7k		
sme3Dc (sme) 42.9k×42.9k 3.1M 73.3(37.0), 405.0 73.3(37.0), 405.0		
parabolic_fem (para) 525.8k×525.8k 3.7M 7.0(0.2), 7.0 7.0(0.2), 7.0		
cont11_l (cont) 1.5M×2.0M 5.4M 3.7(0.9), 5.0 2.7(2.5), 7.0		
Rucci1 (rucci) 2.0M×109.9k 7.8M 3.9(0.3), 4.0 70.9(11.4), 108.0		
kkt_power (kkt) 2.1M×2.1M 14.6M 7.1(7.4), 96.0 7.1(7.4), 96.0		
rajat31 (rajat) 4.7M×4.7M 20.3M 4.3(1.1), 1.3k 4.3(1.1), 1.3k		
bone010 (bone) 986.7k×986.7k 23.9M 24.2(5.3), 27.0 24.2(5.3), 27.0		
asia_osm (asia) 12.0M×12.0M 25.4M 2.1(0.5), 9.0 2.1(0.5), 9.0		
ldoor (ldoor) 952.2k×952.2k 46.5M 48.9(11.9), 77.0 48.9(11.9), 77.0		
europe_osm (euro) 50.9M×50.9M 108.1M 2.1(0.5), 13.0 2.1(0.5), 13.0		
Flan_1565 (flan) 1.6M×1.6M 117.4M 75.0(11.4), 81.0 75.0(11.4), 81.0		

Fig. 2: Our experiments cover matrices of various sizes and densities. The accompanying statistics show the mean number of entries per row/column the standard deviation of the entry count in parenthesis, and the max number of entries. The performance plots show the achieved performance in GFLOPS (higher is better), measured as three times nnz (multiplication, addition, and per element scaling). CSR, CSR T (lighter color), CSC, and CSC T (lighter color) correspond to an SpMV in CSR format, SpMVT in CSR, SpMV in CSC, and SpMVT in CSC, repsectively. The performance includes the time spent on explicit transpose operations (CSR T and CSC) and format conversion (bhSparse). Note the high performance of the naive approach for CSR T and CSC, where atomic operations are used.

		poiss	asic	web	sme	para	cont	rucci	kkt	rajat	bone	asia	ldoor	euro	flan	avg
CSR	Naive	0.1	25.1	0.8	0.8	0.3	**0.3**	**0.3**	1.1	**0.9**	2.8	**1.6**	8.8	**8.7**	25.4	5.5
		0.1	25.1	0.8	0.8	0.3	0.3	0.3	1.1	0.9	2.8	1.6	8.8	8.7	25.4	5.5
	cuSparse	0.0	28.5	0.8	0.2	0.2	0.4	0.5	1.1	1.4	**1.1**	2.9	**2.0**	13.1	**4.3**	4.0
		0.0	28.5	0.8	0.2	0.2	0.4	0.5	1.1	1.4	1.1	2.9	2.0	13.1	4.3	4.0
	CUSP	**0.0**	4.7	**0.5**	**0.1**	**0.2**	0.3	0.4	**1.0**	1.2	1.4	3.1	2.8	16.1	5.3	**2.7**
		0.0	4.7	0.5	0.1	0.2	0.3	0.4	1.0	1.2	1.4	3.1	2.8	16.1	5.3	2.7
	bhSparse	2.1	**2.1**	3.5	2.6	3.6	3.0	3.8	6.5	9.9	5.3	10.2	9.1	40.1	19.9	8.7
		0.2	0.3	0.4	0.4	0.5	0.4	0.6	1.1	1.3	1.4	2.2	2.3	11.4	5.3	2.0
CSR Transpose	Naive	**0.1**	**14.5**	**0.8**	**0.7**	**0.3**	**0.4**	**0.1**	**1.5**	**1.2**	**3.1**	**2.7**	**8.6**	**16.0**	22.8	**5.2**
		0.1	14.5	0.8	0.7	0.3	0.4	0.1	1.5	1.2	3.1	2.7	8.6	16.0	22.8	5.2
	cuSparse	2.2	16.6	5.9	10.2	4.3	4.9	19.5	57.9	24.2	10.5	39.0	10.4	198.1	**18.7**	30.2
		2.2	16.6	5.9	10.2	4.3	4.9	19.5	57.9	24.2	10.5	39.0	10.4	198.1	18.7	30.2
	CUSP	5.6	15.2	15.9	13.6	17.2	19.2	26.9	52.1	63.3	74.4	79.7	147.2	352.0	374.4	89.8
		0.0	4.7	1.6	0.1	0.2	0.5	0.4	1.2	1.2	1.4	3.1	2.8	16.1	5.2	2.8
	bhSparse	6.7	16.8	22.4	19.8	25.5	32.3	39.8	78.2	107.4	99.5	143.1	193.2	633.5	493.2	136.5
		0.2	0.2	0.4	0.4	0.4	0.4	1.1	1.1	1.3	1.3	2.3	2.3	12.2	5.3	2.1
CSC	Naive	**0.1**	**14.5**	**2.4**	**0.7**	**0.3**	**0.4**	**5.1**	**1.5**	**1.2**	**3.0**	**2.7**	**8.5**	**16.0**	22.9	**5.7**
		0.1	14.5	2.4	0.7	0.3	0.4	5.1	1.5	1.2	3.0	2.7	8.5	16.0	22.9	5.7
	cuSparse	2.1	16.7	6.0	10.1	4.3	5.1	14.6	57.9	24.2	10.6	39.0	10.4	198.2	**18.7**	29.8
		2.1	16.7	6.0	10.1	4.3	5.1	14.6	57.9	24.2	10.6	39.0	10.4	198.2	18.7	29.8
	CUSP	6.7	16.0	15.0	16.5	16.9	19.6	28.4	54.4	66.5	82.9	81.0	158.9	354.9	408.4	94.7
		0.0	4.8	0.6	0.2	0.2	0.4	0.4	1.2	1.3	1.4	3.7	2.8	19.6	5.3	3.0
	bhSparse	6.3	18.0	22.9	19.4	25.0	31.9	42.1	78.7	107.9	100.1	142.4	192.1	634.6	492.3	136.7
		0.2	0.2	0.4	0.4	0.4	0.4	1.1	1.1	1.3	1.3	2.3	2.3	12.2	5.3	2.1
CSC Transpose	Naive	0.1	25.2	4.3	0.8	0.3	**0.4**	3.4	1.1	**0.9**	2.8	**1.6**	8.6	**8.7**	25.0	5.9
		0.1	25.2	4.3	0.8	0.3	0.4	3.4	1.1	0.9	2.8	1.6	8.6	8.7	25.0	5.9
	cuSparse	**0.0**	28.6	4.2	**0.2**	0.2	0.5	**0.5**	**1.1**	1.4	**1.1**	2.9	**2.0**	13.1	**4.3**	4.3
		0.0	28.6	4.2	0.2	0.2	0.5	0.5	1.1	1.4	1.1	2.9	2.0	13.1	4.3	4.3
	CUSP	0.0	4.8	**1.8**	0.2	**0.2**	0.5	0.5	1.2	1.3	1.4	3.7	2.8	19.6	5.4	**3.1**
		0.0	4.8	1.8	0.2	0.2	0.5	0.5	1.2	1.3	1.4	3.7	2.8	19.6	5.4	3.1
	bhSparse	2.1	**2.1**	3.5	2.6	3.6	3.0	3.8	6.5	9.9	5.3	10.2	9.1	40.1	19.9	8.7
		0.2	0.3	0.4	0.4	0.5	0.4	0.6	1.1	1.3	1.4	2.2	2.3	11.4	5.3	2.0

(a) Single Precission

		poiss	asic	web	sme	para	cont	rucci	kkt	rajat	bone	asia	ldoor	euro	flan	avg
CSR	Naive	0.1	35.6	1.0	1.1	0.5	0.5	**0.4**	2.8	1.7	5.8	**2.6**	14.2	12.6	40.9	8.2
		0.1	35.6	1.0	1.1	0.5	0.5	0.4	2.8	1.7	5.8	2.6	14.2	12.6	40.9	8.2
	cuSparse	**0.0**	34.3	1.0	0.2	0.4	0.7	0.8	1.5	2.1	**1.6**	4.8	**2.9**	x	**6.5**	4.4
		0.0	34.3	1.0	0.2	0.4	0.7	0.8	1.5	2.1	1.6	4.8	2.9	x	6.5	4.4
	CUSP	0.0	5.3	**0.6**	**0.2**	**0.3**	**0.4**	0.5	**1.5**	**1.6**	2.5	4.0	3.4	x	6.9	**2.1**
		0.0	5.3	0.6	0.2	0.3	0.4	0.5	1.5	1.6	2.5	4.0	3.4	x	6.9	2.1
	bhSparse	2.5	**3.1**	4.3	3.1	2.4	3.2	5.6	5.9	10.7	6.0	12.2	10.8	x	23.8	7.2
		0.3	0.4	0.5	0.4	0.6	0.8	0.8	1.4	1.7	1.6	3.4	2.7	x	6.0	1.6
CSR Transpose	Naive	**0.3**	526.0	95.7	**1.9**	**1.1**	**1.0**	**0.3**	**3.9**	**4.7**	**7.0**	**4.9**	22.1	22.0	62.7	56.3
		0.3	526.0	95.7	1.9	1.1	1.0	0.3	3.9	4.7	7.0	4.9	22.1	22.0	62.7	56.3
	cuSparse	2.9	22.5	**8.2**	15.4	6.8	8.0	29.0	87.8	37.8	17.6	58.9	**16.9**	x	**29.9**	**26.3**
		2.9	22.5	8.2	15.4	6.8	8.0	29.0	87.8	37.8	17.6	58.9	16.9	x	29.9	26.3
	CUSP	6.1	**16.7**	17.1	14.9	17.5	21.6	29.0	56.7	71.0	84.6	92.7	165.9	x	428.4	78.6
		0.1	5.3	2.0	0.2	0.3	0.6	0.5	2.5	1.6	2.5	4.0	3.4	x	6.8	2.3
	bhSparse	6.6	17.2	22.8	21.6	23.8	32.9	42.0	77.5	113.1	114.9	146.9	214.5	x	551.2	106.5
		0.3	0.3	0.6	0.4	0.6	0.7	2.3	1.3	1.8	1.4	3.5	2.6	x	6.1	1.7
CSC	Naive	**0.3**	523.3	25.0	**1.9**	**1.1**	**1.1**	**11.9**	**3.9**	**4.2**	**6.7**	**4.4**	22.1	21.9	62.8	51.4
		0.3	523.3	25.0	1.9	1.1	1.1	11.9	3.9	4.2	6.7	4.4	22.1	21.9	62.8	51.4
	cuSparse	2.8	22.6	**8.2**	15.4	6.8	8.2	24.2	88.0	37.8	17.5	59.4	**16.9**	x	**29.9**	**26.0**
		2.8	22.6	8.2	15.4	6.8	8.2	24.2	88.0	37.8	17.5	59.4	16.9	x	29.9	26.0
	CUSP	6.3	18.0	15.7	17.9	18.5	22.4	31.3	58.8	76.3	92.1	93.0	182.2	x	467.3	84.6
		0.1	5.5	0.7	0.2	0.4	0.5	0.5	2.3	1.6	2.5	4.7	3.5	x	6.9	2.3
	bhSparse	6.0	**17.1**	22.9	20.9	22.6	33.1	43.5	76.9	112.1	114.3	144.5	216.0	x	521.0	103.9
		0.3	0.3	0.6	0.4	0.6	0.7	2.3	1.3	1.8	1.4	3.5	2.6	x	6.1	1.7
CSC Transpose	Naive	0.1	35.5	5.5	1.1	0.5	**0.6**	5.2	2.8	1.7	5.5	**2.5**	14.2	12.6	40.7	8.9
		0.1	35.5	5.5	1.1	0.5	0.6	5.2	2.8	1.7	5.5	2.5	14.2	12.6	40.7	8.9
	cuSparse	**0.0**	34.3	5.4	**0.2**	0.4	0.9	0.8	**1.5**	2.1	**1.6**	4.9	**2.9**	x	**6.5**	4.7
		0.0	34.3	5.4	0.2	0.4	0.9	0.8	1.5	2.1	1.6	4.9	2.9	x	6.5	4.7
	CUSP	0.1	5.5	**2.1**	0.2	**0.4**	0.7	**0.8**	2.3	**1.6**	2.5	4.7	3.5	x	7.1	**2.4**
		0.1	5.5	2.1	0.2	0.4	0.7	0.8	2.3	1.6	2.5	4.7	3.5	x	7.1	2.4
	bhSparse	2.5	**3.1**	4.3	3.1	2.4	3.2	5.6	5.9	10.7	6.0	12.2	10.8	x	23.8	7.2
		0.3	0.4	0.5	0.4	0.6	0.8	0.8	1.4	1.7	1.6	3.4	2.7	x	6.0	1.6

(b) Double Precission

Fig. 3: Detailed timings in ms for SpMV and SpMVT for all tested implementations and matrices (lower is better, best in bold). The first row corresponds to the time needed for the entire operation (starting with a matrix in CSR/CSC format); the second row is the pure performance of the involved multiplication, excluding transpose and data format conversion operations. Note that the euro matrix caused “out of memory”-errors in double precision format with all but the naive implementation.

Especially in the case, when a matrix is only used for a single SpMV or SpMVT operation, where format conversion and transposition cannot be amortized over many multiplications, a naive implementation is a viable alternative. If the matrix is large and shows little variance among the number of non-zeros per row, the naive implementation even outperformed the other tested approaches for SpMV (leading the charts in 5 test cases). In the case of SpMVT, where explicit transpose operations hurt performance, the naive implementation using atomic operations was the fastest implementation in all but one case. Also, the naive SpMVT does not require any additional memory for data transposition.

While we do not presume that a naive implementation is a universal tool to tackle SpMV and SpMVT, it is time that strategies that did not work well on previous hardware generations are revisited without previous misconceptions. For example, the use of atomic operations for SpMVT is not limited to the naive approach, but approaches like cuSparse or bhSparse can be adapted in this way, building on the advantages of both respective ideas. According to our results for the transition from naive SpMV to SpMVT, we believe the performance of SpMV and SpMVT could also be leveled in other approaches, yielding high performance in all situations. Alternatively, performance gains can also be achieved in practice by switching to a naive SpMV implementation when matrices show a very regular structure and little variance within the number of non-zeros per row.

Acknowledgment

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

References

[1] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder, “Sparse matrix solvers on the GPU: Conjugate gradients and multigrid,” *ACM Trans. Graph.*, vol. 22, no. 3, pp. 917–924, Jul. 2003.

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

[3] J. W. Choi, A. Singh, and R. W. Vuduc, “Model-driven autotuning of sparse matrix-vector multiply on GPUs,” *SIGPLAN Not.*, vol. 45, no. 5, pp. 115–126, Jan. 2010.

[4] A. Monakov, A. Lokhmotov, and A. Avetisyan, “Automatically tuning sparse matrix-vector multiplication for GPU architectures,” in *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers*, ser. HiPEAC’10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 111–125.

[5] J. C. Pichel, F. F. Rivera, M. Fernández, and A. Rodríguez, “Optimization of sparse matrix-vector multiplication using reordering techniques on GPUs,” *Microprocess. Microsyst.*, vol. 36, no. 2, pp. 65–77, Mar. 2012.

[6] M. M. Baskaran and R. Bordawekar, “Optimizing sparse matrix-vector multiplication on GPUs using compile-time and run-time strategies,” *IBM Reserach Report, RC24704 (W0812-047)*, 2008.

[7] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, “Optimization of sparse matrix-vector multiplication on emerging multicore platforms,” *Parallel Comput.*, vol. 35, no. 3, pp. 178–194, Mar. 2009.

[8] 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*, ser. ICS ’12. New York, NY, USA: ACM, 2012, pp. 353–364. [Online]. Available: http://doi.acm.org/10.1145/2304576.2304624

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

[10] F. Vázquez, J. J. Fernández, and E. M. Garzón, “A new approach for sparse matrix vector product on NVIDIA GPUs,” *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 8, pp. 815–826, Jun. 2011.

[11] H. Yoshizawa and D. Takahashi, “Automatic tuning of sparse matrix-vector multiplication for crs format on GPUs,” in *Computational Science and Engineering (CSE), 2012 IEEE 15th International Conference on*, Dec 2012, pp. 130–136.

[12] 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*, ser. PPoPP ’14. New York, NY, USA: ACM, 2014, pp. 107–118.

[13] 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*, ser. SC ’14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 769–780.

[14] Y. Liu and B. Schmidt, “LightSpMV: Faster CSR-based sparse matrix-vector multiplication on CUDA-enabled GPUs,” in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, July 2015, pp. 82–89.

[15] W. Liu and B. Vinter, “CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication,” in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS ’15. New York, NY, USA: ACM, 2015, pp. 339–350.

[16] Y. Saad, *Iterative methods for sparse linear systems*. Siam, 2003.

[17] A. Ashari, N. Sedaghati, J. Eisenlohr, and P. Sadayappan, “An efficient two-dimensional blocking strategy for sparse matrix-vector multiplication on GPUs,” in *Proceedings of the 28th ACM International Conference on Supercomputing*, ser. ICS ’14. New York, NY, USA: ACM, 2014, pp. 273–282.

[18] NVIDIA, *The API reference guide for cuSPARSE, the CUDA sparse matrix library.*, v7.5 ed., NVIDIA, September 2015.

[19] N. Bell and M. Garland, “Efficient sparse matrix-vector multiplication on CUDA,” NVIDIA, Technical Report NVR-2008-004, Dec. 2008.

[20] Nvidia, “NVIDIA Kepler GK110 architecture whitepaper,” 2012.

[21] M. Harris, “Maxwell: The most advanced CUDA GPU ever made,” Nvidia, 2014.

[22] Nvidia, “NVIDIA GF100 whitepaper,” 2010.

[23] H. M. Markowitz, “The elimination form of the inverse and its application to linear programming,” *Manage. Sci.*, vol. 3, no. 3, pp. 255–269, Apr. 1957. [Online]. Available: http://dx.doi.org/10.1287/mnsc.3.3.255

[24] N. Sato and W. F. Tinney, “Techniques for exploiting the sparsity or the network admittance matrix,” *IEEE Transactions on Power Apparatus and Systems*, vol. 82, no. 69, pp. 944–950, Dec 1963.

[25] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, “Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks,” in *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA ’09. New York, NY, USA: ACM, 2009, pp. 233–244.

[26] Nvidia, “NVIDIA Tesla P100 whitepaper,” 2016.

[27] T. A. Davis and Y. Hu, “The university of florida sparse matrix collection,” *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.