COMPRESSED MULTIROW STORAGE FORMAT FOR SPARSE MATRICES ON GRAPHICS PROCESSING UNITS[*]

ZBIGNIEW KOZA[†], MACIEJ MATYKA[†], SEBASTIAN SZKODA[†], AND
ŁUKASZ MIROSŁAW[‡]

Abstract. A new format for storing sparse matrices is proposed for efficient sparse matrix-vector (SpMV) product calculation on modern graphics processing units (GPUs). This format extends the standard compressed row storage (CRS) format and can be quickly converted to and from it. Computational performance of two SpMV kernels for the new format is determined for over 130 sparse matrices on Fermi-class and Kepler-class GPUs and compared with that of five existing generic algorithms and industrial implementations, including Nvidia cuSparse CSR and HYB kernels. We found the speedup of up to $\approx 60\%$ over the best of the five alternative kernels.

Key words. SpMV, CRS format, CSR format, CUDA, hardware accelerators

AMS subject classifications. 65F50, 65Y10

DOI. 10.1137/120900216

1. Introduction. The sparse matrix-vector (SpMV) multiplication is one of the most important kernels in scientific computing with numerous applications ranging from sparse linear solvers to the PageRank algorithm used by Google in its Web search engine. However, its high memory bandwidth requirements combined with the poor data locality exhibited by typical sparse matrices result in poor performance on general purpose processors, which usually attain only a small fraction of their peak performance in this kernel. The literature devoted to SpMV optimization techniques on traditional, cache-based processor designs is ample (see [26] for an extensive overview), and currently one of the major issues is how to exploit new features available in multicore hardware. Several SpMV optimization strategies were already proposed for various designs of chip multiprocessors, including multicore central processing units (CPUs) from Intel and AMD [27], single- and dual-socket STI Cell [27], Sun UltraSPARC T2 [27], field programmable gate arrays (FPGAs) [7, 8], Intel Xeon Phi coprocessors [14, 20], and graphics processing units (GPUs) [1, 2, 3, 9, 11, 15, 17, 18, 23, 24, 28].

In just a few years GPUs have evolved from fixed-pipeline application-specific integrated circuits into highly programmable, versatile computing devices with peak computational performance matching that of the most powerful supercomputers of only a decade ago. These devices can be programmed in high-level programming languages, e.g., Nvidia's C++ for CUDA (proprietary) or OpenCL (open standard). The major problem in their usage is software: any new hardware architecture will

---

[†]Faculty of Physics and Astronomy, University of Wrocław, Wrocław, Poland (zkoza@ift.uni.wroc.pl, maq@ift.uni.wroc.pl, sebastian.szkoda@ift.uni.wroc.pl). The work of the third author was supported by Polish Ministry of Science and Higher Education grant N N519 437939.

[‡]Institute of Informatics, Wrocław University of Technology, Wrocław, Poland, and Vratis Ltd., Wrocław, Poland (lukasz.miroslaw@vratis.com).

succeed only if appropriate software can be developed to exploit the parallelism in the hardware efficiently. However, the current multicore architectures are so diverse and are subject to changes so frequent that to exploit their potential the applications must be highly specialized and use architecture-specific optimization strategies. Moreover, massively parallel architectures, like the one utilized in modern GPUs, require the use of a new programming paradigm, which in turn requires radical rethinking of how numerical computations should be performed. Since the vast majority of the existing scientific software adheres to "old" programming paradigms, and since development of this software took millions of man-hours, perhaps the best way to introduce new concepts is by concentrating on the most important kernels and showing their usability in the "old" environment.

In this paper we present an efficient SpMV algorithm optimized for processors with programmable on-chip shared memory and examine its implementation for Nvidia's CUDA-enabled GPUs. The algorithm is based on a new sparse matrix storage format, CMRS, designed as an extension of a popular compressed row storage (CRS; also known as compressed sparse row, CSR) format. It is characterized by a small memory footprint and small conversion times to other storage formats, which should facilitate its adoption in existing applications. It also turns out to be among the fastest SpMV algorithms available for GPUs. Moreover, in contrast to many recent studies on GPU SpMV, which were based on small sets of sparse matrices, which in turn almost reduced the papers to case studies, here we use a far larger set from the University of Florida Sparse Matrix Collection [6]. This enabled us to make some general statements not only about the absolute performance of our CMRS-based implementation, but also about relative performance of several other alternative solutions for two generations of GPU architectures.

2. Problem statement.

2.1. SpMV multiplication. The aim of the SpMV multiplication algorithm is to calculate the product $\mathbf{y} = \hat{A}\mathbf{x}$, where $\hat{A}$ is a large sparse matrix and $\mathbf{x}$, $\mathbf{y}$ are dense vectors. Typical matrices involved in the SpMV product have thousands or even millions of rows and columns, but the average number of nonzero elements per row rarely exceeds 100. The most interesting—and difficult—matrices are those whose distribution of nonzero elements appears to be unpredictable.

Nonzero elements of $\hat{A}$ are usually stored in an auxiliary array, and additional information is needed to uniquely map the values in the array to their locations in $\hat{A}$. The way this information is stored is called a sparse matrix format. Calculation of an SpMV product essentially reduces to many "multiply and add" operations, which in modern GPUs are implemented as a single fused multiple-add (FMA) instruction. Since SpMV multiplication involves several memory accesses per arithmetic instruction, the SpMV kernel is inherently memory-bound. For example, a server-class Tesla K20X GPU can perform $\approx 6.5 \times 10^{11}$ FMA operations per second and can access its main memory at $\approx 2.5 \times 10^{11}$ B/s, which yields approximately 2.5 operations per byte. For the SpMV kernel this sets the upper bound for the processor computational efficiency to $\approx 1/80$ of its peak theoretical value. Although this value can be increased by using fast on-chip caches, other factors, like additional memory transactions necessary to read sparse matrix format data or reduced off-chip memory throughput due to poor data locality, can decrease it to even smaller values. The main challenge is thus how to exploit and balance all the performance-related features available in hardware, focusing on the utilization of the memory.

2.2. GPU architecture and the CUDA programming model. The architecture of modern GPUs is a massively parallel design which excels in computing-intensive stream data processing. Here we briefly discuss the main properties of the "Fermi" (2010) and "Kepler" (2012) GPU architectures from Nvidia [19].

A GPU contains a number of units called multiprocessors, each one containing a set of relatively simple computing cores called CUDA cores. From the programmer's perspective, multiprocessors are essentially independent single-instruction multiple-data (SIMD) devices in which groups of 32 CUDA threads, called warps, execute the same instruction on multiple data simultaneously. Multiprocessors are connected to high bandwidth (up to $\approx$ 290 GB/s), high latency ($\approx$ 800 clock cycles), limited size ($\leq$ 12 GB) external dynamic random access memory through a coherent L2 cache (up to 1.5 MB). Each multiprocessor has an L1 cache, a read-only texture cache, a 48 KB read-only cache (Kepler K20 and K40 GPUs only), and a constant cache (8 KB). Besides these hardware-managed caches, multiprocessors contain also several fast on-chip memories managed in software: so-called shared memory (up to 48 KB) and registers. The shared memory is shared by all threads belonging to well-defined groups of warps (called blocks) executing on the same multiprocessor. The sizes of these resources are available and, to some extent, configurable at run-time.

Various memories available in GPUs differ not only in their size and speed, but also in latencies. For example, the latency of registers is $\approx$ 20 clock cycles, whereas the latency of the global memory accesses can be as high as 800 clock cycles. To hide such high latencies, each multiprocessor loads into its registers the states of up to 1536 (Fermi) or 2048 (Kepler) threads and attempts to execute the warp that has all operands ready for execution. This leads to massive parallelism with thousands of threads being processed on-chip simultaneously. For this approach to be efficient, the occupancy, defined as the ratio of the number of resident threads to the maximum number of resident threads, must be sufficiently high. Another factor crucial for GPU efficiency is the memory access pattern. For example, the condition for the global memory to be utilized at full speed is that all threads in a warp should access contiguous, 128-byte aligned locations.

CUDA is an abstract general-purpose parallel computing architecture, programming model, and programming environment designed for Nvidia's GPUs [10]. It is based on a few key concepts such as groups of threads (arranged hierarchically in warps, blocks, and a grid), shared memories, and barrier synchronization. These concepts are exposed to the programmer through a minimal set of extensions to a high-level programming language (C or C++). Warps within a block of threads can be executed in any order; similarly, each block of threads can be run on any of the available multiprocessors in an arbitrary order, sequentially or in parallel.

2.3. Existing matrix formats for SpMV multiplication. The simplest sparse matrix format is the coordinate (COO) format, in which the information about the row index, column index, and the value of each nonzero matrix element is stored in three one-dimensional arrays, `RowInd`, `ColInd`, and `Val`, respectively. As an example, consider a 5×5 matrix:

$$(2.1) \qquad \hat{M} = \begin{bmatrix} 1 & 0 & 0 & 2 & 0 \\ 0 & 3 & 0 & 0 & 4 \\ 0 & 0 & 5 & 0 & 6 \\ 0 & 0 & 7 & 8 & 9 \\ 0 & 0 & 0 & 0 & 10 \end{bmatrix}.$$

Its COO representation (with zero-based indexing) reads

$$\texttt{Val} = [\,1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\,],$$
$$\texttt{ColInd} = [\,0\ 3\ 1\ 4\ 2\ 4\ 2\ 3\ 4\ 4\,],$$
$$\texttt{RowInd} = [\,0\ 0\ 1\ 1\ 2\ 2\ 3\ 3\ 3\ 4\,].$$

To complete the matrix definition, one also needs to supply three integers: the number of matrix rows (`rows`), columns (`cols`), and nonzero elements (`nnz`).

In the above example the row-major ordering was used, i.e., the matrix index arrays were first sorted by row indices and then by column indices. In such a case array `RowInd` will typically contain sequences of many identical entries. This property is utilized in the CRS format to reduce the memory footprint by replacing array `RowInd` with a shorter array `RowPtr`. In the most general case this array is defined by the requirement that `RowPtr[j+1] - RowPtr[j]` be equal to the number of nonzero elements in the jth row ($j = 0, \ldots, \texttt{rows} - 1$). If the matrix contains no empty rows, `RowPtr`[j] gives the index into `Val` corresponding to the first nonzero element in the jth matrix row. Array `RowPtr` has exactly `rows`+1 elements and `RowPtr[rows]` $=$ `nnz`. Thus, the CRS representation of $\hat{M}$ reads

$$\texttt{Val} = [\,1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\,],$$
$$\texttt{ColInd} = [\,0\ 3\ 1\ 4\ 2\ 4\ 2\ 3\ 4\ 4\,],$$
$$\texttt{RowPtr} = [\,0\ 2\ 4\ 6\ 9\ 10\,].$$

Note that arrays `Val` and `ColInd` are the same as in the COO format.

Let k be the maximum number of nonzero elements per row. In the ELLPACK/ITPACK (ELL) format an $n \times m$ sparse matrix is represented by two $n \times k$ dense arrays, `Val` and `ColInd`. Array `Val` is constructed from the original matrix by removing all zeros, while `ColInd` holds column indices into `Val`. The rows with fewer than k nonzero elements are padded in `Val` and `ColInd` arrays with 0 and -1, respectively. The ELL representation of $\hat{M}$ is thus

$$\texttt{Val} = \begin{bmatrix} 1 & 2 & 0 \\ 3 & 4 & 0 \\ 5 & 6 & 0 \\ 7 & 8 & 9 \\ 10 & 0 & 0 \end{bmatrix}, \quad \texttt{ColInd} = \begin{bmatrix} 0 & 3 & -1 \\ 1 & 4 & -1 \\ 2 & 4 & -1 \\ 2 & 3 & 4 \\ 4 & -1 & -1 \end{bmatrix}.$$

While the ELL format belongs to the most efficient sparse matrix formats for vector architectures, it may involve a costly storage overhead. Several attempts have been made to modify this format so as to extend its practical usability for general sparse matrices. One such attempt is the hybrid (HYB) format [2, 3], which is a combination of the ELL and COO formats. Another idea is to divide the matrix into several slices, each represented separately in the ELL format, and/or use some kind of matrix transformation, e.g., permutation of rows [17, 9], to reduce padding.

2.4. Existing GPU implementations. One of the first efficient implementations of SpMV on the GPU architecture was proposed by Bell and Garland [2, 3]. They implemented SpMV kernels for several sparse matrix formats, including COO, ELL, and HYB. In addition, two SpMV kernels for the CRS format were provided: scalar and vector. The scalar kernel assigns one thread per matrix row, which results in noncoalesced access to memory and poor performance. The vector kernel assigns

a 32-thread warp to each row; while this ensures contiguous access to the memory, it leads to a large bandwidth waste whenever a row size is much smaller than the warp size. As for COO, the tests showed that it is not flexible enough to handle unstructured matrices efficiently. The ELL format is often the fastest, but fails whenever row sizes vary significantly, as it leads to a large memory overhead. This problem was addressed in the HYB format, in which the matrix is partitioned into a regular part, stored in ELL, and an irregular part, stored in COO [3, 19]. The partitioning of a general matrix is a rather complex operation which requires building a histogram of the row sizes to find the balance between the potential storage overhead of ELL and the computational inefficiency of COO. The authors recommended HYB as the fastest format for a broad selection of unstructured matrices.

Bell's and Garland's SpMV kernels served as building blocks for the CUSP [4] library. To improve coalescing of matrix data accesses for matrices in the CRS representation, CUSP can virtually divide each warp into 2, 4, 8, or 16 smaller parts and assign them to different rows. Mukunoki and Takahashi used the same idea to optimize their CRS kernel for the Kepler GPU architecture [18]. Baskaran and Bordawekar [1] proposed a few other optimization techniques based on exploiting synchronization-free parallelism and optimized off-chip memory access. Another direction of research on improving the efficiency of the SpMV kernel on GPUs focuses on various extensions and modifications of the ELL or CRS formats. This resulted in the development of the ELL-R [23], sliced-ELL [17], ELLR-T [24, 22], and sliced ELLR-T [9] formats, tiling and composite storage [28], as well as the CRS-T [29] and CSR SIC [11] formats.

3. Compressed multirow sparse format. Efficiency of existing GPU implementations of the SpMV product is often significantly better if an ELL-based format, e.g., HYB, is used instead of CRS. The main reason for this is that GPUs are SIMD-like machines with relatively wide SIMD units, often far wider than the average row length. Since efficient utilization of the CRS format requires the matrix elements to be accessed row by row, processing short rows in long SIMD-like units leads to wasting of the computational capability of the device. Therefore, our main idea is to process a sparse matrix in chunks larger than individual rows, at the same time preserving the overall structure of the matrix representation typical of the CRS format. A group of rows processed by an individual SIMD unit shall be called a "strip," and the number of rows in a strip shall be called the "strip height" and denoted `height`. The number of strips, `strips`, is thus equal to `ceil(rows/height)`.

The new sparse matrix format, which we call compressed multirow storage (CMRS) format, comprises one integer parameter `height` and four arrays: data array `Val` and three auxiliary integer arrays, `ColInd`, `StripPtr`, and `RowInStrip`. Arrays `Val` and `ColInd` are the same as in the CRS format. Array `StripPtr` is a generalization of CRS array `RowPtr` and is defined by the requirement that `StripPtr[j+1] - StripPtr[j]` be equal to the number of nonzero elements in the jth strip ($j = 0, \ldots, \texttt{strips} - 1$). If the sparse matrix contains no empty strips, `StripPtr[j]` gives the index into `Val` corresponding to the first nonzero element in the jth strip. Finally, array `RowInStrip`, of length `nnz`, holds the row numbers within individual strips.

Assume `height` $= 2$. Then the CMRS representation of $\hat{M}$ reads

$$\texttt{Val} = [\,1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\,],$$
$$\texttt{ColInd} = [\,0\ 3\ 1\ 4\ 2\ 4\ 2\ 3\ 4\ 4\,],$$
$$\texttt{StripPtr} = [\,0\ 4\ 9\ 10\,],$$
$$\texttt{RowInStrip} = [\,0\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 1\ 0\,].$$

Note that the conversion between the CRS and CMRS formats is trivial and easy to parallelize. In particular, $\texttt{StripPtr}[\texttt{j}] = \texttt{RowPtr}[\texttt{j} * \texttt{height}]$ for $j <$ `strips` and $\texttt{StripPtr}[\texttt{strips}] = \texttt{nnz}$, whereas $\texttt{RowInStrip}[\texttt{k}]$ is the remainder of the row number divided by `height`. It is also clear that both formats are equivalent if $\texttt{height} = 1$; hence CMRS can be regarded as an extension of the CRS format.

The idea that a warp could process a group of adjacent matrix rows was already exploited in [4, 11, 29, 18]. They all used a static mapping of warp threads to the rows, which is inefficient if the lengths of adjacent rows vary significantly. In particular, the CSR SIC format [11] interleaves several matrix rows to form a new SIC row. This, however, requires padding shorter rows with explicit zeros. As this could easily lead to prohibitive memory overhead, the rows must be first reordered according to their lengths, then combined into larger SIC rows, and these are then combined into a few large segments processed by separate GPU kernels. The CMRS format solves these problems by dynamically assigning threads to rows through the `RowInStrip` array. An efficient CMRS-based SpMV kernel requires neither zero-padding nor row reordering and can be implemented as a single GPU kernel.

4. Implementation. Our GPU implementation of the SpMV product for matrices in the CMRS format is based on the vector kernel by Garland and Bell [3], with rows replaced by strips. Each SIMD unit, or warp made of $\texttt{W\_SIZE} = 32$ threads, is assigned a strip to process. The method of doing the SpMV product in a strip is presented in Algorithm 1 and explained below. Note that we used 28 bits of `ColInd[j]` to store a column index and the remaining 4 bits (denoted as `CMRS_BITS`) to store the corresponding value of `RowInStrip`. This made array `RowInStrip` superfluous and explains "uncompression" steps in Algorithm 1.

This algorithm is assumed to be executed in parallel by all threads in a warp. Implicit synchronization of the threads forming a warp is assumed. All auxiliary buffers and temporary variables are local to a warp, so no explicit synchronization of different warps is necessary, which allows for massively parallel processing of strips. The values of matrix elements and the corresponding column indices are read in parallel directly from arrays `Val` and `ColInd`, whereas row indices are assembled from the information held in arrays `StripPtr` and `RowInStrip`. For sufficiently long strips these memory operations are coalesced to a high degree (j runs through consecutive matrix elements), and hence are very fast. Then the necessary elements of the input vector are fetched from the memory. This is the most sensitive part of each parallel SpMV implementation, as the vector elements required by an SIMD unit are often stored in memory locations scattered almost randomly in the memory, and hence their parallel processing is very problematic. Individual products of the matrix by vector elements are computed and stored in a buffer `buf` allocated in the fast on-chip shared memory. The buffer size is quite large, $\texttt{height} \cdot \texttt{W\_SIZE}$, as each thread needs its own memory buffer for each row. The exact mapping of thread lanes and row numbers into `buf` is arbitrary, as long as it is one-to-one, and affects the number of shared memory bank conflicts and the efficiency of the parallel reduction step. The mapping presented in Algorithm 1, i.e., a cyclic assignment of threads, is designed to minimize the latter factor. For example, for $\texttt{height} = 8$ one can reduce 32×8 partial row sums in `buf` into 8 row sums using just 9 instructions.

Our implementation needs `height`-fold more shared memory than the vector kernel of [3]. On the one hand this is beneficial for the parallel reduction, but on the other hand it imposes a severe limit on acceptable values of `height`, as the buffer size in currently available GPUs is restricted to 48 KB. For example, if we take

Algorithm 1. Processing of a strip in the SpMV kernel, $\mathbf{y} = \hat{A} \cdot \mathbf{x}$, for $\hat{A}$ in the CMRS format. This algorithm is executed in parallel by `W_SIZE` ("warp size") threads identified by `thread_lane` $\in \{0, 1, \ldots, \text{W\_SIZE} - 1\}$ and making up a warp. Parameter `strip_id` identifies the strip in the matrix, and `height` is the number of rows making up a strip. Suggested value of `CMRS_BITS` is 4.

Require: Val, ColInd, StripPtr, height, $\mathbf{x}$, $\mathbf{y}$, $0 \leq$ strip_id $<$ strips

$\text{buf}_{i,j} \leftarrow 0$ for all i, j
$M \leftarrow 2^{\text{CMRS\_BITS}}$
strip_start $\leftarrow$ StripPtr$_{\text{strip\_id}}$
strip_end $\leftarrow$ StripPtr$_{\text{strip\_id}+1}$
$\{j$ is the current index into Val and ColInd$\}$
$j \leftarrow$ strip_start $+$ thread_lane
while $j <$ strip_end **do**
  $\{$Load compressed values into register$\}$
  $c \leftarrow$ ColInd$_j$
  $\{$Uncompress the value of RowInStrip$_j\}$
  $r \leftarrow c \mod M$
  $\{$Uncompress the value of ColInd$_j\}$
  $c \leftarrow \lfloor c/M \rfloor$
  $\{$Threads update partial sums$\}$
  $\text{buf}_{\text{thread\_lane},r} \leftarrow \text{buf}_{\text{thread\_lane},r} + x_c \cdot \text{Val}_j$
  $j \leftarrow j +$ W_SIZE
end while

$\{$Parallel reduction of partial sums in rows$\}$
$\text{buf}_{0,r} \leftarrow \sum_{i=0}^{\text{W\_SIZE}-1} \text{buf}_{i,r}$, $r = 0, \ldots,$ height $- 1$

row $\leftarrow$ strip_id $\cdot$ height $+$ thread_lane
$\{$height elements of buf_0 contain row sums$\}$
if thread_lane $<$ height **and** row $<$ num_rows **then**
  $y_{\text{row}} \leftarrow \text{buf}_{0,\text{thread\_lane}}$
end if

---

height $= 16$ and store the data as 4-byte numbers, 64 bytes of the shared memory will be needed for each thread, and so the maximum number of resident threads per multiprocessor will be limited to 768, which translates into the occupancy of 50% for the Fermi and only 37.5% for the Kepler architecture. Taking into account that a large number of resident threads is necessary to hide large memory latencies, we can safely assume that the maximum value of `height` in an efficient implementation for Fermi- or Kepler-class GPUs does not exceed 16. This, in turn, implies that the values stored in array `RowInStrip` are in the range $0, \ldots, 15$ and hence can be encoded in just `CMRS_BITS` $= 4$ bits. The remaining 28 bits are enough to store column indices of matrices with fewer than 2^{28} columns. This is ≈ 20 times more than the size of the largest sparse matrix that we were able to test on a 6 GB device. The SpMV kernel on GPUs is so memory-bound that it is of utmost importance to reduce its memory footprint, even at the cost of several arithmetic operations, which in this kernel are almost free, hence the idea of compressing two integers into a single 32-bit word.
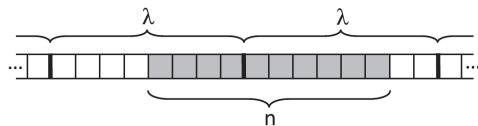
FIG. 1. *A warp transfers a chunk of n consecutive bytes to or from the device accessing the memory in λ-byte-long segments. This can lead to bandwidth waste and low kernel efficiency.*

We also implemented several optional performance optimizations. The first one consists in buffering the input vector in the texture cache [3] or the new 48 KB read-only cache [19] rather than in the L1 cache. The second one consists in enlarging the shared memory size from 16 to 48 KB, at the cost of the L1 cache size. The third optimization strategy, adapted from [1], aims at improving the effective memory bandwidth for arrays `Val` and `ColInd`, for large $\mu = $ `nnz/rows`, by first accessing the nonaligned portion of a strip and then accessing the remaining, aligned portion at full speed. The fourth one consists in reordering the elements of CMRS arrays so that the index array `ColInd` is first sorted by strip indices and then within the same strip by column indices. The idea behind such ordering is the same as for the row-major ordering in the CRS format: enhance the frequency of coalesced or cache-buffered accesses of a SIMD unit to the elements of the input vector. Note that most matrices coming from real problems have some internal structure and the locations of nonzero elements in neighboring rows are correlated. In such cases reordering the entries in the CMRS arrays can have a pronounced impact on SpMV efficiency. Assuming again `height` $= 2$, the CMRS representation of $\hat{M}$ after data reordering would read

$$\mathtt{Val} = [\,1\ 3\ 2\ 4\ 5\ 7\ 8\ 6\ 9\ 10\,],$$
$$\mathtt{ColInd} = [\,0\ 1\ 3\ 4\ 2\ 2\ 3\ 4\ 4\ 4\,],$$
$$\mathtt{RowInStrip} = [\,0\ 1\ 0\ 1\ 0\ 1\ 1\ 0\ 1\ 0\,],$$
$$\mathtt{StripPtr} = [\,0\ 4\ 9\ 10\,].$$

Note that the reordering affects only the matrix internal representation and does not involve any actions on the input and output vectors. Moreover, reordering is local to strips and hence is prone to parallelization.

5. Performance model. The performance model of the CMRS format is based on a few simplifying assumptions: the kernel is memory-bound; each warp processes exactly z nonzero matrix elements; the data are read from or written to contiguous chunks of memory of size $n = bz$, where $b = 4$ or 8 is the number of bytes occupied by a data item; finally, the number of memory transactions is equal to the number of distinct memory segments of size λ touched by the warp while accessing the n-byte chunk of memory; see Figure 1. If one also assumes that the beginning of the n-byte-long chunk is uncorrelated with the the global memory segment boundaries, one concludes (see the Additional Material) that the ratio of the mean number bytes transferred to the bytes actually requested by the kernel is

$$(5.1) \qquad\qquad f(h, \mu) = 1 + \frac{\lambda - b}{h\mu b},$$

where h denotes the strip height (note that $\mu = z/h$ and $b/\lambda \ll 1$). This number must be as close to 1 as possible for the kernel to be efficient. For $h = 1$ this

TABLE 1
Theoretical peak capabilities of the devices used in tests.

	GTX 480	K20M
Single prec. perform. [Tflop/s]	1.3	3.5
Double prec. perform. [Tflop/s]	0.17	1.2
Memory bandwidth [GB/s]	177	208

formula estimates the efficiency of the vector kernel in the plain CRS format. Thus, substituting $h = 1, \lambda = 128$ [19], $b = 4$ (single precision), and $\mu = 2$ (only two nonzero matrix elements per row on average), one obtains $f = 16.5$, which means that for every 16 bytes transferred by the CRS vector kernel, ≈ 15 are wasted. However, the CMRS kernel with $h = 16$ would reduce f down to ≈ 2. Equation (5.1) can be used to estimate the acceleration of the CMRS kernel over the plain CRS vector kernel,

$$(5.2) \qquad a(h, \mu) = \frac{f(1, \mu)}{f(h, \mu)} \approx 1 + \frac{h - 1}{1 + h\mu b/\lambda}.$$

Assuming that $b/\lambda = 1/32$, this formula suggests that for extremely sparse matrices ($\mu \lesssim 5$) and small values of h the CMRS format should be able to accelerate the plain CRS-based vector kernel `height`-fold, as in this case $a(h, \mu) \approx h$. However, the advantages of the CMRS format are not expected to be particularly high for $\mu \gtrsim 100$. Moreover, for $h \gtrsim 16$ the value of $\partial a/\partial h$ is rather small, which gives a theoretical justification for setting 16 as the upper bound for `height` in our implementation. This formula implies also that processing several matrix rows with a single warp will become even more critical if the value of λ increases in some future GPU architectures.

6. Results.

6.1. Hardware and software specification. The tests were performed on two Nvidia devices, GTX 480 (1.5 GB, "Fermi" architecture) and Tesla K20M (5 GB, "Kepler" architecture). The ECC memory support in the Tesla device was switched off for a larger bandwidth. In both cases the operating system was a 64-bit Linux with Nvidia GPU driver v. 319.21 and CUDA 5.5. Theoretical capabilities of these devices are listed in Table 1.

6.2. Test matrices. The tests were performed using 132 square real matrices from the University of Florida Sparse Matrix Collection (UF SMC) satisfying $10^6 \leq$ `nnz` $\leq 10^8$, including all matrices with `nnz` $\geq 5 \times 10^6$, and three additional sparse matrices of our choice. The matrices are of various sizes and represent a wide spectrum of applications and structure patterns. In particular, our tests include all square matrices used in several recent studies on GPU SpMV performance [12, 3, 23, 24, 17, 13]. We excluded from the tests a few sparse matrices with dense rows (`lp1`, `circuit5M`, `Chebyshev4`, `rajat30`, `FullChip`), because such matrices require special algorithms, as will be discussed below. Some UF SMC matrices, e.g., `shipsec8`, are available in two versions: with and without explicit zero entries. In such cases we tested both representations if the number of explicit zeros is larger than `nnz`/10. The additional matrices of our choice include a synthetic matrix, `p7`, which is a large ($10^7 \times 10^7$) random permutation matrix, `dense4`, which is a dense $10^4 \times 10^4$ matrix treated as a sparse one, and `aorta`, which is a sparse matrix representing the pressure equation in the problem of the flow through the human abdominal aorta [16]. We included `p7` to get a better insight into the role played by structural correlations between

adjacent rows and the impact of (un)coalesced accesses to the input vector; `dense4` is an example of a matrix which can be processed at the highest performance; and `aorta` is an example of a sparse matrix for which efficiency of the SpMV kernel is of utmost importance, as it directly affects the time to solution in biomedical applications.

6.3. Optimal CMRS parameters. For each test matrix and each combination of optimization parameters, our CMRS implementation of the SpMV product was called 11 times and the execution times were recorded. The largest time was omitted and the remaining 10 results were analyzed to find their average and standard deviation. The optimization parameters included the strip height ($\mathtt{height} = 2, 3, \ldots, 16$), the number of threads per block ($\mathtt{BS} = 64j, j = 1, \ldots, 8$), and four Boolean parameters referring to the optimization techniques described in section 4, independently for K20M (Kepler) and GTX 480 (Fermi) GPUs. We searched this parameter space for universal values that would give SpMV times as close as possible to the shortest SpMV execution time $\tau_{\min}$ obtained through the brute-force search, for as many test matrices as possible. We came to the following conclusions. The optimal block size is 128 threads. The optimal strip height depends on the GPU architecture and the number of bytes occupied by each matrix value and reads 6 (K20M, float), 4 (K20M, double), 12 (GTX 480, float), or 8 (GTX 480, double). The data should be sorted, the size of the shared memory per multiprocessor should be set to the maximum value (48 KB), the input vector should be cached either in the texture cache (GTX 480) or, if supported by the device, in the new 48 KB read-only cache (K20M), and arrays `Val` and `ColInd` should be aligned for reading if $\mathtt{nnz/rows} \geq 32$.

One should bear in mind that all these optimization parameters are not only correlated with each other, but also depend on the representation of the matrix values (float or double) and on the matrix structure. For example, for some matrices the texture cache turns out to be more efficient than the much larger 48 KB cache, and quite often there exist better values of the strip height. Fortunately, the above-mentioned choice of the optimization parameters yields optimal or nearly optimal SpMV times for most of the matrices (see section 6.8 below).

While the choice of the Boolean optimization parameters can be rather easily justified based on general properties of GPUs, the values of the optimal block size and strip height deserve closer inspection. The value of $\mathtt{BS} = 128$ is the smallest block size which allows for the full utilization of GPU's memory bandwidth (data not shown), and small blocks are preferable for problems where different warps may have to process different amounts of data. As for the optimum value of the strip height, its value limits the occupancy, which, in turn, has a profound impact on the kernel bandwidth. This is illustrated in Figure 2. Panel (a) shows how the speed of reading arrays `ColInd` and `Val` depends on `height`. We measured this relation using a very simple kernel that does nothing but read simultaneously two streams of data from the arrays, given that `height` words (floats or doubles) per thread are reserved in the shared memory and limit the occupancy. The results show some regression of the Kepler architecture relative to its predecessor. First, since the maximum number of concurrent threads per multiprocessor was increased in Kepler by $4/3$ without increasing the size of the shared memory, the number of shared memory words per thread available in Kepler for a given target occupancy was lowered by $3/4$. Second, while lowering the occupancy down to $2/3$ does not affect the speed at which Fermi can access the global memory, this speed deteriorates quickly in Kepler once the occupancy drops below 100%. Consequently, the optimal value of `height` for Kepler is expected to be $4/3 \times 3/2 = 2$ times lower than for Fermi, which we can actually see in our tests.
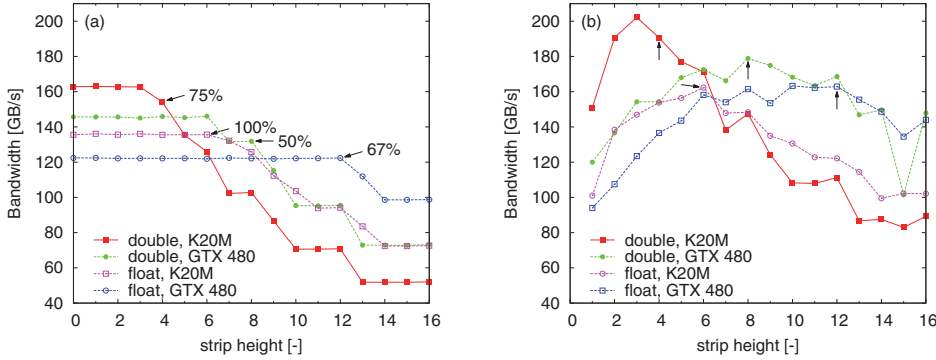
FIG. 2. *(Color online) The dependency of the memory bandwidth on the strip height for* (a) *an idealized case of a kernel that only reads arrays* `ColInd` *and* `Val` *and* (b) *full CMRS SpMV kernel for matrix* `rma10` *calculated using the* β_- *metric of* (6.1). *Arrows point out the data obtained for the values of the optimal strip height, as adopted in our CMRS implementation. The arrow labels in panel* (a) *show the corresponding multiprocessor occupancy, whereas "float" and "double" refer to the type of elements stored in* `Val`.

The impact of the bandwidth-occupancy relation on the actual performance of the CMRS SpMV kernel is visualized in Figure 2 (b). Note that as the value of `height` is increased, the kernel bandwidth initially quickly increases and either saturates at `height` ≈ 8 or hits the threshold value above which the GPU memory bandwidth starts to deteriorate. From this point on the performance of the SpMV kernel starts to decrease, as it is limited by the occupancy-related factors. It is instructive to see how closely the curves in panel (b) follow those in panel (a), especially for K20M.

6.4. Methodology. All computations were also repeated using two standard CRS-based SpMV kernels: scalar and vector, as described in section 2.4. We used our own implementations of these kernels and applied the brute-force method to find the best possible SpMV times. The vector kernel was optimized with respect to all relevant parameters used for CMRS optimization, whereas the scalar kernel was optimized with respect to the value of the block size and the usage of the cache(s). The purpose of using extensive brute-force search for the CRS data format was to ensure that any acceleration of the CMRS over CRS implementation is related to the data format. Finally, we also measured the computational efficiency of three freely available SpMV implementations for GPUs: Nvidia cuSparse 5.5 implementations for CRS and HYB formats and the CUSP 0.3.0 implementation (CSR-tex) for the CRS format. Each library function was treated as a black box and called using the default configuration. CuSparse is a closed-source, proprietary library that can be regarded as an industry standard and reference point, whereas CUSP is an open-source library containing several SpMV implementations for various data formats. While we found that the CUSP kernels are generally less efficient than other kernels considered in this study, we decided to include the data for the CUSP CSR-tex kernel, as it features an improved version of the CRS-vector kernel, aimed at accelerating the SpMV operation for extremely sparse matrices.

For each matrix the computational efficiency of the CMRS SpMV kernel was determined as the ratio of the number of elementary arithmetic operations to the SpMV kernel time, i.e., $(2\texttt{nnz} - \texttt{rows})/\tau$. The bandwidth was calculated as the total number of bytes that had to be transferred to or from the GPU main memory, β, divided
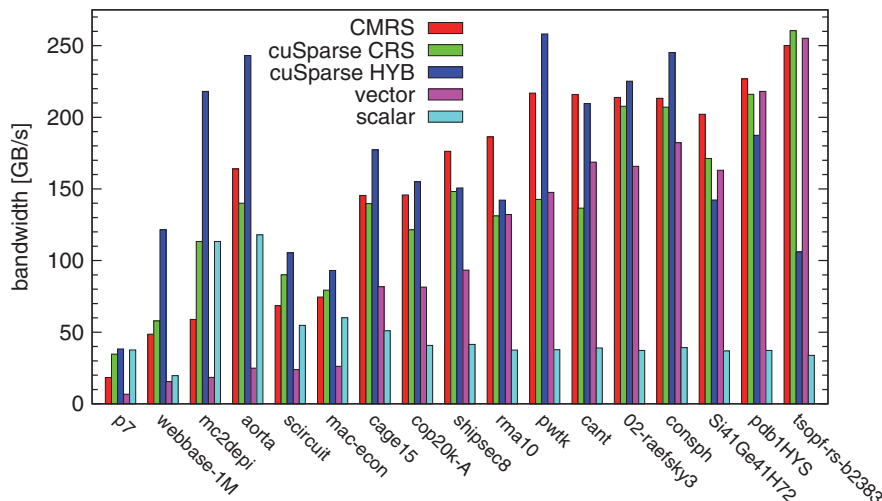
Fig. 3. *(Color online) Effective memory bandwidth β_-/τ of five SpMV kernels on the Kepler K20M GPU for selected sparse matrices in double precision representation. The matrices are ordered according to μ.*

by τ. We considered two extreme cases: the input vector either is not cached or is fully cached. The bytes transferred in each case, β_- and β_+, respectively, are calculated from

$$\beta_- = (2s_\mathrm{v} + s_\mathrm{i}) * \mathtt{nnz} + s_\mathrm{i} * \mathtt{strips} + s_\mathrm{v} * \mathtt{rows},$$
(6.1)
$$\beta_+ = (s_\mathrm{v} + s_\mathrm{i}) * \mathtt{nnz} + s_\mathrm{i} * \mathtt{strips} + 2s_\mathrm{v} * \mathtt{rows},$$

where s_v is the size (in bytes) of data entries in $\mathtt{val}$ and $s_\mathrm{i} = \mathtt{sizeof(int)} = 4$. The expression for β_+ is the number of bytes necessary to store the matrix and the input and output vectors. The value of β_- exceeds β_+ by $s_\mathrm{v}(\mathtt{nnz} - \mathtt{rows})$, for if the cache is absent, reading elements of the input vector requires $\mathtt{nnz}$ rather than $\mathtt{rows}$ transfers of the input vector components.

The bandwidth can be defined either as β_-/τ or β_+/τ, which leads to two definitions of memory utilization efficiency, $\eta_\pm$:

$$\eta_\pm = \frac{\beta_\pm}{\tau} \frac{1}{B},$$
(6.2)

where B is the theoretical hardware bandwidth of the device. Clearly, $\eta_+ < 1$ and a value of $\eta_- \geq 1$ indicates that the device efficiently buffers the input vector in its caches.

6.5. SpMV multiplication results. The memory bandwidth β_-/τ for five SpMV kernels running on the Kepler K20M GPU for selected matrices (double precision) is presented in Figure 3. The matrices in this figure include all square matrices used in [3]. They were ordered according to the average row length, μ, which ranges from 1 (matrix $\mathtt{p7}$) to ≈ 484 ($\mathtt{tsopf\text{-}rs\text{-}b2383}$). In this set, matrix $\mathtt{p7}$ constitutes an extreme case in which accesses to the input vector are totally uncoalesced. Moreover, since in this case the CMRS algorithm uses the value of $\mathtt{height} = 4$, only 4 of the 32 threads in a warp are actively processing the matrix elements. This leads to very inefficient memory bandwidth utilization, with $\eta_+ = \eta_- \approx 0.09$. Matrix $\mathtt{tsopf\text{-}rs\text{-}b2383}$
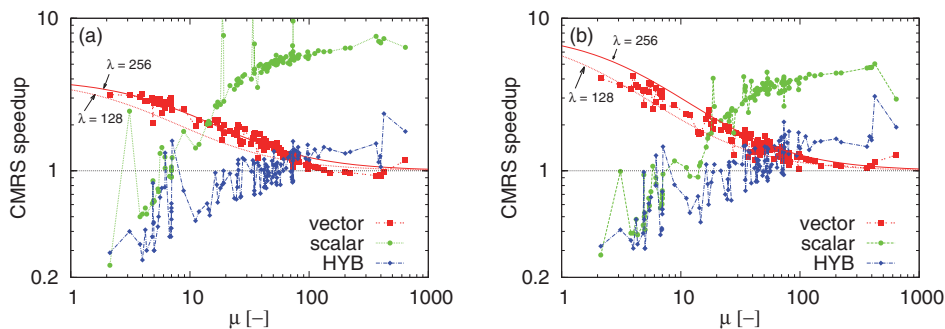
FIG. 4. *(Color online) Performance of the CMRS SpMV kernel relative to three other kernels for the test matrices as a function of the mean row length, μ, for* (a) *K20M and* (b) *GTX 480 in double precision. The smooth lines were computed from* (5.2) *with $\lambda = 128, 256$.*

constitutes another extreme case in which accesses to the input vector are well co-alesced. The memory bandwidth attained for this matrix by the CMRS kernel is 250 GB/s, which yields $\eta_- = 1.22$ and $\eta_+ = 0.73$. The fact that $\eta_- > 1$ indicates that K20M can efficiently buffer the input vector in its caches. However, the efficiency of the previous generation GPU, GTX 480, turned out to be even better for this matrix ($\eta_- = 1.40$, $\eta_+ = 0.80$, 248 GB/s), even though GTX 480 has a smaller L2 cache, has no 48 KB read-only cache, and is 7 times slower at double precision arithmetic (cf. Table 1). Note that pre-Fermi GPUs, e.g., GTX 285, which had neither L1 nor L2 caches, allowed for far less efficient data caching ($\eta_- \leq 1.08$) [3].

To compare different SpMV algorithms, we analyzed the results obtained for the sparse matrices from UF SMC, assuming that this collection contains a representative sample of sparse matrices. The speedup of our implementation over three other SpMV kernels, vector, scalar, and hybrid, for K20M (double precision) and GTX 480 (single precision) is shown in Figure 4. Results for K20M in double precision (`height` = 4) are interesting from a practical point of view, whereas the results for GTX 480 in single precision (`height` = 12) allow us to estimate to what extent the performance of the CMRS kernel is affected by the kernel occupancy. Clearly, μ turns out to be a relevant parameter for determining relative performance of various SpMV implementations. However, perhaps an even more striking feature of the two graphs is their similarity, which reflects the fact that the performance of SpMV kernels is highly influenced by the matrix structure. The scalar and hybrid kernels give the shortest SpMV times for small μ, but their efficiency decreases as μ is increased, with the scalar kernel being very inefficient for large μ. This is related to the inability of the scalar kernel to coalesce data transfers if μ is large. The vector kernel behaves in just the opposite way: its efficiency relative to other kernels is very good for large μ, but it decreases as μ drops below ≈ 100, as predicted by (5.2).

The smooth lines in Figure 4 show the speedup of the CMRS kernel over the vector kernel, as predicted by (5.2), for the data in double precision ($b = 8$). We used two values of λ, 128 (as suggested by Nvidia for accessing contiguous streams of 4-byte data [19]) and 256. For K20M the agreement is very good for $\lambda = 256$, whereas for the older architecture the experimental values appear to lie between the two theoretical curves. The superiority of $\lambda = 256$ for 8-byte data on the Kepler architecture will be also discussed in section 6.7.

To better validate the performance model of the CMRS format, in Figure 5 we
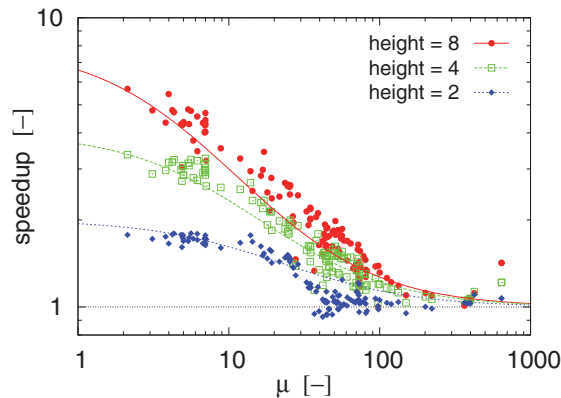
FIG. 5. *(Color online) The speedup of the CMRS SpMV kernel over the vector kernel as a function of the mean row length, μ, for GTX 480 in single precision and* `height` $= 2, 4, 8$ *(symbols) and the performance model predictions,* (5.2) *with $b = 4, \lambda = 128$ (lines).*
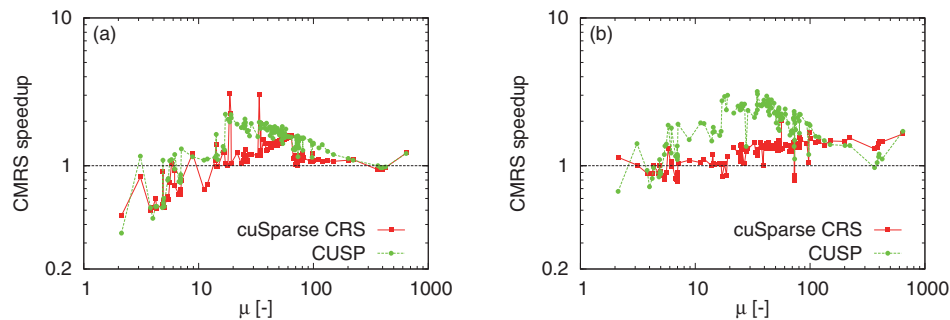

FIG. 6. *(Color online) The speedup of the CMRS SpMV kernel over the cuSparse CRS and CUSP CRS-tex kernels as a function of the mean row length, μ, for* (a) *K20M in double precision and* (b) *GTX 480 in single precision.*

compare its predictions for the 4-byte data with the results obtained for GTX 480 in single precision. With this choice of the matrix value representation and the GPU architecture, the device runs at full occupancy up to `height` $= 8$. As can be seen, the model describes the actual speedup well.

The speedup of our CMRS SpMV kernel over two remaining CRS-based SpMV kernels, cuSparse and CUSP, is shown in Figure 6. Again we show only the data for the extreme cases of K20M in double precision (left panel) and GTX 480 in single precision (right panel). The cuSparse implementation turns out to be better optimized than CUSP, and our CMRS SpMV kernel outperforms each of them for sufficiently large values of μ.

6.6. Comparison of the computational efficiency of different SpMV kernels. To compare the computational efficiency of different SpMV kernels, we adopt a convention that implementation A is significantly faster than B if and only if its execution time is at least 10% shorter.

We found only one matrix (`kktpower`) for which the cuSparse 5.5 CRS is significantly faster than any other SpMV kernel considered here. Although for 26 matrices this kernel turns out to be significantly faster than our CMRS implementation, each
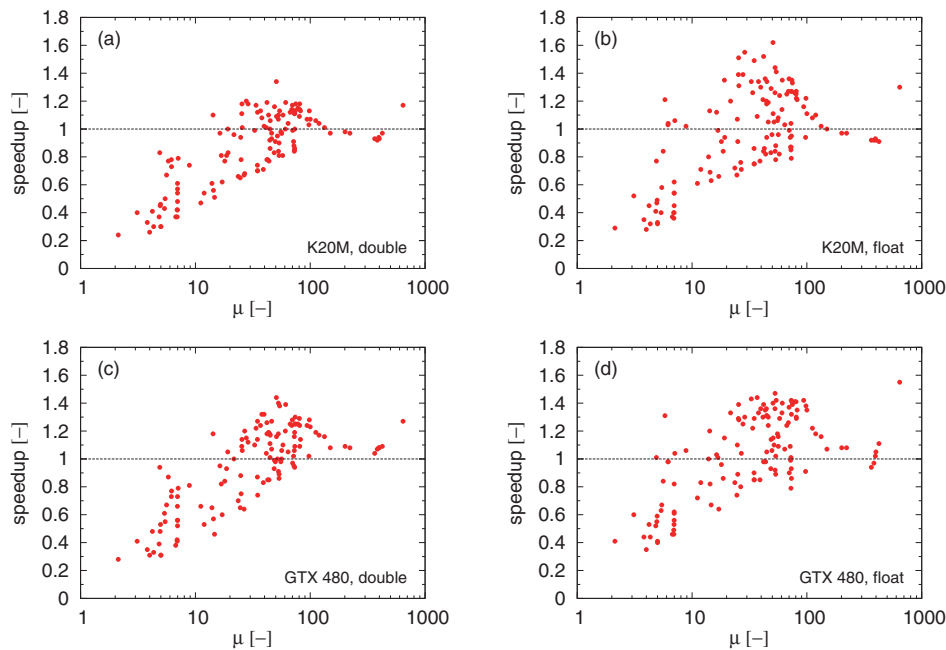
FIG. 7. *(Color online) The speedup of our CMRS implementation over the best of all five alternative SpMV kernels as a function of the nonzero matrix elements per row (μ), for K20M and GTX 480, in double and single precision.*

of these matrices is characterized by a low number of nonzero elements per row, and for such matrices the HYB kernel is usually even faster. A similar situation is observed for the scalar kernel, which is significantly faster than any other SpMV kernel for only one matrix (`asia_osm`). The CUSP implementation is generally even less efficient than cuSparse. We found no matrix for which the vector kernel is significantly faster than any other kernel. Similar results were obtained for GTX 480 as well as for calculations in single precision.

The most interesting is comparison of our algorithm with the cuSparse 5.5 HYB implementation. We found HYB to be significantly faster than any implementation (our implementation) for 55 (58) matrices. On the other hand, our implementation is significantly faster than any other (HYB) implementation for 29 (46) matrices. This is a good result, especially if one takes into account that the HYB implementation analyzes the matrix structure and transforms it (e.g., by zero-padding) accordingly before the first SpMV routine can be called on it. In section 6.7 we shall examine how techniques like zero-padding could be used to further optimize the CMRS SpMV kernel. Note also that the currently available implementation of the HYB format has rather high memory requirements. For this reason the HYB implementation could not be run on GTX 480 for 14 of the largest matrices in double precision.

From Figure 4 it can be immediately seen that our CMRS implementation generally does not yield much improvement over the vector implementation for $\mu \gtrsim 150$, and tends to be systematically slower than HYB for $\mu \lesssim 20$. Hence one expects that the advantages of CMRS will be most pronounced for moderate values of μ. This is confirmed by Figure 7, which depicts the speedup of our CMRS SpMV implementa-
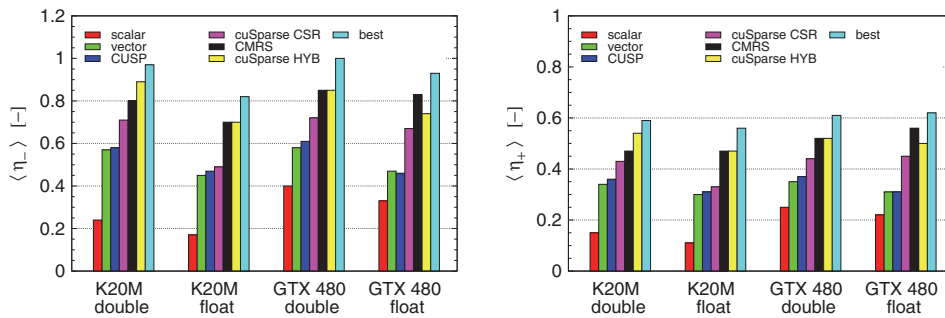
FIG. 8. *(Color online) The average memory utilization efficiency, $\langle\eta_-\rangle$ (left) and $\langle\eta_+\rangle$ (right), as defined in (6.2), for all tested SpMV kernels, GPU devices, and matrix value representations, as a function of μ. Label "best" denotes the results for the most efficient of all 6 SpMV kernels considered in this study, selected individually for each matrix.*

tion against the best of all five alternative SpMV implementations considered here, calculated individually for each matrix, for the K20M and GTX 480 GPUs running in double and single precision modes. A striking similarity of the results obtained for different architectures and different matrix value representations indicates that the efficiency of an SpMV kernel depends mainly on the matrix structure. It is also clear that the efficiency of our CMRS implementation in the most important case of the Kepler architecture (K20M) in double precision is $\approx 10\%$ worse than for the Fermi architecture (GTX 480). In particular, the largest speedup for K20M and GTX 480 is 34% and 44%, respectively. We believe this is an effect of the bandwidth-occupancy relation in GPUs, as discussed in section 6.3. As might be expected, the CMRS format allows for even better acceleration of the SpMV kernel if the calculations are performed in single precision. The maximum speedup is $\approx 62\%$ for K20M and $\approx 55\%$ for GTX 480, even though in the former case we used a smaller value of the CMRS strip height. We attribute this to the fact that the cuSparse 5.5 CSR kernel is apparently not well optimized for the Kepler architecture in single precision (data not shown).

Since the SpMV operation is memory-bound, efficiency of various implementations of this kernel can be compared using the memory utilization efficiency parameters $\eta_\pm$, (6.2). The results for all tested SpMV kernels, GPU devices, and matrix value representations, averaged over all tested matrices, are shown in Figure 8. We also included the results for a hypothetical kernel, denoted as "best," in which the most optimal kernel is selected for a given sparse matrix (in practice, this choice is limited to choosing between HYB and CMRS). These results confirm that the scalar kernel is very inefficient as a general-purpose SpMV kernel, especially in the newer (Kepler) architecture. Optimization of the cuSparse 5.5 CSR kernel appears to be unsatisfactory for single precision arithmetic on K20M. The best results, on average, are obtained for the cuSparse 5.5 HYB and our CMRS implementation. Note that if one used the best kernel for a given matrix, $\langle\eta_-\rangle$ would rise to ≈ 1 for double precision arithmetic on both Fermi and Kepler architectures, which is a very good result.

Since the value of η_+ is bounded from above by 1, its value carries valuable information about the extent to which a kernel utilizes the hardware. Its mean value for the best kernel is ≈ 0.6 for both architectures, which again should be considered a very good result. Its value for individual matrices varies from ≈ 0.13 for permutation

matrices to ≈ 0.80 for the `af_shell10` sparse matrix and can be as high as ≈ 0.86 for a dense $10\,000 \times 10\,000$ matrix treated as a sparse one. A value of η_+ much smaller than 1 might be used as an indicator that a significant performance boost could probably be achieved through reordering of the matrix rows.

6.7. Matrix transformation for better performance. The basic CMRS format, as defined in section 3, allows for a quick and straightforward conversion to and from the CRS format without any memory overhead. Can we relax these two conditions to allow for an even faster SpMV kernel?

A major issue with the implementation presented in section 4 is that it requires each warp to reserve a `WARP_SIZE`×`HEIGHT` data array in the shared memory, of which only `WARP_SIZE` elements are utilized simultaneously. In many cases most of the shared memory may never be used by the warp that controls it. The central problem is, however, that reducing the size of per-warp buffers in the shared memory would allow one to increase the value of `height`, which, following (5.2), should result in a significant kernel performance boost.

This problem can be dealt with by changing the structure of the sparse matrix. Here we briefly examine one such approach. In Algorithm 1 the buffer is accessed always through the same pattern: `buf[thread_lane, r]`. If we could replace it with `buf[thread_lane mod M, r]`, where $M < $ `WARP_SIZE`, the size of each per-warp buffer could be reduced to `M`×`HEIGHT`, i.e., by a factor of `WARP_SIZE`/`M`. This will work provided that no threads in a warp can access the same buffer location simultaneously. In most cases this condition can be met by taking advantage of the fact that the CMRS format permits one to reorder the items in a strip arbitrarily: it suffices to arrange the items in such a way that each warp processes at most `M` items from a given row `r`, and these items are stored contiguously in the array. Such an arrangement ensures that if the value of the row identifier `r` in `buf[thread_lane mod M, r]` is the same for some threads, the values of the first indices into the buffer are different. Such arrangement can be, however, impossible for some sparse matrices with highly variable row lengths, especially for small values of `M`. In such cases the matrices must be filled with explicit zeros, which modifies the structure of the matrix internal representation.

We examined numerically the case `HEIGHT` $= 16$ and `M` $= 8$, which requires the same buffer size as in the implementation analyzed in the previous section for K20M and double precision, but is characterized by a 4-fold larger value of the strip height. Since the value of `HEIGHT` is now relatively high so that each strip contains hundreds or even thousands of matrix elements, we also applied another optimization: all strips were padded with zeros to ensure the number of matrix items they contain is a multiple of `WARP_SIZE`. In this way all memory accesses to arrays `Val` and `ColInd` are fully coalesced. This comes at the cost of an additional modification of the internal matrix representation, which in some cases may result in a noticeable memory overhead. For 4 matrices, the implementation considered here turned out to be significantly slower than that defined in section 4, and we excluded them from further analysis.

The results are shown in Figure 9 (a), which corresponds directly to Figure 7 (a). Clearly, padding the CMRS matrix with explicit zeros can result in a considerable shortening of the SpMV execution time, especially for small values of μ, as expected from (5.2). For example, the new algorithm turned out to be 3 times faster than the implementation presented in section 4 for matrix `mc2depi`. Our modified implementation is significantly faster than any of five alternative SpMV kernels for 58 matrices, with the greatest relative speedup reaching 1.63 for the matrix `mac_econ_fwd500`, and significantly slower than an alternative solution for only 13 cases, the worst case being
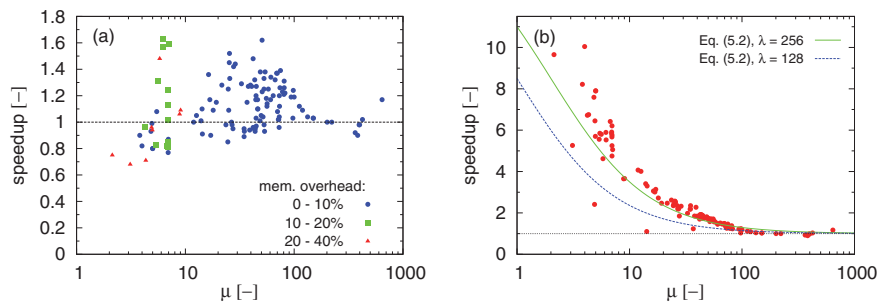
FIG. 9. *(Color online) The speedup of the modified CMRS SpMV kernel over* (a) *the best of all five alternative SpMV kernels and* (b) *the vector kernel as a function of the nonzero matrix elements per row (μ), for K20M in double precision. Different symbols in panel* (a) *represent different levels of memory overhead related to padding the internal matrix representation with explicit zeros.*

`webbase-1M`, for which the execution time relative to the HYB kernel is 0.68. The average memory efficiency is $\eta_- \approx 1.0$ and $\eta_+ \approx 0.6$; cf. Figure 8. Note that the zero-padding introduced a significant memory overhead only for some matrices with $\mu \lesssim 10$. Thus, the main advantage of CMRS over HYB is that it allows for an efficient single-kernel implementation without significant memory overhead. In contrast to this, HYB attempts to strike a balance between a format that is computationally efficient but often requires a prohibitive memory overhead (ELL) and a format that imposes no memory overhead but is computationally inefficient (COO).

Figure 7 (b) shows the speedup of the modified algorithm over the vector kernel for sparse matrices in double precision on K20M and compares it with the model (5.2) with two values of $\lambda = 128, 256$. Clearly, the fit is much better for $\lambda = 256$.

6.8. Potential for further performance tuning. Performance of many SpMV kernels can be significantly improved by adjusting kernel optimization parameters to both the structure of the matrix and the hardware on which the kernel is to be executed [5, 14, 25, 26]. For example, Vuduc [26] showed an up to 4-fold acceleration for modern cache-based superscalar machines. However, finding optimal optimization parameters is usually costly and hence is often performed "off-line." In particular, Choi et al. [5] developed efficient autotuning techniques for their BELLPACK sparse matrix format and tested it on pre-Fermi GPUs. However, unlike BELLPACK, CMRS does not use explicit storage of dense blocks to compress the data structure and hence requires a different optimization strategy.

By comparing the default CMRS kernel times with those obtained for the same kernel launched with the optimal parameters determined by the brute-force search, we found that while the tuning of the default CMRS parameters is possible, it is not expected to give a spectacular performance boost. For example, the CMRS parameters could be tuned to speed up the kernel by at least 10% for only 14 (16) test matrices on K20M (GTX 480) in double precision, and the maximum acceleration was 18% and 36% for K20M and GTX 480, respectively. Our preliminary results presented in the previous section indicate that better results can be obtained by modifying the internal structure of the sparse matrix, e.g., by a suitable zero-padding. Further research on this issue is necessary.

7. Conclusions and outlook. The CMRS format is designed specifically for optimizing the SpMV operation on modern graphics processing units. It has several

features distinguishing it from other formats developed for the same purpose: (i) it is an extension of a popular format, CRS, with a quick and in-place conversion to and from it; (ii) it has an efficient, single-kernel implementation; (iii) it allows for dynamic assignment of threads to the matrix rows; (iv) it does not *require* zero-padding, row reordering, or any other matrix transformations for good performance; and (v) it has a great potential for off-line optimization techniques, including zero-padding and row reordering, which improve its efficiency for extremely sparse matrices and can turn it into one of the most efficient SpMV formats for GPUs. Property (iii) distinguishes CMRS from other derivatives of the CRS format in which a warp processes more than one matrix row, while properties (ii) and (v) distinguish it from the HYB format. These features should facilitate its adoption in existing software and open new possibilities for its further optimization. Moreover, the fact that our CMRS-based implementation of the SpMV kernel often approaches the hardware limit suggests that this format will scale well into future GPU architectures.

The performance model of the CMRS SpMV kernel, despite its simplicity, turns out to fit the actual results of numerical experiments well. This indicates that the structure of typical sparse matrices from the UF SMC is at least partially ordered—otherwise the SpMV efficiency would be determined by indirect addressing of the input vector, a factor completely neglected in the model. The model explains the acceleration of the CMRS over the standard vector kernel. It also identifies the mean number of nonzero elements per row (μ) as a relevant parameter for the CRS-based SpMV kernels on GPUs. The bandwidth efficiency η_- (or η_+) can be used to identify matrices for which further optimization is required as well as help determine the quality of hardware support for the SpMV operation.

Our current implementations of the CMRS kernel are not without limitations. The column index is stored on only 28 bits, which might prove insufficient for future devices with larger amounts of memory. However, applications usually store much more data than just a single sparse matrix. For example, a GPU-based computational fluid dynamics solver may require ≈ 500 bytes of storage per each column of several of its sparse matrices [21], which corresponds to the memory threshold at $2^{28} \times 500$ B ≈ 130 GB, far above the 12 GB available in modern accelerators. Consequently, compression of the column index should not become a serious problem very soon. A much more serious problem is related to the fact that CMRS is inherently limited by the amount of the shared memory per multiprocessor. While in most cases this can be circumvented by a suitable matrix transformation, as explained in section 6.7, its efficient usage on Kepler-class GPUs for matrices with values occupying more than 8 bytes, e.g., double precision complex numbers, may be problematic. It is also not clear whether CMRS can be efficiently implemented on architectures lacking a programmable, on-chip shared memory buffer. Moreover, CMRS requires the matrix to be sufficiently large. Further research is also required to find the best "off-line" CMRS matrix optimization strategy—while our preliminary results with zero-padding are very encouraging, our approach is rather complex and is not universal.

Finally, our results reveal the importance of developing a representative collection (or collections) of sparse matrices for which the SpMV product is a truly relevant operation. The UF SMC contains some very unusual matrices for which the SpMV product is unlikely to be applicable, e.g., matrices with empty rows or columns. Such atypical matrices can obfuscate the general picture of the SpMV performance and its dependence on the matrix format and techniques used to implement it.

REFERENCES

[1] M. M. Baskaran and R. Bordawekar, *Optimizing Sparse Matrix-Vector Multiplication on GPUs*, Tech. Report RC24704, IMB Research, 2008.

[2] N. Bell and M. Garland, *Efficient Sparse Matrix-Vector Multiplication on CUDA*, NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, 2008.

[3] 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, ACM, New York, 2009, pp. 1–11.

[4] N. Bell and M. Garland, *CUSP: A C++ Templated Sparse Matrix Library*, Version 0.3.0, 2012.

[5] J. W. Choi, A. Singh, and R. W. Vuduc, *Model-driven autotuning of sparse matrix-vector multiply on GPUs*, SIGPLAN Not., 45 (2010), pp. 115–126.

[6] T. A. Davis and Y. Hu, *The university of Florida sparse matrix collection*, ACM Trans. Math. Softw., 38 (2011), pp. 1–25.

[7] D. DuBois, A. DuBois, T. Boorman, C. Connor, and S. Poole, *An Implementation of the Conjugate Gradient Algorithm on FPGAs*, in FCCM '08: Proceedings of the 2008 16th International Symposium on Field-Programmable Custom Computing Machines, IEEE Computer Society, Washington, DC, 2008, pp. 296–297.

[8] D. DuBois, A. DuBois, C. Connor, and S. Poole, *Sparse matrix-vector multiplication on a reconfigurable supercomputer*, in Proceedings of the 16th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, IEEE Computer Society, 2008, pp. 239–247.

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

[10] R. Farber, *CUDA Application Design and Development*, Morgan Kaufmann, 2011.

[11] X. Feng, H. Jin, R. Zheng, K. Hu, J. Zeng, and Z. Shao, *Optimization of sparse matrix-vector multiplication with variant CSR on GPUs*, in Proceedings of the 17th International IEEE Conference on Parallel and Distributed Systems (ICPADS), IEEE Computer Society, 2011, pp. 165–172.

[12] M. Garland, *Sparse matrix computations on manycore GPUs*, in DAC '08: Proceedings of the 45th Annual Conference on Design Automation, ACM, New York, 2008, pp. 2–6.

[13] D. Grewe and A. Lokhmotov, *Automatically generating and tuning GPU code for sparse matrix-vector multiplication from a high-level representation*, in GPGPU-4, Proceedings of the 4th Workshop on General Purpose Processing on Graphics Processing Units (GPGPU2011), ACM, New York, 2011, p. 12.

[14] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, *Efficient sparse matrix-vector multiplication on x86-based many-core processors*, in Proceedings of the 27th ACM International Conference on Supercomputing (ICS '13), ACM, New York, 2013, pp. 273–282.

[15] M. Maggioni and T. Berger-Wolf, *An architecture-aware technique for optimizing sparse matrix-vector multiplication on GPUs*, Procedia Comput. Sci., 18 (2013), pp. 329–338.

[16] Z. Malecha, Ł. Mirosław, T. Tomczak, Z. Koza, M. Matyka, W. Tarnawski, and D. Szczerba, *GPU-based simulation of 3D blood flow in abdominal aorta using OpenFOAM*, Arch. Mech., 63 (2011), pp. 137–161.

[17] A. Monakov, A. Lokhmotov, and A. Avetisyan, *Automatically tuning sparse matrix-vector multiplication for GPU architectures*, in High Performance Embedded Architectures and Compilers, Y. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, and X. Martorell, eds., Lecture Notes in Comput. Sci. 5952, Springer-Berlin, Heidelberg, 2010, pp. 111–125.

[18] D. Mukunoki and D. Takahashi, *Optimization of sparse matrix-vector multiplication for CRS format on NVIDIA Kepler architecture GPUs*, in Computational Science and Its Applications – ICCSA 2013, B. Murgante, S. Misra, M. Carlini, C. M. Torre, H.-Q. Nguyen, D. Taniar, B. O. Apduhan, and O. Gervasi, eds., Lecture Notes in Comput. Sci. 7975, Springer-Berlin, Heidelberg, 2013, pp. 211–223.

[19] NVIDIA, *CUDA C Programming Guide Version* 5.5, 2013.

[20] E. Saule, K. Kaya, and Ü. V. Çatalyürek, *Performance Evaluation of Sparse Matrix Multiplication Kernels on Intel Xeon Phi*, preprint, arXiv:1302.1078 [cs.PF], 2013.

[21] T. Tomczak, K. Zadarnowska, Z. Koza, M. Matyka, and Ł. Mirosław, *Acceleration of iterative Navier-Stokes solvers on graphics processing units*, Int. J. Comput. Fluid Dyn., 27 (2013), pp. 201–209.

[22] F. Vázquez, J. J. Fernández, and E. M. Garzón, *Automatic tuning of the sparse matrix vector product on GPUs based on the ELLR-T approach*, Parallel Comput., 38 (2012), pp. 408–420.

[23] F. Vázquez, E. M. Garzón, J. A. Martinez, and J. J. Fernández, *Accelerating sparse matrix vector product with GPUs*, in Proceedings of the International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE 2009), CMMSE, 2009, pp. 1081–1092.

[24] F. Vázquez, G. Ortega, J. J. Fernández, and E. M. Garzón, *Improving the performance of the sparse matrix vector product with GPUs*, in Proceedings of the 10th IEEE International Conference on Computer and Information Technology (CIT 2010), IEEE Computer Society, 2010, pp. 1146–1151.

[25] R. Vuduc, J. W. Demmel, and K. A. Yelick, *Oski: A library of automatically tuned sparse matrix kernels*, J. Phys. Conference Ser., 16 (2005), p. 521.

[26] R. W. Vuduc, *Automatic Performance Tuning of Sparse Matrix Kernels*, Ph.D. thesis, University of California, Berkeley, 2003.

[27] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, *Optimization of sparse matrix-vector multiplication on emerging multicore platforms*, in SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, ACM, New York, 2007, pp. 1–12.

[28] X. Yang, S. Parthasarathy, and P. Sadayappan, *Fast sparse matrix-vector multiplication on GPUs: Implications for graph mining*, Proc. VLDB Endowment, 4 (2011), pp. 231–242.

[29] H. Yoshizawa and D. Takahashi, *Automatic tuning of sparse matrix-vector multiplication for CRS format on GPUs*, in Proceedings of the IEEE 15th International Conference on Computational Science and Engineering, 2012, pp. 130–136.