Filippone S, Cardellini V, Barbieri D, Fanfarillo A, Sparse matrix-vector multiplication on GPGPUs,
ACM Transactions on Mathematical Software, Volume 43, Issue 4 March 2017, Article 30.

Sparse Matrix-Vector Multiplication on GPGPUs

SALVATORE FILIPPONE VALERIA CARDELLINI
Cranfield University

DAVIDE BARBIERI
ALESSANDRO FANFARILLO
Università degli Studi di Roma "Tor Vergata"

Abstract

The multiplication of a sparse matrix by a dense vector (SpMV) is a centerpiece of scientific computing applications: it is the essential kernel for the solution of sparse linear systems and sparse eigenvalue problems by iterative methods. The efficient implementation of the sparse matrix-vector multiplication is therefore crucial and has been the subject of an immense amount of research, with interest renewed with every major new trend in high performance computing architectures. The introduction of General Purpose Graphics Processing Units (GPGPUs) is no exception, and many articles have been devoted to this problem.

With this paper we provide a review of the techniques for implementing the SpMV kernel on GPGPUs that have appeared in the literature of the last few years. We discuss the issues and tradeoffs that have been encountered by the various researchers, and a list of solutions, organized in categories according to common features. We also provide a performance comparison across different GPGPU models and on a set of test matrices coming from various application domains.

Categories and subject descriptors:D.2.11[Software Engineering]: Software Architectures—*Data abstraction*; G.1.3[Numerical Analysis]: Numerical Linear Algebra—*Sparse, structured, and very large systems (direct and iterative methods)*; G.4[Mathematical Software]: Algorithm design and analysis.

General terms:Mathematics of computing, Algorithms, Design
Additional Key Words:Sparse Matrices, GPU programming.

Author's addresses: School of Aerospace, Transport and Manufacturing, Cranfield University, Cranfield, MK43 0AL, United Kingdom, e-mail address: salvatore.filippone@cranfield.ac.uk; Dipartimento di Ingegneria Civile e Ingegneria Informatica, Università degli Studi di Roma "Tor Vergata", Via del Politecnico 1, 00133 Roma, Italy, e-mail addresses: cardellini@ing.uniroma2.it, fanfarillo@ing.uniroma2.it.

1 Introduction

The topic we are about to discuss is a single, apparently very simple, computational kernel: the multiplication of a vector by a sparse matrix. This opening statement begs the obvious question of what we mean by sparse matrix. The most famous definition of "sparse matrix" is attributed to James Wilkinson, one of the founding fathers of numerical linear algebra [28]:

> Any matrix with enough zeros that it pays to take advantage of them.

Denoting by NZ the number of nonzero coefficients, we then have that an $M \times N$ matrix is sparse when

$$NZ \ll M \times N.$$

Indeed, it is quite common for $A \in \mathbb{R}^{n \times n}$ to have a number of nonzeros that is $O(n)$, that is, the average number of nonzero elements per row (column) is bounded independently of the total number of rows (columns).

Sparse matrices are a main staple of scientific computations; indeed, they are one of the "Seven Dwarfs", a set of numerical methods essential to computational science and engineering [23, 102, 35] that have driven the development of software for high performance computing environments over the years.

Most problems of mathematical physics require the approximate solution of differential equations; to this end, the equations have to be transformed into algebraic equations, or *discretized*. A general feature of most discretization methods, including finite differences, finite elements, and finite volumes [62, 86, 84], is that the number of entries in each discretized equation depends on local topological features of the discretization, and not on the global domain size. Thus, many problems deal with sparse matrices; far from being unusual, sparse matrices are extremely common in scientific computing, and the related techniques are extremely important.

The fact that the discretized linear systems are sparse affects the techniques used for their solution. The usual factorization methods found in LAPACK and similar libraries [4, 21] normally destroy the sparsity structure of a matrix by introducing *fill-in*, that is, new nonzero entries in addition to those already present in the original matrix; thus, memory requirements grow, and the matrices become unmanageable.

The main alternative is to use an iterative solver, and currently the most popular ones are those based on Krylov subspace projection methods [88, 53, 42]. From a software point of view, all Krylov methods employ the matrix A only to perform matrix-vector products $y \leftarrow Ax$, hence they do *not* alter the nonzero structure and memory requirements, and they require an efficient implementation of the matrix-vector product.

The optimization of the Sparse Matrix-Vector multiplication (SpMV) presents significant challenges in any computer architecture. The SpMV kernel is well-known to be a memory bounded application; and its bandwidth usage is strongly dependent on both the input matrix and on the underlying computing platform(s). The history of its efficient implementations is largely a story of data

structures and of their match (or lack thereof) to the architecture of the computers employed to run the iterative solver codes. Many research efforts have been devoted to managing the complexity of multiple data storage formats; among them we refer the reader to [33] and [39].

General Purpose Graphics Processing Units (GPGPUs) [69] are today an established and attractive choice in the world of scientific computing, found in many among the fastest supercomputers on the Top 500 list, and even being offered as a Cloud service, e.g., Amazon EC2. The GPGPU cards produced by NVIDIA are today among the most popular computing platforms; their architectural model is based on a scalable array of multi-threaded streaming multi-processors, each composed by a fixed number of scalar processors, a set of dual-issue instruction fetch units, on-chip fast memory, plus additional special-function hardware. A large variety of complex algorithms can indeed exploit GPGPUs and gain significant performance benefits (e.g., [50, 9, 99]). For graphics cards produced by NVIDIA, the programming model of choice is CUDA [81, 89]; a CUDA program consists of a host program that runs on the CPU host, and a kernel program that executes on the GPU itself. The host program typically sets up the data and transfers it to and from the GPU/GPGPU, while the kernel program performs the main processing tasks.

GPGPUs have a Single Instruction Multiple Threads (SIMT) architecture, meaning that the same instruction is scheduled across different threads; thus they implement in a natural way the Single Instruction Multiple Data (SIMD) programming paradigm. GPGPUs appear good candidates for scientific computing applications, and therefore they have attracted much interest for operations on sparse matrices, such as the matrix-vector multiplication; many researchers have taken interest in the SpMV kernel, as witnessed for example by the works [11, 14, 22, 27, 56, 78, 85, 96], and the development of CUSP [24] and NVIDIA's cuSPARSE [82] libraries.

Implementation of SpMV on computing platforms such as GPUs is certainly no exception to the general trends we have mentioned. The main issue with the SpMV kernel on GPGPUs is the (mis)match between the SIMD architecture and the irregular data access pattern of many sparse matrices; hence, the development of this kernel revolves around devising data structures acting as "adapters".

In the rest of this paper we provide an extensive review of storage formats employed on modern GPGPUs that have appeared in the literature in recent years, including some of our own variations on the theme. We also present a performance comparison of some publicly available (or kindly provided by their authors) storage formats across different GPU models and on a set of test matrices coming from various application domains.

We begin in Section 2 by presenting an overview of traditional sparse storage formats, detailing their advantages and disadvantages in the context of the SpMV kernel. In Section 3 we briefly introduce some of the characteristics of the GPGPU computing platform. This prepares the ground for the overview of recent efforts in the GPGPU programming world in Section 4, giving the reader an up-to-date picture of the state of the art, including the variations on existing

storage formats that we have devised in the framework of PSBLAS [39, 19]. Section 5 provides a comparison of the various formats on different GPU models, highlighting the complex and sometimes surprising interactions between sparse matrix structures, data structures and architectures. Section 6 closes the paper and outlines future work.

2 Storage Formats for Sparse Matrices

Let us return to the sparse matrix definition by Wilkinson:

> Any matrix with enough zeros that it pays to take advantage of them.

This definition implicitly refers to some operation in the context of which we are "taking advantage" of the zeros; experience shows that it is impossible to exploit the matrix structure in a way that is uniformly good across multiple operators, let alone multiple computing architectures. It is therefore desirable to have a flexible framework that allows to switch among different formats as needed; these ideas are further explored in [19, 39].

The usual two-dimensional array storage is a linear mapping that stores the coefficient $A(I, J)$ of an $M \times N$ matrix at position $(J-1) \times M + I$ of a linear array. This formula assumes column-major ordering used in Fortran and Matlab; an analogous formula applies for row-major storage used in C and Java. Thus representing a matrix in memory requires just one linear array, and two integer values detailing the size of the matrix.

Now enter sparse matrices: "taking advantage" of the zeros essentially means avoiding their explicit storage. But this means that the simple mapping between the index pair (I, J) and the position of the coefficient in memory is destroyed. Therefore, all sparse matrix storage formats are devised around means of rebuilding this map using auxiliary index information: a pair of dimensions does not suffice any longer. How costly this rebuilding is in the context of the operations we want to perform is the critical issue we need to investigate. Indeed, performance of sparse matrix kernels is typically much less than that of their dense counterparts, precisely because of the need to retrieve index information and the associated memory traffic. Moreover, whereas normal storage formats allow for sequential and/or blocked accesses to memory in the input and output vectors x and y, sparse storage means that coefficients stored in adjacent positions in the sparse matrix may operate on vector entries that are quite far apart, depending on the *pattern* of nonzeros contained in the matrix.

By now it should be clear that the performance of sparse matrix computations depends critically on the specific representation chosen. Multiple factors contribute to determine the overall performance:

- the match between the data structure and the underlying computing architecture, including the possibility of exploiting special hardware instructions;

- the suitability of the data structure to decomposition into independent, load-balanced work units;
- the amount of overhead due to the explicit storage of indices;
- the amount of padding with explicit zeros that may be necessary;
- the interaction between the data structure and the distribution of nonzeros (pattern) within the sparse matrix;
- the relation between the sparsity pattern and the sequence of memory accesses especially into the x vector.

Many storage formats have been invented over the years; a number of attempts have also been directed at standardizing the interface to these data formats for convenient usage (see e.g., [33]).

We will now review three very simple and widely-used data formats: CO-Ordinate (COO), Compressed Sparse Rows (CSR), and Compressed Sparse Columns (CSC). These three formats are probably the closest we can get to a "general purpose" sparse matrix representation. We will describe each storage format and the related algorithms in a pseudo-Matlab notation, with 1-base indexing, which can be readily translated into either Fortran or C. For each format, we will show the representation of the example matrix shown in Figure 1. Throughout the paper we will refer to the notation introduced in Table 1.

Table 1: Notation for parameters describing a sparse matrix

Name	Description
M	Number of rows in matrix
N	Number of columns in matrix
NZ	Number of nonzeros in matrix
AVGNZR	Average number of nonzeros per row
MAXNZR	Maximum number of nonzeros per row
NDIAG	Number of nonzero diagonals
AS	Coefficients array
IA	Row indices array
JA	Column indices array
IRP	Row start pointers array
JCP	Column start pointers array
NZR	Number of nonzeros per row array
OFFSET	Offset for diagonals

2.1 COOrdinate

The COO format is a particularly simple storage scheme, defined by the three scalars `M`, `N`, `NZ` and the three arrays `IA`, `JA` and `AS`. By definition of number of rows we have $1 \leq IA(i) \leq M$, and likewise for the columns; a graphical description is given in Figure 2.

The code to compute the matrix-vector product $y = Ax$ is shown in Alg. 2.1; it costs five memory reads, one memory write and two floating-point operations

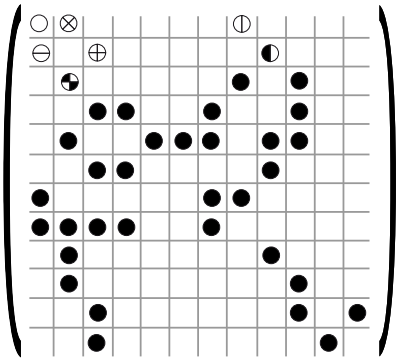


Figure 1: Example of sparse matrix

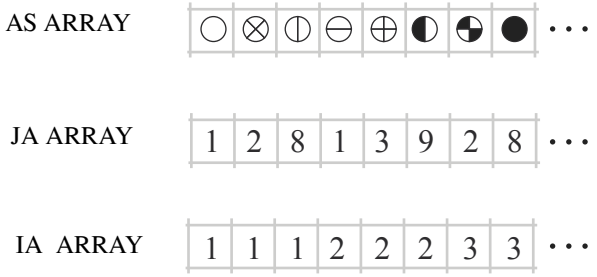


Figure 2: COO compression of matrix in Figure 1

per iteration, that is, per nonzero coefficient. Note that the code will produce the result y even if the coefficients and their indices appear in an arbitrary order inside the COO data structure.

2.2 Compressed Sparse Rows

The CSR format is perhaps the most popular sparse matrix representation. It explicitly stores column indices and nonzero values in two arrays `JA` and `AS` and uses a third array of row pointers `IRP`, to mark the boundaries of each row. The name is based on the fact that the row index information is compressed with

Algorithm 2.1 Matrix-Vector product in COO format

```
for i=1:nz
  ir = ia(i);
  jc = ja(i);
  y(ir) = y(ir) + as(i)*x(jc);
end
```

respect to the COO format, after having sorted the coefficients in row-major order. Figure 3 illustrates the CSR representation of the example matrix shown in Figure 1.

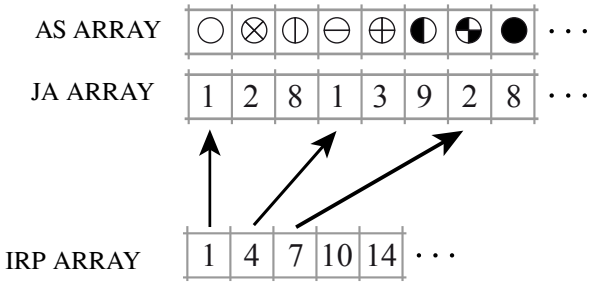


Figure 3: CSR compression of matrix in Figure 1

Algorithm 2.2 Matrix-Vector product in CSR format

```
for i=1:m
  t=0;
  for j=irp(i):irp(i+1)-1
     t = t + as(j)*x(ja(j));
  end
  y(i) = t;
end
```

Algorithm 2.3 Matrix-Vector product in CSC format

```
for j=1:n
  t=x(j);
  for i=jcp(j):jcp(j+1)-1
     y(ia(i)) = y(ia(i)) + as(i)*t;
  end
end
```

The code to compute the matrix-vector product $y = Ax$ is shown in Alg. 2.2; it requires three memory reads and two floating-point operations per iteration of the inner loop, i.e., per nonzero coefficient; the cost of the access to `x(ja(j))` is highly dependent on the matrix pattern and on its interaction with the memory hierarchy and cache structure. In addition, each iteration of the outer loop, i.e., each row of the matrix, requires reading the pointer values `irp(i)` and `irp(i+1)`, with one of them available from the previous iteration, and one memory write for the result.

2.3 Compressed Sparse Columns

The CSC format is extremely similar to CSR, except that the matrix values are first grouped by column, a row index is stored for each value, and column pointers are used; the resulting code is shown in Alg. 2.3. This format is used by the UMFPACK sparse factorization package [29], although it is less common for iterative solver applications.

2.4 Storage Formats for Vector Computers

The previous data formats can be thought of as "general-purpose", at least to some extent, in that they can be used on most computing platforms with little or no changes. Additional (and somewhat machine oriented) formats become necessary when moving onto special computing architectures if we want to fully exploit their capabilities.

Vector processors were very popular in the 1970s and 80s, and their tradition is to some extent carried on by the various flavors of vector extensions available in x86-like processors from Intel and other manufacturers. The main issue with vector computers is to find a good compromise between the introduction of a certain amount of "regularity" in the data structure to allow the use of vector instructions and the amount of overhead entailed by this preprocessing.

Many storage formats were developed for vector machines, including the diagonal (DIA), ELLPACK (or ELL), and Jagged Diagonals (JAD) formats.

The ELLPACK/ITPACK format (shown in Figure 4) in its original conception comprises two 2-dimensional arrays `AS` and `JA` with `M` rows and `MAXNZR` columns, where `MAXNZR` is the maximum number of nonzeros in any row [54]. Each row of the arrays `AS` and `JA` contains the coefficients and column indices; rows shorter than `MAXNZR` are padded with zero coefficients and appropriate column indices (e.g., the last valid one found in the same row).

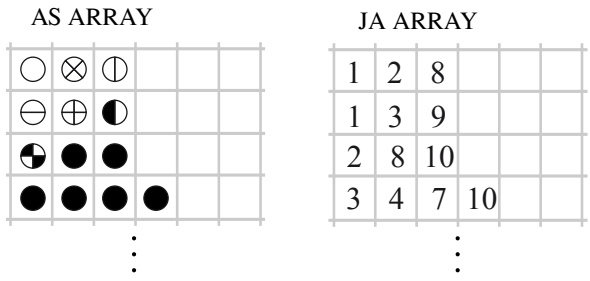


Figure 4: ELLPACK compression of matrix in Figure 1

The code to compute the matrix-vector product $y = Ax$ is shown in Alg. 2.4; it costs one memory write per outer iteration, plus three memory reads and two floating-point operations per inner iteration. Unless all rows have exactly the same number of nonzeros, some of the coefficients in the `AS` array will be zeros; therefore this data structure will have an overhead both in terms of memory

Algorithm 2.4 Matrix-Vector product in ELL format

```
for i=1:m
 t=0;
  for j=1:maxnzr
    t = t + as(i,j)*x(ja(i,j));
  end
  y(i) = t;
end
```

space and redundant operations (multiplications by zero). The overhead can be acceptable if:

1. The maximum number of nonzeros per row is not much larger than the average;
2. The regularity of the data structure allows for faster code, e.g., by allowing vectorization, thereby offsetting the additional storage requirements.

In the extreme case where the input matrix has one full row, the ELLPACK structure would require more memory than the normal 2D array storage.

A popular variant of ELLPACK is the JAgged Diagonals (JAD) format. The basic idea is to preprocess the sparse matrix by sorting rows based on the number of nonzeros; then, an ELLPACK-like storage is applied to blocks of rows, so that padding is limited to a given block. On vector computers the size of the block is typically determined by the vector register length of the machine employed.

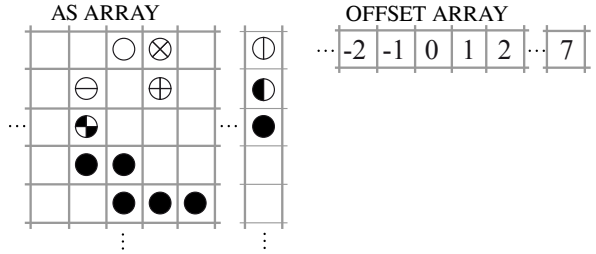


Figure 5: DIA compression of matrix in Figure 1

The DIAgonal (DIA) format (shown in Figure 5) in its original conception comprises a 2-dimensional array `AS` containing in each column the coefficients along a diagonal of the matrix, and an integer array `OFFSET` that determines where each diagonal starts. The diagonals in `AS` are padded with zeros as necessary.

The code to compute the matrix-vector product $y = Ax$ is shown in Alg. 2.5; it costs one memory read per outer iteration, plus three memory reads, one memory write and two floating-point operations per inner iteration. The accesses

Algorithm 2.5 Matrix-Vector product in DIA format

```
for j=1:ndiag
  if (offset(j) > 0)
    ir1 = 1; ir2 = m - offset(j);
  else
    ir1 = 1 - offset(j); ir2 = m;
  end
  for i=ir1:ir2
    y(i) = y(i) + as(i,j)*x(i+offset(j));
  end
end
```

to `AS` and `x` are in strict sequential order, therefore no indirect addressing is required.

3 GPGPUs

The NVIDIA GPGPU architectural model is based on a scalable array of multi-threaded, streaming multi-processors, each composed of a fixed number of scalar processors, one or more instruction fetch units, on-chip fast memory, which is partitioned into shared memory and L1 cache on older Fermi and Kepler generations but is physically separated on newer Maxwell and Pascal, plus additional special-function hardware.

CUDA is the programming model provided by NVIDIA for its GPGPUs; a CUDA program consists of a *host* program that runs on the CPU host, and a *kernel* program that executes on the GPU *device*.

The computation is carried on by threads grouped into blocks. More than one block can execute on the same multiprocessor, and each block executes concurrently. During the invocation (also called *grid*) of a kernel, the host program defines the execution configuration, that is:

- how many blocks of threads should be executed;
- the number of threads per block.

Each thread has an identifier within the block and an identifier of its block within the grid (see Figure 6). All threads share the same entry point in the kernel; the thread ID can then be used to specialize the thread action and coordinate with that of the other threads.

Figures 7 and 8 describe the underlying SIMT architecture. Note that in principle a single host may be connected with multiple devices. Each GPU device is made up by an array of multiprocessors and a global memory. The host is connected to devices using a bus, often having a much smaller bandwidth than that of the device global memory. Multiprocessors execute only vector instructions; a vector instruction specifies the execution on a set of threads

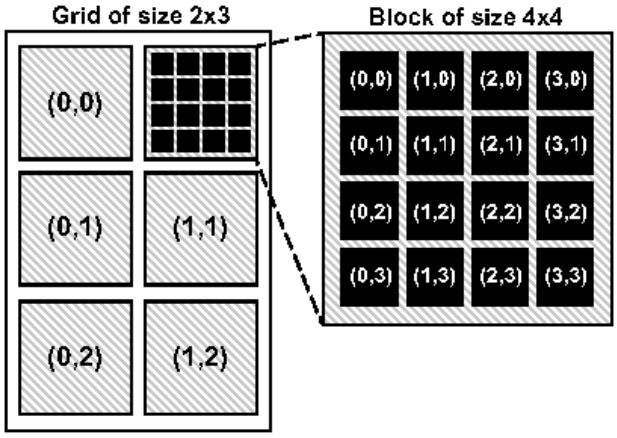


Figure 6: A 2D grid of threads

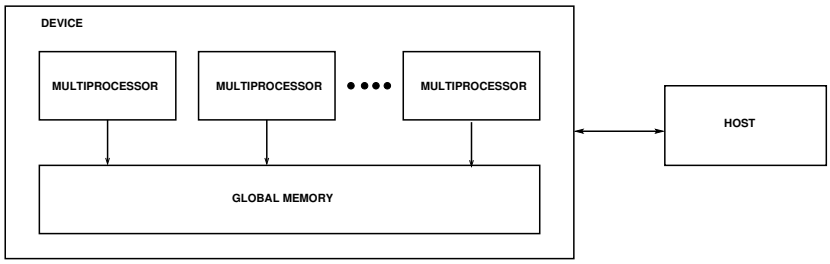


Figure 7: SIMT model: host and device

(called *warp*) with contiguous identifiers inside the block. The warp size is a characteristic constant of the architecture; its value is currently 32 for NVIDIA's GPUs. If threads within a warp execute different branches, the warp will issue a sequence of instructions covering all different control flows, and mask execution on the various threads according to their paths; this phenomenon is called *thread divergence*.

Each grid is executed on a single device; each thread block is enqueued and then scheduled on a multi-processor with enough available resources (in terms of registers, shared memory, and block slots) and retains all its resources until completion. A warp instruction is issued by a *scheduler* on an available vector unit that supports the relevant class of instructions. Threads in the same block share data using the *shared memory* and synchronize their execution waiting on a *barrier*.

To fully exploit the available bandwidth of both shared memory and global memory, some access rules should be followed by the threads of a warp (and, in some cases, of a half warp, i.e., a group of 16 consecutive threads for NVIDIA's

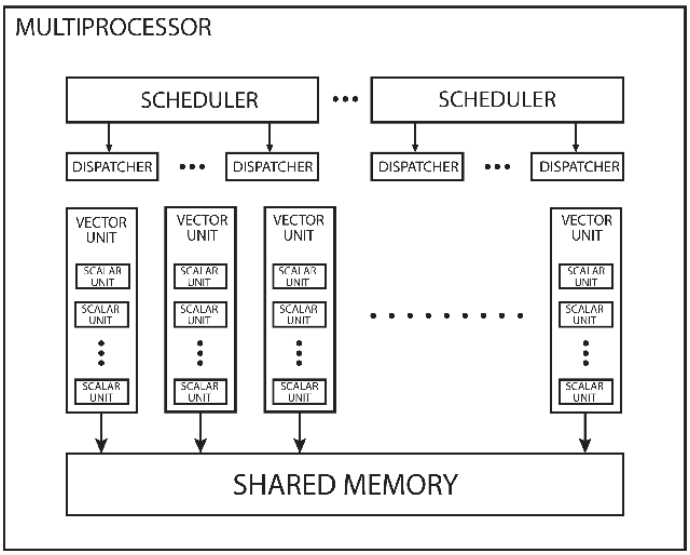


Figure 8: SIMT model: a multi-processor

GPUs). Shared memory, for example, is divided into banks, each one providing a constant throughput in terms of bytes per clock cycle. For each different architecture there is a specification on the correct access pattern that allows to avoid bank conflicts. Ideally, threads with increasing identifier in the same warp should read sequential elements of either 4 or 8 bytes in memory. Similarly, different access patterns should be followed according to the target GPGPU architecture to exploit the full global memory bandwidth. When these criteria are met, accesses are called *coalesced*. On NVIDIA hardware, a portable pattern which provides coalesced accesses is the following: each thread with index k within the warp ($0 \leq k < warpSize$) should access the element of size D (with D equal to 4, 8 or 16 bytes) at address $D \cdot (Offset \cdot warpSize + k)$.

The GPGPU uses different pipelines (modeled as groups of *units* on a multiprocessor) to manage different kind of instructions, and so it has different instruction throughputs. A basic arithmetic instruction, e.g., a floating-point addition, runs in parallel with a memory load request issued by another warp in the same clock cycle on the same multiprocessor. Therefore, to ensure best performance we need to optimize the bottleneck caused by the slowest set of instructions running on the same group of units. There are essentially three types of pipeline that can be replicated on the same GPU multiprocessor:

- a pipeline for groups of floating-point and integer units;
- a pipeline for groups of special function units (used for certain specialized arithmetic operations);
- a pipeline for groups of load/store units.

It is therefore pointless to speed up the arithmetic instructions of a given kernel if its performance is limited by the memory accesses; in such a case we need to concentrate on the efficiency of read and write requests to memory. This is the case for the sparse matrix-vector kernel, where the amount of arithmetic operations is usually comparable with the access requests in global memory (having a significantly lower throughput than the ALUs).

We have to accommodate for the coordinated action of multiple threads in an essentially SIMD fashion; however, we have to make sure that many independent threads are available. The number of threads that are active at any given time is called *occupancy*; the optimal value to exploit the GPGPU architecture is typically much larger than the vector lengths of vector computers, and is limited by the availability of registers.

Performance optimization strategies also reflect the different policies adopted by CPU and GPGPU architectures to hide memory access latency. The GPGPU does not make use of large cache memories, but it rather exploits the concurrency of thousands of threads, whose resources are fully allocated and whose instructions are ready to be dispatched on a multiprocessor.

The main optimization issue to support a GPGPU target then revolves around how an algorithm should be implemented to take advantage of the full throughput of the device. To make good use of the memory access features of the architecture, we need to maximize the regularity of memory accesses to ensure coalesced accesses. In the SpMV context, the ELLPACK format entails a regular access pattern to read the sparse matrix values and the y input vector, provided that we choose a memory layout for arrays ensuring their alignment to the appropriate boundaries in memory. The access pattern is easily exploited by assigning each row of the matrix to one thread; threads in a warp will work on a block of contiguous rows.

4 A Survey of Sparse Matrix Formats on GPGPUs

A significant number of research efforts have been devoted in the last years to improving the performance of the SpMV kernel on GPGPUs. They can be mainly categorized into three different development directions:

- applying novel sparse matrix formats, typically derived from classic ones;
- applying architecture-specific optimizations to existing formats;
- applying automated performance tuning of the matrix format and parameters.

These directions are clearly not mutually exclusive. Sparse matrices coming from different applications exhibit different sparsity patterns (i.e., distribution of nonzero entries). Therefore, devising a one-size-fits-all solution is a nearly impossible task; many GPGPU-specific sparse matrix representations have been

designed by applying some specific optimization technique best suited for a particular sparsity pattern. Different matrices may need have their own most appropriate storage formats to achieve best performance. Auto-tuning methods determining a combination of parameters that provides best performance can be effective for SpMV, since the optimal storage scheme is dependent upon the sparsity pattern. However, it is also important to consider general storage schemes that can efficiently store matrices with arbitrary sparsity patterns.

In this section we first classify and review a number of research efforts that proposed novel formats or format optimizations to efficiently represent sparse matrices on GPGPUs. Each format has its own strengths and weaknesses as explained in the following sections. We mainly classify the formats according to the base sparse matrix format (i.e., COO, CSR, CSC, ELL, DIA) they extend or derive from. We also consider those efforts that either propose a *hybrid* approach, where multiple formats can be used depending on the matrix sparsity pattern and other matrix parameters, or define a new storage format which does not directly extend existing ones. Then, in Section 4.8 we review some works devoted to study performance optimizations and propose automated tuning frameworks for SpMV on GPGPUs.

Table 2 summarizes the results of our taxonomy and provides a quick reference guide to the reader. In the table we also include the availability of the source code to the best of our knowledge. As also observed in [60], it would be beneficial for the HPC community to have open source code of matrix formats available, so to allow a detailed performance comparison. According to our survey, at the time of writing only 24 out of the 71 formats that we analyzed have been made available by their authors, either publicly or on demand. We also observe that few papers include links about implementation codes that are publicly available and some of these links are no longer valid.

Table 2: Taxonomy of GPGPU-specific sparse matrix formats

Base format	Issues	GPGPU variants	Available
COO	Memory footprint, atomic data access	ALIGNED_COO [91]	✗
		SCOO [27]	✓
		BRO-COO [95]	✗
		BCCOO and BCCOO+ [107]	✓
CSR	Coalesced access, thread mapping, data reuse	CSR optimization [13]	✓
		CSR optimization [11]	✗
		CSR optimization [46]	✗
		CSR optimization [36]	✗
		CSR optimization [87]	✓
		CSR optimization [112]	✗
		CSR optimization [100]	✗
		CSR optimization [79]	✓
		CSR optimization [3]	✗
		CSR optimization [44]	✗
		ICSR [108]	✗
		CSR-Adaptive [41, 25]	✓
		CSR5 [67]	✓
		ACSR [7]	✗
		MGPU CSR+ [12]	✓
		SIC [37]	✗

Continued on next page

Continued from previous page

Base format	Issues	GPGPU variants	Available
		RgCSR [83]	✓
		ArgCSR [48]	✓
		BIN-CSR and BIN-BCSR [101]	✗
		CMRS [55]	✗
		PCSR [31]	✗
		LightSpMV [68]	✓
		BCSR optimization [16]	✗
		BCSR optimization [22]	✗
		BCSR optimization [106]	✗
		BCSR optimization [98]	✓
CSC	Coalesced access	CSC optimization [49]	✗
ELLPACK	Zero padding	ELLPACK-R [96]	✓
		Sliced ELLPACK [78]	✓
		Warped ELL [72]	✗
		SELL-C-σ [57]	✗
		SELL-P [6]	✓
		ELLR-T [97]	✓
		Sliced ELLR-T [34]	✗
		HLL [10]	✓
		BELLPACK [22]	✗
		BSELLPACK [92]	✗
		AdELL [70]	✗
		CoAdELL [71]	✗
		AdELL+ [73]	✗
		JAD optimization [65]	✓
		ELLPACK-RP [17]	✗
		BiELL and BiJAD [114]	✗
		BRO-ELL [95]	✗
		Enhanced JDS [20]	✗
		pJDS [56]	✗
		ELL-WARP [105]	✗
		BTJAD [1]	✓
		BRC [8]	✗
		DDD-NAIVE, DDD-SPLIT [113]	✗
DIA	Zero padding	CDS [40]	✗
		HDI [10]	✓
Hybrid		HYB [13]	✓
		CSR + ELL [75]	✗
		CSR + ELL [76]	✗
		HEC [66]	✗
		TILE-COMPOSITE [111]	✓
		SHEC [38]	✗
		Cocktail [92]	✓
		HDC [110]	✗
		ELLPACK + DIA [72]	✗
		BCOO + BCSR [77]	✗
		BRO-HYB [95]	✗
Hybrid		BRO-HYBR, BRO-HYBR(S) [94]	✗
New		BLSI [80]	✗
		CRSD [93]	✗

One of the first SpMV kernel implementations for GPGPUs was proposed by Bolz et al. in [15]. The seminal work on accelerating the SpMV kernel on CUDA-enabled GPGPUs was presented by Bell and Garland in [13, 14], who provided a detailed study of sparse matrix formats and their access pattern on GPGPU and implemented CUDA kernels for the main classic storage formats, including COO, CSR, and ELL.

4.1 COO Variants

COO is the most intuitive storage format, is generic with respect to the sparsity pattern (e.g., its memory footprint depends only on the number of nonzeros and not on their distribution) but usually has worse performance than other formats on the SpMV kernel. Such poor behavior is due to the impossibility to achieve a simple decomposition of independent work into threads of execution; it is not possible to know in advance, without reading the content of the stored matrix itself, which set of coefficients will be used to calculate a single element of the resulting vector. Consequently, intermediate results should be integrated using *segmented reductions* or atomic data access [13, 14], because the absence of any ordering makes it impossible to foresee which threads will contribute to any given output vector entry, hence any two of them might produce conflicting updates. Preprocessing the COO format through parallel sorting by row and then employing parallel prefix scan is a suitable alternative to parallelization using atomic read-modify-write [12]. Although COO does not make use of padding, it still needs to explicitly store both the row and column indices for every nonzero element; therefore, it provides a bad memory footprint in many cases [92].

Shah and Patelin [91] proposed the ALIGNED_COO format, which extends COO to optimize performance of large sparse matrices presenting a skewed distribution of nonzero elements. The ALIGNED_COO format exploits some specific techniques, such as load balancing among threads by applying a data reordering transformation, synchronization-free load distribution, reuse of the input vector, reduction of memory fetch operations. However, its performance depends on the proper setting of some parameters and therefore a heuristic policy is required.

The Sliced COO (SCOO) format has been suggested by Dang and Schimdt [27]. As the name suggests, it exploits the idea of decomposing the matrix into a number of slices with respect to varying sparsity patterns. Their CUDA implementation exploits typical GPGPU computing features such as atomic operations and texture caching. However, the SCOO performance improvement for double-precision matrices is moderate, since double-precision atomic operations on common CUDA-enabled GPGPUs either do not exist or have lower performance than their single-precision counterpart.

Tang et al. [95] proposed to exploit lossless compression schemes tailored for the GPGPU in order to reduce the storage size of bits required to represent a sparse matrix and thus the memory bandwidth usage. Their idea is to compress the index data using an efficient bit representation format. The BRO-COO format applies the compression scheme only in the row index array `IA`, which is divided into intervals organized into a two-dimensional array and each interval is processed by a warp. The exploitation of a compression scheme requires to perform compression and decompression operations. The compression can be executed offline on the host CPU, while the decompression has to be performed online in the GPGPU before computing the matrix-vector product, thus introducing some overhead.

Yan et al. [107] presented the Blocked Compressed common COOrdinate (BCCOO) format, which extends the COO format with blocking to reduce the size for both `IA` and `JA` arrays. BCCOO uses bit flags to store the row indices so as to alleviate the bandwidth problem. The drawback of the BCCOO format, which is common to all block-based formats, is the zero fill-in in the data value array when a nonzero block contains zeros. In the same work, the authors also proposed an extension to the BCCOO format, called BCCOO+, to improve the access locality to the multiplied vector. In this format, the sparse matrix is first partitioned into vertical slices and then the slices aligned in a top-down manner. Then, the BCCOO format is applied to the vertically sliced and rearranged matrix. Since BCCOO+ requires a temporary buffer to store the intermediate results that are then combined to generate the final results, it is not always preferable to BCCOO; therefore, an auto-tuning strategy is suggested to select which format should be used. In addition, Yan et al. [107] also proposed an efficient matrix-based segmented sum/scan to maximize the benefit from their BCCOO and BCCOO+ formats on GPGPUs.

4.2 CSR Variants

Like COO, CSR is a flexible and general purpose format. However, this flexibility has a cost when distributing the work across multiple threads; problems range from lack of coalescing, causing an inefficient memory access pattern, to load imbalance and thread divergence (i.e., many threads are idle while others are busy), which can lead to a serious loss of parallel efficiency.

Various research efforts on SpMV for GPGPUs have focused on optimizing the base CSR format. Therefore, we first analyze them and then we review those proposals devoted to suggesting new storage formats that stem from CSR.

4.2.1 CSR Optimizations

Bell and Garland in [13] implemented two kernels for the CSR format, namely scalar and vector, that differ in the number of threads assigned to a single row. The scalar kernel assigns one thread per row, while the vector kernel assigns multiple threads (precisely one warp, that is 32 threads) per row and can thus achieve more efficient memory access patterns by exploiting coalescing, but only when the matrix has an average number of nonzero elements per row greater than 32. Therefore, subsequent works [11, 46, 36, 87, 112] on the CSR storage format have focused on determining the optimal number of threads per row on the basis of the number of nonzero elements per row. Guo and Wang [46] proposed a method that selects the number of threads to either half-warp (16 threads) or one warp (32 threads) on the basis of the characteristics of the input matrix. El Zein and Rendell [36] suggested to switch between the scalar and vector methods according to the number of nonzero elements per row. The works in [87, 112] improved the performance of the CSR vector kernel by selecting the optimal number of threads equal to a power of 2 and no more than the warp size on the basis of either the average [87] or the maximum [112] number of nonzero

elements per row. The average is preferable to the maximum because the first does not require to pre-scan the input matrix for its computation.

The work by Baskaran and Bordawekar [11] presents some optimization techniques for SpMV kernels focusing on the CSR matrix format. Their optimizations, that have become a reference point for subsequent works, include the exploitation of synchronization-free parallelism, global memory access coalescing, aligned global memory access, and data reuse of input and output vectors. As regards the thread mapping strategy, Baskaran and Bordawekar used 16 threads per row in order to alleviate thread divergence.

Ahamed and Magoules [3] focused on finite element matrices and presented how to dynamically tune the gridification (i.e., the dimension of the block and number of threads per block) when the sparse matrix is stored in CSR format.

Wang et al. [100] suggested some techniques to optimize the CSR storage format and threads mapping, to avoid thread divergence. Specifically, they slightly modified the CSR format to reduce the transmission time between host and device, and set the number of threads per row to slightly more than the average number of nonzero elements per row.

Yang et al. [108] presented an improved CSR (ICSR) storage format to address the problem of global memory alignment of the CSR vector kernel by adding some zero elements to the end of every row in the CSR storage format.

Mukunoki et al. [79] also proposed optimization techniques for the CSR format but on the more recent NVIDIA Kepler architecture, taking advantage of three new features: 48KB read-only data cache, shuffle instructions, and expanding the number of thread blocks in the x-direction that can be defined in a grid.

Guo and Gropp [44] presented a simple auto-tuning method to improve SpMV performance when the sparse matrix is stored in CSR format. It consists in sorting the rows in increasing order of the number of nonzero elements per row and partitioning them into several ranges, and then assigning a given number of threads for different ranges of the matrix rows in order to balance the threads workload.

The parallelized CSR-based implementation named as LightSpMV [68] takes advantage from the fine-grained dynamic distribution of the matrix rows. The authors investigated two parallelization approaches at the vector and warp levels using atomic operations and warp shuffle functions.

The CSR optimization taken by Baxter [12] with the CSR+ format in his ModernGPU library is quite different from all others. An SpMV is considered as a sequence of two steps. The first step, which is highly parallelizable, consists of an element-by-element multiplication of matrix entries by vector entries. The second step employs a segmented reduction to produce the final output vector. The resulting algorithm has high efficiency and is very fast, but also very complex to implement and modify.

Since CSR is the most memory efficient and its usage can avoid or have minimal conversion overheads (that we analyze in Section 5.4), some more recent efforts have also focused on keeping the CSR format intact and possibly limiting its preprocessing [7, 41].

Greathouse and Daga [41] proposed the CSR-Stream and the related CSR-Adaptive algorithms. CSR-Stream is a CSR optimization that statically fixes the number of nonzeros that will be processed by one warp and streams these values into the GPGPU shared memory, thus improving access coalescing and parallelism. Since the CSR-Stream efficiency is lost when a warp operates on rows with a large number of nonzeros, the companion CSR-Adaptive algorithm (whose code is available in the ViennaCL library) allows to adaptively switch between using CSR-Stream for rows with relatively few nonzeros and the traditional vector kernel [13] for rows with a large number of nonzeros. An improved version of CSR-Adaptive has been presented by the same authors in [25] to address the CSR-Adaptive's poor performance on irregular matrices.

Liu and Vinter [67] proposed the CSR5 storage format, that is designed to be cross-platform and insensitive to the sparsity pattern of the matrix, thus being appropriate for both regular and irregular matrices. These results are achieved through the addition of extra data structures to the standard CSR and the execution of in-place transpose of parts of the matrix. However, the limitations of CSR5 lie in its complexity and the matrix transpose operations, which may cause significant overheads.

ACSR, another adaptive SpMV algorithm that relies on the standard CSR format, has been proposed by Ashari et al. [7] for large power-law matrices. It aims at reducing thread divergence by grouping rows into bins on the basis of the number of nonzeros; it also leverages dynamic parallelism, a functionality on recent NVIDIA GPUs, in order to improve memory coalescing.

4.2.2 New Formats Based on CSR

The Compressed Sparse Row with Segmented Interleave Combination (SIC) storage format has been proposed by Feng et al. [37]. Stemming from CSR, the SIC format employs an interleave combination pattern, that combines certain amount of CSR rows to form a new SIC row, plus reordering and segmented processing to further improve the performance. The SIC goals are to alleviate thread divergence and to lessen the load imbalance among warps. However, its drawbacks are the need to first transform in CSR and to carefully tune some parameters, namely the number of contiguous CSR rows that will form the new SIC row and the thread block size.

The Row-grouped CSR (RgCSR) format proposed by Oberhuber et al. [83] aims at fulfilling the coalesced access to the array values and columns by distributing threads to matrix rows according to their computational load. The RgCSR format first divides the matrix into groups of rows with similar workload; then, a different number of threads within the group is assigned to each row depending on the number of nonzero elements in it. RgCSR requires block-level synchronization which is less efficient than warp-level synchronization; its row partition policy depends on a user-given parameter and may fail in the case of matrices having a very skewed distribution of the nonzero elements. ArgCSR, an improved version of RgCSR, was later presented [48].

The BIN-CSR storage format has been proposed by Weber et al. [101] for

arbitrary sparsity patterns and presents some similarity with RgCSR. BIN-CSR combines the CSR format with a partitioning scheme that groups the matrix entries into bins, being a bin a portion of the matrix that is accessed concurrently by a group of threads. To optimize global memory access by coalescing, all rows in each bin should have the same length, therefore padding is used to complete shorter rows within a bin. The diagonal is stored separately to allow an efficient Jacobi preconditioner. However, the storage format considers only single precision.

A similar idea of processing a sparse matrix in chunks larger than individual rows has been also exploited in the Compressed Multi-Row Storage (CMRS) format presented in [55]. CMRS extends the CSR format by dividing the matrix rows into strips; a warp is then assigned to a strip for processing. Specifically, the row pointers `IRP` array used in CSR is generalized to mark the boundaries of each strip, rather than each row. The number of rows in a strip is equal for all the strips and is set to be a power of 2 from 1 to 16, the actual value being limited by the buffer size in NVIDIA's Fermi GPUs. The advantage of CMRS is that the multiplied vector is reused. The authors evaluated the performance of the proposed format on a large set of matrices from the UFL collection and measured the CMRS efficiency as a function of the mean and standard deviation of the number of matrix nonzero elements per row.

Dehnavi et al. [31] proposed the Prefetch-Compressed Row Storage (PCSR) format, that combines CSR with a novel vector partitioning scheme, zero padding of the matrix rows and computation strategy. Their goal was to accelerate finite-element SpMV kernels on NVIDIA GT8800.

Blocking storage techniques can be used to improve CSR compression and data reuse, especially of the input vector elements. BCSR is the classical blocked version of CSR, storing and indexing two-dimensional small dense blocks with at least one nonzero element and uses zero padding to construct full blocks [52]. Thus, BCSR reduces the indexing overhead for storing a sparse matrix but it needs zero fill-in in the blocks. BCSR has been first investigated on GPGPUs by Buatois and al. [16], who did not optimize for memory coalescing, and then tuned for GPGPU by Choi et al. [22]. Xu et al. [106] proposed a cache blocking method for the NVIDIA's Fermi architecture: first the matrix is transformed from CSR to BCSR format; then, one block row of the matrix is assigned to one block of threads so that the corresponding part of the input vector can be reused in the cache on Fermi GPU. Verschoor and Jalba [98] studied a number of mappings for the BCSR format, that is how to map multiple block rows to thread blocks; their results also show that block reordering can lead to a good performance improvement thanks to better memory coalescing, especially when small blocks are used.

A extension of the BIN-CSR format that exploits blocking and is therefore called BIN-BCSR has been proposed in [101].

4.3 CSC Variants

The CSC format, which compresses each column of the matrix rather than the rows as in CSR, has the advantage of allowing a regular access to the input vector at the expense of irregularity of access to the output vector. Its GPGPU implementation has been presented in [49] in two versions, with and without vectorization. However, in their performance analysis, CSC turns out to be the worst format because the computation of each output element requires irregular accesses to a temporary array where intermediate multiplication results are stored to be later summed together. These poor results also explain the lack of other works on this format, which (as already noted) is more commonly used in the context of sparse matrix factorizations.

4.4 ELLPACK Variants

The ELLPACK format and its variants are perhaps the most effective formats for GPGPUs and many research works have focused on them. ELLPACK is efficient for matrices with regular or partially regular sparsity patterns, in which the maximum number of nonzero elements per row does not differ significantly from the average. However, when the number of nonzero elements per row varies considerably, the ELLPACK performance degrades due to the overhead of storing a large number of padding zeros (we recall that rows are zero-padded in order to reach the length of the longest nonzero entry row, as described in Section 2.4). Therefore, several proposals have been made to modify this format so as to reduce the storage overhead and make it effective for general sparse matrices.

The ELLPACK-R format proposed by Vázquez et al. [96] is the first variant we examine that aims to reduce the memory bandwidth usage associated with the padding zeros. To this end, it avoids accessing the zeros by storing the number of nonzeros in each matrix row in an additional array. The usage of the ELLPACK basic storage allows the GPGPU to perform coalesced access to global memory, while the availability of the length of each row reduces the amount of overhead arithmetic operations. ELLPACK-R is now considered a reference storage format on GPGPUs and has been used in various papers for performance comparison, e.g., [56, 95].

To address the zero padding issue and reduce the memory overhead of the ELLPACK format, Monakov et al. [78] proposed a sliced version of ELLPACK (called Sliced ELLPACK and usually abbreviated as SELL or SELL-C), which can be considered as a generalization of ELLPACK-R. Indeed, during a preprocessing the rows of the sparse matrix are reordered and partitioned into several slices of similar lengths (being a slice a set of adjacent rows) and each slice is then packed separately in the ELLPACK format. An additional data structure is required to keep track of the indices of the first element in each slice. The finer granularity allows to reduce the amount of zero padding, because in Sliced ELLPACK the number of extra zeros depends on the distances between the shortest and the longest rows in slices, rather than in the whole matrix as in ELLPACK.

The slice size (i.e., the number of rows per slice, also denoted with C) is either fixed for all the slices or variable for each slice; in the latter case, a heuristics is used to find the proper slice size. Using a slice size equal to 1 results in the CSR format, while using a slice size equal to the matrix size results in the basic ELLPACK format. Each slice is assigned to a block of threads in CUDA and thread load balancing can be achieved by assigning multiple threads to a row if required. The drawback of the Sliced ELLPACK format is related to its susceptibility in picking up the right slice size, because a wrong slice configuration can adversely affect the performance.

An improvement of Sliced ELLPACK has been proposed by Maggioni et al. [72] and is based on warp granularity and local rearrangement to reduce the overhead associated with the data structure. In the Warped ELL format, the slice size is chosen to match the warp size and a local row reordering within each slice is applied to reduce the variability of the number of nonzeros per row and improve the data structure efficiency without affecting the cache locality. The proposed format achieves a reasonable performance improvement over Sliced ELLPACK for the considered set of matrices.

The use of some kind of preprocessing where rows or columns are permuted is another interesting idea that has been exploited in a number of works [97, 57, 6, 34, 114] to improve the ELLPACK performance. Vázquez et al. [97] presented the ELLR-T format, which extends their previously proposed ELLPACK-R format [96] by applying a preprocessing phase. During this preprocessing, nonzero elements and their column indices are permuted and zero padding occurs and each row is a multiple of 16 (half-size warp). Such modification aims at guaranteeing coalesced and aligned access to the global memory. In contrast to ELLPACK-R, T multiple threads (with T varying from 1 to 32) may operate on a single row to collaboratively compute one element of the output vector. However, the amount of zero padding with respect to ELLPACK-R is not reduced significantly. The authors also presented an auto-tuning strategy to find out the best combination of ELLR-T configuration parameters (the number of threads T and the size of thread block) on the basis of the sparse matrix/GPU architecture combination.

The Sliced ELLR-T is s a modification of Sliced ELLPACK and ELLR-T and has been suggested by Dziekonski et al. [34] with the aim of reducing the memory overhead of ELLPACK and obtain a good throughput. As in Sliced ELLPACK, the matrix is divided into slices and, as in ELLR-T, multiple threads operate on single row. Thanks to the partition into slices, Sliced ELLR-T reduces the amount of memory required for storage as in Sliced ELLPACK.

SELL-C-σ [57] is a variant of Sliced ELLPACK that applies row sorting so that rows with a similar number of nonzero elements are gathered in one slice, thus minimizing the storage overhead. Such format, which has been proposed as a unified sparse matrix data layout for modern processors with wide SIMD units, allows optimization for a certain hardware architecture via parameters C (the slice size) and σ (the sorting scope). Currently, such parameters need to be manually tuned. A modification of SELL-C-σ tailored for an efficient implementation on NVIDIA GPUs and resulting in the Padded Sliced ELLPACK

(SELL-P) format (whose code is available in the MAGMA library) has been proposed by Anzt et al. [6]. By exploiting the assignment of multiple threads per row as in Sliced ELLR-T, they modified SELL-C-σ by padding rows with zeros such that the row length of each slice becomes a multiple of the number of threads assigned to each row. However, efficient row sorting required by these formats remains an issue to be addressed.

Tang et al. [95] proposed to apply compression schemes to the ELLPACK format, similarly to what already done with COO. Their BRO-ELL scheme compresses the `JA` array using bit packing.

Zheng et al. [114] proposed the Bisection ELLPACK (BiELL) storage format with the aims to reduce the zero padding and improve load balance. It is an ELLPACK based storage format; the bisection in the name refers to the fact that when half of the unknown results are generated, all the threads are reassigned to the other half of the rows. The BiELL format is built in a number of preprocessing steps: (1) the matrix rows are partitioned into strips of warp size; (2) the rows within each strip are sorted in non-increasing order of the number of nonzero elements per row; (3) the resulting matrix is compressed by shifting all the nonzero elements to the left; (4) according to the bisection technique, the columns in the strips are divided into groups; (5) the elements in each group are stored in ELL format. Similarly to Sliced ELLPACK, BiELL partitions the matrix into strips but then it divides the strips into groups based on the bisection technique to improve load balance.

Choi et al. [22] proposed the blocked ELLPACK (BELLPACK) storage format which exploits block structures that appear in some sparse matrices. BELLPACK first rearranges the rows of the sparse matrix in decreasing order of the number of nonzero elements; then, the rows are separated into blocks and each block is stored in the ELLPACK format. BELLPACK is thus an extension of ELLPACK with explicit storage of dense blocks to compress data structure and row permutations to avoid unevenly distributed workloads. The authors also proposed a model to predict matrix-dependent tuning parameters, such as the block size. BELLPACK obtains performance improvements only on matrices that have small dense block sub-structures, as those arising in applications based on the finite-element method. Indeed, its drawback is the overhead introduced by zero fill-in in the blocks and zero padding to let the number of blocks per row be the same.

A blocked version of Sliced ELLPACK, called BSELLPACK, is discussed in [92]. It presents a slightly lower overhead than BELLPACK, because it requires fewer padding zeros.

The major drawback of formats such as Sliced ELLPACK and its variants, BELLPACK and BiELL, is related to the preprocessing of the sparse matrix, which uses computational resources and may involve a large memory overhead, especially when the matrix has not a natural block structure.

Adaptive ELL (AdELL) [70] is an ELLPACK-based format proposed for matrices with an irregular sparsity pattern. It pursues a self-adaptation goal by exploiting the idea of adaptive warp-balancing, where the computation on nonzero elements is distributed among balanced hardware-level blocks (warps).

Specifically, each warp can process from one to multiple rows, thus exploiting the vectorized execution, and each row can have a different number of working threads by following a best-fit heuristic policy where heavyweight rows are assigned to more threads. AdELL has been inspired by the idea first presented in the Sliced ELLPACK format to partition the matrix in slices and to represent each slice with a local ELLPACK structure. It presents a number of advantages: it ensures load balance, does not require the tuning of any parameter and the use of block-level synchronization primitives, and preserves cache locality for the dense x vector.

The AdELL format has been extended by the same authors into the CoAdELL format [71] by exploiting a lossless compression technique that targets column indices and is based on delta encoding between consecutive nonzeros. More recently, an advanced version of the AdELL format, called AdELL+, has been proposed in [73]. In particular, it integrates complementary memory optimization techniques into AdELL in a transparent way, reduces the preprocessing cost through an improved adaptive warp-balancing heuristic with respect to that of AdELL, and presents a novel online auto-tuning approach.

Since the JAD (or JDS) format can be considered as a generalization of ELLPACK without the assumption on the fixed-length rows, we consider in this context some works that proposed the JAD format and its variants for GPGPUs. JAD is a more general format than ELLPACK but it can still guarantee coalesced memory accesses. However, the JAD kernel requires to preprocess the matrix, because it first sorts the rows in non-increasing order of the number of nonzero elements per row, then shifts all nonzero elements to the left.

The base JAD format for GPGPU has been suggested in [65, 20]. In both works the matrix rows are first sorted in non-increasing order of the number of nonzeros per row and then compressed. The JAD kernel presented by Li and Saad [65] (whose code is available in the CUDA-ITSOL library) assigns one thread per row to exploit fine-grained parallelism. However, thanks to the the JAD format, the kernel does not suffer from the performance drawback of the scalar CSR kernel in [13, 14] due to the noncontiguous memory access. JAD can reduce the computational efforts and obtains a performance improvement with respect to CSR and ELL, but its kernel may suffer from unused hardware reservation [114].

Cao et al. [17] presented the ELLPACK-RP format, which combines ELLPACK-R with JAD. The matrix is stored in the ELLPACK-RP format by first constructing the ELLPACK-R format and then permuting the rows by shifting the longer rows upwards and the shorter ones downwards in decreasing order as in JAD. However, ELLPACK-RP has been evaluated only on a limited set of matrices without taking into account the preprocessing time and showed improvement with respect to ELLPACK-R only when the matrix has a very irregular sparsity pattern.

Kreutzer et al. [56] proposed the Padded Jagged Diagonals Storage (pJDS) format, which is also based on ELLPACK-R. pJDS reduces the memory footprint of ELLPACK-R by first sorting the rows of the ELLPACK scheme according to the number of nonzeros, then padding blocks of consecutive rows to the

longest row within the block. Such format maintains load coalescing while most of the zero entries can be eliminated. Furthermore, with respect to BELLPACK and ELLR-T, the pJDS format is suited for general unstructured matrices and does not use any a priori knowledge about the matrix structure. There are also no matrix dependent tuning parameters.

A variant of pJDS, called ELL-WARP, was recently presented in [105] for finite element unstructured grids. It first sorts the rows by length, then arranges rows into groups of warp size and pads accordingly; finally, it reorders the data within each warp in a column-major coalesced pattern. Similarly to other formats that employ reordering of values, the reordering overhead should be taken into account as it is expensive with respect to the actual SpMV computation time.

Ashari et al. [8] have recently proposed the Blocked Row-Column (BRC) storage format (which has been integrated with PETSc but is not yet publicly available). BRC is a a two-dimensional row-column blocked format, where row-blocking of the matrix aims at reducing thread divergence and column-blocking at improving load balance. We classify BRC has an ELLPACK variant because it combines the row permutation (based on the number of nonzeros in each row) of JDS and the padding mechanism of ELLPACK. BRC requires to set two parameters (the number of rows in a block that maps to a warp and the block size) that depend on the warp size and the matrix structure. Interestingly, the authors also evaluated the preprocessing time needed to set up the matrix in the BCR format, which turned out to be two orders of magnitude larger than the SpMV time.

Besides the BiELL format, Zheng et al. [114] suggested also the Bisection JAD (BiJAD) format, which exploits the same bisection technique of BiELL to improve load balance. While BiELL sorts the rows only within each strip, BiJAD sorts all the rows of the matrix in order to reduce the padding zeros.

The Transpose Jagged Diagonal (TJAD) storage format has been tailored for GPGPUs in [1]. TJAD is inspired by JAD but reduces the storage space used by JAD; specifically, TJAD sorts the columns rather than the rows according to the number of nonzeros. The authors used registers to enhance data reuse and proposed a blocked version of TJAD to allow computing from the fast shared memory. TJAD achieves a significant performance improvement for matrices that are neither highly uniform nor highly non-uniform in the number of nonzeros in the rows.

In the context of the PSBLAS software [39, 19], we have implemented two variations on the ELLPACK format. The first is quite similar to the base format, except that it additionally stores a vector with the actual number of nonzero elements per row. Each row is assigned to a set of threads which computes all the associated operations and stores the result in the output vector. The set comprises one or two threads depending on the average number of nonzeros per row; this is somewhat similar to what is done in ELLPACK-R [96]; however, in our implementation we do *not* need to change the data structure to reflect the usage of more threads.

The second ELLPACK-derived format we proposed in [10] (all our code is

available in the PSBLAS library) is Hacked ELLPACK (HLL), which alleviates one major issue of the ELLPACK format, that is, the amount of memory required by padding for sparse matrices in those cases where the maximum row length is substantially larger than the average. To limit the padding overhead we break the original matrix into groups of rows (called *hacks*), and then store these groups as independent matrices in ELLPACK format; a very long row only affects the memory size of the hack in which it appears. The groups can be optionally arranged selecting rows in an arbitrary order; if the rows are sorted by decreasing number of nonzeros we obtain essentially the JAgged Diagonals format, whereas if each row makes up its own group we are back to CSR storage. Threads within a hack will access consecutive rows of the data structure; hence the size of the hack should be a multiple of the warp size to guarantee coalesced memory accesses. The padding overhead is proportional to the number of rows within the hack; therefore, the hack size should be kept as small as possible, normally just one warp. Similarly to our ELLPACK implementation, we assign one or two threads per row depending on the average row length.

Sorting the rows of the matrix based on rows' length, like in the JAD format, tends to reduce the amount of padding, since large rows will go together in the same submatrix. However, it also entails the use of permutations in computing the output vector, and therefore it does not necessarily improve overall performance. The HLL format is similar to the Sliced ELLR-T format [34], where however no provisions are made for an auxiliary ordering vector.

4.5 DIA Variants

DIA is not a general-purpose format and is only convenient for matrices with a natural diagonal structure, often arising from the application of finite difference stencils to regular grids. This format is efficient for memory bandwidth because there is no indirection as in the COO, CSR, and JAD formats, as confirmed by the results reported in [65] for diagonally structured matrices. However, DIA can potentially waste storage and computational resources because it requires zero padding for non-full diagonals in order to maintain the diagonal structure. Moreover, while the DIA code is easily vectorized, it does not necessarily make optimal use of the memory hierarchy. While processing each diagonal we are updating entries in the output vector `y`, which is then accessed multiple times; if `y` is too large to remain in the cache memory, the associated cache miss penalty is paid multiple times and this may reduce overall performance.

Yuan et al. [113] presented two formats based on DIA, called DDD-NAIVE (Naive Direct Dense Diagonal) and DDD-SPLIT (Split Direct Dense Diagonal).

Godwin et al. [40] proposed the Column Diagonal Storage (CDS) format for block-diagonal sparse matrices that takes advantage of the diagonal structure of matrices for stencil operations on structured grids. CDS minimizes wasted space and maximizes memory coalescing across threads by storing the matrix according to its block-diagonal structure.

We have designed the Hacked DIA (HDI) format [10] to limit the amount of padding, by breaking the original matrix into equally sized groups of rows

(*hacks*), and then storing these groups as independent matrices in DIA format. This approach is similar to that of HLL, and requires using an offset vector for each submatrix. Again, similarly to HLL, the various submatrices are stacked inside a linear array to improve memory management. The fact that the matrix is accessed in slices also helps in reducing cache misses, especially regarding accesses to the vector `y`.

4.6 Hybrid Variants

HYB is the best known hybrid format suitable for GPGPU and is most suitable for matrices that do not have a regular structure. It represents the reference format for most works that proposed novel sparse matrix formats and related optimizations for GPGPUs, e.g., [11, 38, 76, 80, 83], including the present paper. It was proposed by Bell and Garland in [13, 14], included in the cuSPARSE library [82] and is a hybrid combination of ELLPACK and COO formats, where the majority of matrix entries are stored in ELL. Specifically, HYB allocates the first K nonzeros per row (zero padding rows that have less than K nonzeros) in the ELLPACK portion, and stores the remaining nonzero elements in the COO portion. The value for K is determined by the rule that at least one third of the matrix rows contains K or more nonzero elements. The motivation for HYB proposal is that ELLPACK and COO formats are suitable in slightly complementary situations: therefore, the idea is to jointly exploit the ELLPACK high performance potential and the performance invariability of COO. The kernel time of HYB improves significantly with good performance over a large collection of sparse matrices, however at the cost of high data organization, more complex program logic, and memory transfer time.

BRO-HYB, an extension of the HYB format with compression, was proposed by Tang et al. [95] in their already cited work. Similarly to HYB, it combines the BRO-ELL and BRO-COO formats by dividing the matrix into BRO-ELL and BRO-COO partitions with the same algorithm proposed by Bell and Garland. The same authors proposed in [94] two new hybrid variants, called BRO-HYBR and BRO-HYBR(S), with the goal to further reduce the memory-bandwidth usage. Both the variants partition the matrix into BRO-ELL and BRO-CSR components; BRO-HYBR(S) also sorts the rows according to their length.

The works in [75, 76, 111, 66] considered combinations of CSR and ELLPACK formats for storing the matrix.

Maringanti et al. [75] proposed a simple combination of the two formats, motivated by the observation that the CSR format in the vector kernel version (where one warp is assigned to each matrix row) performs well when the number of nonzero elements per row is sufficiently larger than the warp size [13]. Specifically, rows having a number of nonzero elements greater than warp size are encoded in the CSR format, while the remaining ones in the ELLPACK format.

The HEC format by Liu et al. [66] combines CSR and ELLPACK by storing a part of the matrix in ELLPACK and the remaining one in CSR. The partition is determined by a minimum problem, whose formulation requires to know the

relative performance of the ELL and CSR matrices thus needing preliminary tests.

Matam and Khotapalli [76] proposed a data structure to store some rows in CSR and the remaining in ELLPACK. They proposed a methodology that analyzes the sparse matrix structure and chooses the right data structure to represent it. Their methodology uses some threshold parameters (nonzero elements per row, maximum number of elements per thread and per warp) and attempts to balance the load amongst threads and also to identify the right kind of format between CSR and ELLPACK, or a combination of both, to use for storing the sparse matrix. Their hybrid combination of CSR and ELLPACK obtains a performance improvement for the SpMV kernel of 25% on average with respect to the HYB format. However, their approach presents some data structure and operational space overhead.

Yang et al. [111] focused on sparse matrices used in graph mining applications, which have the peculiarity to represent large, power-law graphs. For such power-law matrices, they proposed a composited storage scheme, called TILE-COMPOSITE that combines the CSR and ELL formats. During a preprocessing phase, the matrix is first divided into fixed width tiles by column index, then in each tile the rows are ranked according to the number of nonzero elements and partitioned into balanced workloads. If the length of the longest row in the workload is greater than or equal to the number of rows in the workload, then the CSR format is used, otherwise ELL is used. An automated parameter tuning is also proposed to calculate the number of tiles and the partition strategies for each tile. The drawback of this combined format is the sorting cost for re-structuring the matrix.

Feng et al. [38] proposed the Segmented Hybrid ELL+CSR (SHEC), which exploits the reduction of auxiliary data used to process the sparse matrix elements that allows to improve the SpMV throughput and reduce the memory footprint on the GPGPU. In SHEC the matrix is divided into several segments according to the length of the matrix rows, and then a hybrid combination of ELL and CSR processing pattern is chosen for each segment. In addition, the GPGPU occupancy is taken into consideration to avoid unnecessary idleness of computational resources.

The Cocktail Format by Su and Keutzer [92] takes advantages of different matrix formats by combining them. It partitions the input sparse matrix into several submatrices, each specialized for a given matrix format.

Some works have focused on the combination of the DIA sparse format with other formats [110, 72] for matrices that exhibit a quasi-diagonal sparsity pattern.

Maggioni et al. [72] proposed the combination of DIA with the ELLPACK format to store the dense diagonals of the sparse matrix using the DIA format (adding alignment padding if necessary) and the remaining elements of the matrix using the ELLPACK format. This combination of ELLPACK and DIA is useful for the Jacobi iteration because of the faster access to the elements on the separate diagonal; however, it achieves a limited performance improvement with respect to ELLPACK and only for those matrices having a relatively dense

diagonal band.

Yang et al. [110] proposed the HDC format, which is a hybrid of DIA and CSR, to address the inefficiency of DIA in storing irregular diagonal matrices and the imbalance of CSR in storing nonzero elements. The decision on whether or not to store a diagonal by using either DIA or CSR is based on the number of nonzero elements in that diagonal: if there are more nonzeros than a threshold, the diagonal is stored in DIA, otherwise in CSR. However, the question of how to properly set the threshold is not discussed in the paper.

For hybrid blocked storage formats, Monakov and Avetisyan [77] presented a format based on a combination of BCOO and BCSR features (being BCOO the classical blocked version of COO). Furthermore, nonzero elements not covered by blocks are stored in the ELL format. To select the best block size, they proposed both dynamic programming and a greedy heuristic. However, their approach has large memory requirements because of the need to store additional zero elements.

4.7 New GPGPU-specific Storage Formats

Neelima et al. [80] proposed a new format called Bit-Level Single Indexing (BLSI) that aims to reduce both the data organization time and the memory transfer time from CPU to GPGPU. BLSI uses only one array of size equal to the number of nonzero elements to represent the indices in order to store them by embedding the column information in the bits of row indices information. BLSI reduces the combined time of data preprocessing and movement and it is thus a good choice for a single or a limited number of SPMV computations. However, it increases significantly the running time of the SPMV kernel with respect to the HYB format; it is therefore not attractive for iterative solvers where the same data structure is reused over multiple kernel invocations, amortizing the conversion cost.

Sun et al. [93] proposed a new storage format for diagonal sparse matrices, called Compressed Row Segment with Diagonal-pattern (CRSD), which alleviates the zero fill-in issue of the DIA format. The idea is to represent the diagonal distribution by defining the diagonal pattern, which divides diagonals into different groups, and to split the matrix into row segments. In each row segment, nonzero elements on the diagonals of the same group are viewed as the unit of storage and operation and are therefore stored contiguously.

4.8 Automated Tuning and Performance Optimization

In the previous sections, we have seen in the format-by-format analysis that a number of SpMV optimizations previously used for other architectures [104, 103] have been also applied to GPGPUs. They can be roughly classified into the three categories of reorganizing for efficient parallelization, reducing memory traffic, and orchestrating data movement, and include, among the others, fine-grain parallelism, segmented scan, index compression, and register blocking [103].

However, a result in common with other architectures is that there is no single winner implementation, but rather the best choice of sparse storage format depends on the sparsity pattern of the matrices considered [103]. Therefore, auto-tuning frameworks (i.e., for automated performance tuning) specific for GPGPUs have been proposed to adjust at runtime the matrix format and parameters according to the input matrix characteristics (e.g., matrix size, number of nonzeros, sparsity pattern) and/or the specific architecture. In this section we briefly review them, together with techniques for performance optimizations that do not depend on a specific storage format.

A model-based auto tuning framework designed to choose at runtime the matrix-dependent parameters was presented by Choi et al. [22]. It is based on a generic model of GPGPU execution time which is instantiated for SpMV with off-line benchmarking data. However, the model is specific for the BELLPACK and BCSR storage formats.

Li et al. [63] proposed SMAT, a sparse matrix-vector multiplication auto-tuning system, to bridge the gap between specific optimizations and general-purpose usage. SMAT presents a unified programming interface for different storage formats: the user has to provide in input its matrix in CSR format and SMAT automatically determines the optimal storage format and implementation on a given architecture. The auto-tuning strategy exploits a black-box machine learning model to train representative performance parameters extracted from the sparse matrix pattern and architectural configuration.

A machine learning approach using classification trees to automatically select the best sparse storage scheme among CSR, ELLPACK, COO, and ELL-COO on the basis of input-dependent features on the sparse matrix has been also presented in [90].

A heuristic-based auto-tuning framework for SpMV on GPGPUs has been also proposed in [2]. Given a sparse matrix, their framework delivers a high performance SpMV kernel which combines the use of the most effective storage format and tuned parameters of the corresponding code targeting the underlying GPGPU architecture.

There are also extensive efforts on performance models specific for the SpMV application on GPGPUs [58, 45, 47, 64]. Kubota and Takahashi [58] proposed an algorithm that automatically selects the optimal storage scheme.

Guo et al. [45] presented an integrated analytical and profile-based performance modeling and optimization analysis tool to predict and optimize the SpMV performance on GPGPUs. Their model allows to predict the execution times of CSR, ELL, COO, and HYB kernels for a target sparse matrix. Their goal is to find a storage format (single or a combination of multiple formats) from those available in order to maximize performance improvement. However, their analytical models do not consider in details the sparsity pattern of the input matrix. Guo and Wang [47] also proposed an extension of the tool to provide inter-architecture performance prediction for SpMV on NVIDIA GPU architectures.

A probabilistic method of performance analysis and optimization for SpMV on GPGPU has been recently proposed by Li et al. [64]. Differently from [45],

the proposed method analyzes the distribution pattern of nonzero elements in a sparse matrix and defines a related probability mass function. Once the probability function for the target matrix is built, performance estimation formulas for COO, CSR, ELL, and HYB storage formats can be established for the target matrix and the storage structures of these formats. The SpMV performance using these formats is then estimated using the GPGPU hardware parameters and finally the format that achieves the best performance is selected. The advantage of the approach lies in its being general, because it does not depend on the GPGPU programming language and architecture.

Some works have focused on performance optimization of SpMV on GPGPUs exploiting some specific approach independently from the storage format [85, 43]. Pichel et al. [85] explored performance optimization using reordering techniques on different matrix storage formats. These techniques evaluate the sparsity pattern of the matrix to find a feasible permutation of rows and columns of the original matrix and aim to improve the effect of compression. The problem with applying a reordering technique is that it changes the inherent locality of the original matrix.

Grewe and Lockmotov [43] proposed an abstract, system-independent representation language for sparse matrix formats, that allows a compiler to generate efficient, system-specific SpMV code. The compiler supports several compilation strategies to provide optimal memory accesses to the sparse matrix data structures and the generated code provides similar and sometimes even better performance compared to hand-written code. However, the automated tuning of code can be seriously time consuming because it exhaustively evaluates the configuration space.

Some efforts have been devoted to study SpMV on hybrid GPU/CPU platforms [18, 51, 109, 74]. We discussed in [18] how design patterns for sparse matrix computations and object-oriented techniques allow to achieve an efficient utilization of a heterogeneous GPGPU/CPU platform; we also considered static load balancing strategies for devising a suitable data decomposition on a heterogeneous GPGPU/CPU platform.

Indarapu et al. [51] proposed heterogeneous algorithms for SpMV on a GPGPU/CPU heterogeneous platform. Their algorithms are based on work division schemes that aim to match the right workload for the right device.

Yang et al. [109] have recently presented a partitioning strategy of sparse matrices based on probabilistic modeling of nonzeros in a row. The advantages of the proposed strategy lie in its generality and good adaptability for different types of sparse matrices. They also developed a GPGPU/CPU hybrid parallel computing model for SpMV in a heterogeneous computing platform.

Finally, in a more general context than sparse matrices, the MAGMA project [74] aims at designing linear algebra algorithms and frameworks for hybrid multicore and multi-GPU systems. A general strategy is to assign small tasks to multicores and large tasks to GPUs.

5 Experimental Evaluation

We have presented a number of different storage formats for the SpMV kernel on GPGPU; it is now appropriate to see how well they perform in actual test cases. To this end, we have elected to use:

- A subset of the formats reviewed, based on the availability of the software;
- A number of GPU models, with different architectural features;
- A set of test matrices, coming from different application fields and exhibiting different patterns.

We first present in Section 5.1 the testing environments; in Section 5.2 we then discuss the performance of the SpMV kernels using the throughput as performance metric, and in Section 5.3 we evaluate the results against a roofline model of attainable performance on the GPU computing platform. In Section 5.4 we analyze the time and space overheads associated to the considered subset of storage formats; finally, in Section 5.5 we summarize the analysis and draw some guidelines.

5.1 Testing Environments

5.1.1 Sparse Storage Formats

The set of storage formats includes:

- Two CSR variants: the one originally published by [87] and currently included in the NVIDIA cuSPARSE library, and the one published by [79]; they will be referred to by the names CSR and JSR, respectively;
- The HYBrid format from cuSPARSE [14, 82];
- The SELL-P format implemented in MAGMA 1.7.0 [74];
- The ELLPACK-like and Hacked ELLPACK formats from our group, indicated with ELL and HLL, respectively [10];
- The Hacked Diagonal (HDI) format from our group [10];

In the tests regarding formats that employ hacks (i.e., HLL and HDI), the hack size has been chosen equal to the size of one warp, so to minimize the padding overhead; moreover, we have applied no reordering to the test matrices.

5.1.2 Test Matrices

The test matrices were taken mostly from the Sparse Matrix Collection at the University of Florida (UFL) [30], with the addition of three matrices generated from a model three-dimensional convection-diffusion PDE with finite difference discretization.

Table 3: Sparse matrices used in the experiments and their features

Matrix name	M	NZ	AVG NZR	MAX NZR	Description
cant	62451	4007383	64.2	78	Structural analysis problems
olafu	16146	1015156	62.9	89	
af_1_k101	503625	17550675	34.8	35	
Cube_Coup_dt0	2164760	127206144	58.8	68	
ML_Laplace	377002	27689972	73.4	74	
bcsstk17	10974	428650	39.1	150	
mac_econ_fwd500	206500	1273389	6.2	44	Macroeconomic model
mhd4800a	4800	102252	21.3	33	Electromagnetism
cop20k_A	121192	2624331	21.7	81	
raefsky2	3242	294276	90.8	108	Computational fluid dynamics problems
af23560	23560	484256	20.6	21	
lung2	109460	492564	4.5	8	
StocF-1465	1465137	21005389	14.3	189	
PR02R	161070	8185136	50.8	92	
RM07R	381689	37464962	98.2	295	
FEM_3D_thermal1	17880	430740	24.1	27	Thermal diffusion problems
FEM_3D_thermal2	147900	3489300	23.6	27	
thermal1	82654	574458	7.0	11	
thermal2	1228045	8580313	7.0	11	
thermomech_TK	102158	711558	7.0	10	
thermomech_dK	204316	2846228	13.9	20	
nlpkkt80	1062400	28192672	27.0	28	Nonlinear optimization
nlpkkt120	3542400	95117792	27.3	28	
pde60	216000	1490400	6.9	7	Convection–diffusion PDE unit cube, 7-point stencil.
pde80	512000	3545600	6.9	7	
pde100	1000000	6940000	6.9	7	
webbase-1M	1000005	3105536	3.1	4700	Graph matrices — web connectivity, circuit simulation.
dc1	116835	766396	6.6	114190	
amazon0302	262111	1234877	4.7	5	
amazon0312	400727	3200440	8.0	10	
roadNet-PA	1090920	3083796	2.8	9	
roadNet-CA	1971281	5533214	2.8	12	
web-Google	916428	5105039	5.6	456	
wiki-Talk	2394385	5021410	2.1	100022	

Table 3 summarizes the matrices characteristics; for each matrix, we report the matrix size, the number of nonzero elements, the average and maximum number of nonzeros per row. The sparse matrices we selected from the UFL collection represent different kinds of real applications including structural analysis, economics, electromagnetism, computational fluid dynamics, thermal diffusion, graph problems. The UFL collection has been previously used in most of the works regarding SpMV on GPGPUs, among them [7, 11, 22, 38, 76, 78, 79]. The UFL collection subset we selected includes some large sparse matrices such as Cube_Coup_dt0, StocF-1465, nlpkkt120 and webbase-1M, i.e., matrices having more than a million rows and up to a hundred million nonzeros; moreover, the optimization and graph matrices have a structure that is significantly different from PDE discretization matrices. Since the irregular access pattern to the GPGPU memory can significantly affect performance of GPGPU implementa-

tions when evaluating larger matrices [27], it is important to include large sparse matrices and matrices with different sparsity patterns. Some of the test matrices from graph problems, e.g., webbase-1M and wiki-Talk, exhibit a power-law distribution in the number of nonzeros per row; since the formats we are testing are row-oriented, a large difference between the average and maximum number of nonzeros per row marks difficult test cases in which performance suffers significantly.

The model pdeXX sparse matrices arise from a three-dimensional convection-diffusion PDE discretized with centered finite differences on the unit cube. This scheme gives rise to a matrix with at most 7 nonzero elements per row: the matrix size is expressed in terms of the length of the cube edge, so that the case pde60 corresponds to a $(60^3 \times 60^3 = 216000 \times 216000)$ matrix. We already used this collection in [19].

5.1.3 Hardware and Software Platforms

The performance measurements were taken on four different platforms, whose characteristics are reported in Table 4. Error-Correcting Code (ECC) was disabled on the M2070, on this particular card it makes a substantial difference in memory bandwidth. The experiments were run using the GCC compiler suite and the CUDA programming toolkit; on all platforms we used GCC 4.8.3 and CUDA 6.5, and all tests have been run in double precision.

Table 4: Test platforms characteristics

Platform	1	2	3	4
CPU	Intel Xeon E5645	AMD FX 8120	Intel Xeon E5-2670	Intel Core i7 4820
GPU	M2070 Fermi	GTX 660 Kepler	K20M Kepler	K40M Kepler
Multiprocessors	14	5	13	15
Cores	448	960	2496	2880
Clock (MHz)	1150	1033	706	745
DP peak (GFlop/s)	515.2	82.6	1170	1430
Bandwidth (GB/s)	150.3	144.2	208	288
Compute capability	2.0	3.0	3.5	3.5

To collect performance data we have mostly used the benchmark program contained in our PSBLAS software; our GPU plugin provides wrappers for the cuSPARSE formats as well as for our owns, thereby making it easy to test multiple format variations. For the JSR format we have used the test program provided by its authors; it must be noted that this has apparently been designed to handle matrices coming from PDE discretizations, and it signaled with run-time error messages that the graph matrices in many cases lack coefficients on the main diagonal.

For the SELL-P format we have used the test program provided by the MAGMA library. In the tables we report the performance obtained by using blocksize 32 and alignment 4; we tested many other combinations, but this proved to be quite good on average, either the best or very close to it.

We have also run some tests with the ELLR-T format from [97], with levels of performance in the same range as our ELL variation(s); however the software on the author's website is available in binary format only for the CUDA 5.0 environment, hence it has not been included in the current data set.

As described in Section 4.6, the Hybrid format available in the NVIDIA cuSPARSE library is documented to be a mixture of ELLPACK and COO. The cuSPARSE library allows the user to control the partition of the input matrix into an ELLPACK part and a COO part; for the purposes of this comparison we always used the library default choice.

5.2 SpMV Performance

Our performance results are detailed in Tables 5-8 and in Figures 13(a) through 14(b). The entries in Table 8 highlighted with symbols are those used in the performance model of Section 5.3.

As a general observation, it is rarely the case that the CSR and JSR formats (the latter only on the Kepler cards) are the fastest option. Among the few such cases are the PR02R, RM07R and raefsky2 matrices on the K40M (platform 4); these are matrices with densely populated rows, having an average of 50, 98, and 90 coefficients per row, respectively. The JSR code is sometimes faster than the cuSPARSE CSR; this happens with the Cube_Coup_dt0 matrix, which has on average 59 nonzeros per row, and cant and olafu matrices on the K40M.

On all platforms the ELL, HLL, and HYB formats are almost always quite close in speed; the advantage goes either way and there is no clear winner.

For matrices arising from the discretization of PDEs, the ELLPACK-derived formats (including HYB) are typically the best. When the ratio between MAXNZR and AVGNZR is close to 1, ELL is the best choice, e.g., for ML_Laplace and FEM_3D_thermal matrices because the padding overhead is sufficiently small. As the ratio grows while still remaining within one order of magnitude (e.g., RM07R, bcsstk17, Cube_Coup_dt0, cop20k_A), HLL and HYB are able to keep the memory overhead under control and provide increasingly better performance with respect to ELL. In the limit case of StocF-1465, the memory overhead is so large that we are unable to employ the ELL format on any platform.

There are two main sets of exceptions to the above rules of thumb: the pdeXX, FEM_3D and nlpkk matrices, and the graph matrices.

For the pdeXX matrices, the HYB, ELL and HLL formats are very close. However, these matrices have an extremely regular structure along seven diagonals, making them good candidates for the HDI format; indeed, the HDI format clearly outperforms all others on the pdeXX matrices. This is also true for FEM_3D_thermal2; if we look at the sparsity pattern of this matrix, as shown in Figure 10, we see that most of the coefficients are clustered around the main diagonal; moreover, the off-diagonal part is made up of short diagonal segments,

Table 5: Detailed performance on platform 1: M2070

Matrix	Execution speed (GFLOPS)					
	CSR	HDI	ELL	HLL	HYB	SELL-P
cant	10.0	18.8	17.5	16.9	16.3	13.3
olafu	9.5	10.2	14.0	15.9	13.5	11.9
af_1_k101	11.5	18.5	20.9	19.9	19.6	14.4
Cube_Coup_dt0	10.7	—	18.2	20.2	19.9	14.5
ML_Laplace	12.4	20.3	21.6	21.4	18.8	14.8
bcsstk17	9.1	5.8	5.0	10.8	8.9	8.9
mac_econ_fwd500	7.2	1.5	2.8	5.1	7.0	3.7
mhd4800a	6.7	6.3	10.1	7.8	3.9	6.1
cop20k_A	8.7	1.6	5.0	10.0	9.6	6.7
raefsky2	8.3	3.2	15.1	15.3	5.0	11.5
af23560	8.9	14.1	15.4	13.7	14.3	10.2
lung2	7.3	4.2	9.7	9.1	7.5	5.4
StocF-1465	9.5	3.1	—	14.0	13.4	—
PR02R	9.1	13.9	11.9	14.6	14.0	11.2
RM07R	11.1	7.4	7.2	12.4	12.6	8.3
FEM_3D_thermal1	8.7	16.1	14.8	13.2	15.5	10.4
FEM_3D_thermal2	10.3	21.2	17.0	15.5	16.4	12.0
thermal1	—	—	—	—	—	—
thermal2	8.1	1.6	7.7	8.2	8.9	5.8
thermomech_TK	5.7	0.9	3.9	4.2	5.4	—
thermomech_dK	9.3	1.4	10.7	11.6	9.4	6.5
nlpkkt80	9.7	23.3	18.6	17.7	20.1	13.8
nlpkkt120	9.7	23.1	19.0	17.8	19.4	13.6
pde060	8.6	19.5	17.1	15.7	17.3	9.2
pde080	8.7	20.4	17.6	16.2	18.0	9.3
pde100	8.7	20.6	17.3	15.8	18.2	9.0
webbase-1M	5.4	0.6	—	2.7	8.7	2.9
dc1	2.5	—	—	—	7.3	—
amazon0302	6.4	0.8	4.1	4.0	5.3	3.0
amazon0312	6.2	0.8	3.7	3.7	4.5	3.8
roadNet-PA	5.7	1.6	5.3	6.9	8.0	3.8
roadNet-CA	5.7	1.7	4.2	6.9	8.3	—
web-Google	2.9	0.5	—	1.6	2.7	—
wiki-Talk	2.2	0.2	—	0.2	3.2	—

as shown in the enlarged plot on the right, therefore making the HDI format a suitable choice. Similar considerations apply to nlpkkt80 and nlpkkt120.

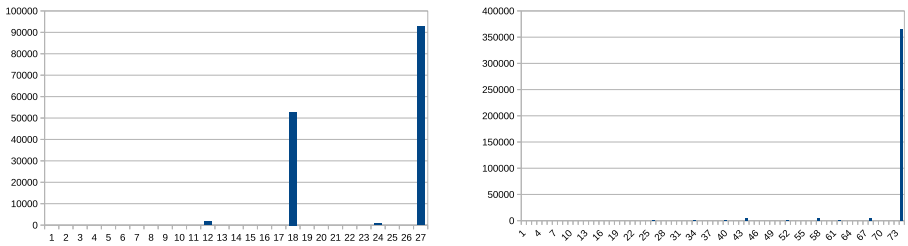
The graph matrices are completely different; many of them, including webbase-1M, dc1, web-Google, and wiki-Talk, have a power-law distribution of entries; webbase-1M and web-Google are web connectivity matrices, whereas dc1 is a connectivity matrix for a circuit simulation. The histograms of row lengths for webbase-1M and wiki-Talk are shown in Figure 11; the range of both x and y axes has been clipped to improve the readability of the graph. By contrast, the roadNet matrices describe the road networks in the states of Pennsylvania and California respectively, and they do not exhibit the same power-law distribution. The sparsity pattern of the roadNet matrices is very scattered and irregular (see Figure 12) and this affects the performance because of poor locality of memory accesses to the x vector.

Table 6: Detailed performance on platform 2: GTX660

Matrix	Execution speed (GFLOPS)					
	CSR	JSR	HDI	ELL	HLL	HYB
cant	12.9	8.4	16.4	15.3	16.0	18.1
olafu	11.4	8.6	10.3	12.4	14.8	13.7
af_1_k101	14.7	5.1	16.4	11.0	—	21.8
Cube_Coup_dt0	13.8	8.0	—	11.2	14.5	21.8
ML_Laplace	19.7	9.8	18.1	19.2	19.1	21.9
bcsstk17	9.1	5.8	5.8	4.8	11.7	7.6
mac_econ_fwd500	7.3	3.8	1.3	2.2	4.6	8.2
mhd4800a	5.4	3.8	6.6	9.0	7.3	3.4
cop20k_A	10.9	3.8	1.4	4.7	9.7	13.9
raefsky2	10.6	8.8	3.4	10.9	14.8	6.1
af23560	10.3	3.9	13.0	14.9	13.7	19.4
lung2	7.0	3.8	3.7	7.4	7.9	7.0
StocF-1465	11.7	5.0	—	—	12.9	17.7
PR02R	13.5	7.2	12.5	10.6	13.5	16.1
RM07R	17.5	9.0	5.3	6.5	11.5	14.6
FEM_3D_thermal1	9.7	4.5	15.3	14.2	13.1	17.9
FEM_3D_thermal2	13.9	4.5	18.9	15.4	14.3	19.6
thermal1	9.8	5.1	1.6	7.1	7.5	9.5
thermal2	12.4	4.9	1.1	5.4	5.2	12.0
thermomech_TK	9.3	3.4	0.8	4.1	4.2	8.8
thermomech_dK	9.5	5.2	1.1	8.7	10.7	15.1
nlpkkt80	14.3	—	20.0	16.6	16.2	21.6
nlpkkt120	14.6	—	—	—	—	21.1
pde060	11.9	5.6	19.6	15.4	14.4	19.2
pde080	12.8	5.6	20.2	15.6	14.0	19.3
pde100	12.7	5.6	18.2	14.2	13.4	19.2
webbase-1M	5.0	3.9	0.6	—	1.5	9.5
dc1	—	0.3	—	0.1	0.1	—
amazon0302	7.6	—	—	3.6	3.5	11.3
amazon0312	6.3	—	—	3.0	3.0	7.7
roadNet-PA	6.6	—	—	4.4	6.2	8.8
roadNet-CA	6.7	—	—	3.5	6.3	8.9
web-Google	2.9	—	—	—	1.5	3.3
wiki-Talk	0.4	—	—	—	0.2	3.5

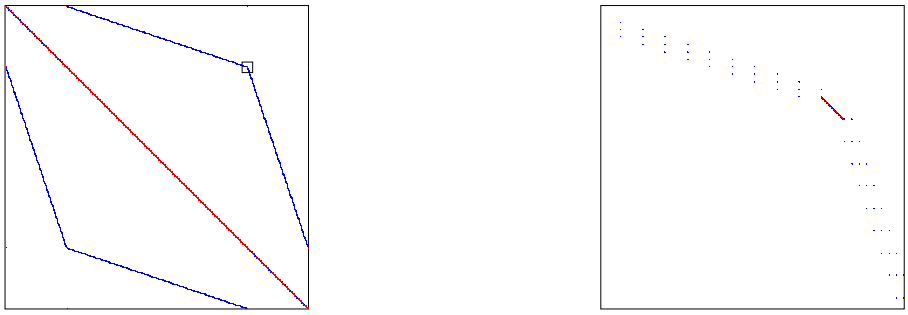
The difference in distribution of nonzeros per row between roadNet-CA and webbase-1M is not immediately apparent if we only look at the sparsity pattern, as shown in Figure 12. The webbase-1M matrix has an average of 3.1 nonzeros per row, and 915506 out of 1000005 rows have up to 3 nonzeros. Of the remaining rows, 51087 have more than 4 entries, 55 rows have more than 1000 nonzeros, with the largest row having 4700 entries, and the next largest 3710. DC1 has an average of 6.6 nonzeros per row, and 98705 out of 116835 of its rows have up to 7 nonzeros. All rows but two have up to 288 nonzeros; however the two largest rows have 47193 and 114190 nonzeros respectively, i.e., one row is almost full.

Thus, for power-law matrices there are a few scattered rows that are much longer than all the others, and therefore the ELL and HLL schemes suffer because they assign one thread per row (indeed, ELL cannot be used due to excessive memory footprint); the threads having long rows take a long time to



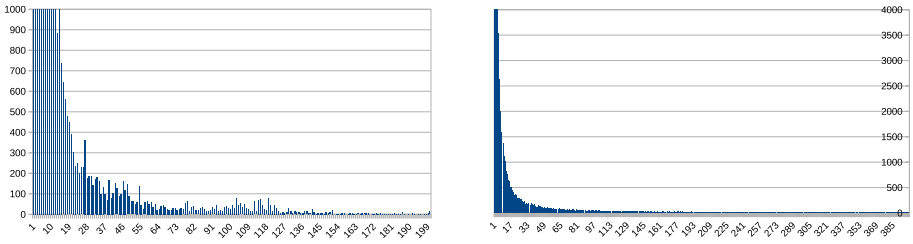
(a) FEM_3D_thermal2 (b) ML_Laplace

Figure 9: Row length histograms



(a) General (b) Off-diagonal detail

Figure 10: Sparsity pattern of FEM_3D_thermal2



(a) webbase-1M (b) wiki-Talk

Figure 11: Row length histograms

Table 7: Detailed performance on platform 3: K20M

Matrix	Execution speed (GFLOPS)						
	CSR	JSR	HDI	ELL	HLL	HYB	SELL-P
cant	13.3	16.7	20.3	19.2	20.1	18.2	19.6
olafu	12.7	15.6	7.4	15.9	16.8	13.7	17.4
af_1_k101	15.2	11.0	19.7	22.3	22.5	21.8	21.7
Cube_Coup_dt0	13.8	15.9	—	19.3	22.4	21.8	21.9
ML_Laplace	19.9	18.2	21.9	23.3	23.0	21.9	22.3
bcsstk17	11.4	11.3	3.1	5.9	11.3	7.7	13.0
mac_econ_fwd500	7.7	7.8	1.6	3.2	6.6	8.2	5.5
mhd4800a	10.1	7.2	0.7	8.3	8.9	3.4	7.8
cop20k_A	11.5	8.7	1.8	6.0	15.1	13.9	11.8
raefsky2	15.3	14.8	2.1	13.8	13.7	6.1	14.9
af23560	12.8	8.4	3.5	19.8	15.2	19.4	14.9
lung2	8.1	7.2	3.6	10.8	8.7	7.0	8.6
StocF-1465	11.7	10.6	3.4	—	16.1	17.7	13.3
PR02R	13.7	15.1	14.9	12.5	16.5	16.1	16.6
RM07R	17.6	19.0	8.2	7.9	15.3	14.6	12.3
FEM_3D_thermal1	12.3	9.4	3.1	18.4	16.0	17.9	14.6
FEM_3D_thermal2	14.4	9.9	23.3	19.3	18.6	19.7	16.9
thermal1	11.5	9.7	1.9	12.0	12.5	9.6	9.5
thermal2	12.5	10.1	1.8	12.1	12.8	12.0	9.9
thermomech_TK	10.6	9.5	1.2	11.7	11.7	9.0	—
thermomech_dK	9.7	10.6	1.5	13.0	14.2	15.1	11.8
nlpkkt80	14.4	—	25.5	21.1	21.2	21.6	20.0
nlpkkt120	14.6	—	25.8	20.8	20.7	21.1	20.1
pde060	12.8	11.1	10.9	19.1	18.1	19.1	14.1
pde080	13.2	11.3	24.8	19.5	18.6	19.4	14.0
pde100	13.0	11.3	25.8	19.4	18.6	19.2	14.4
webbase-1M	5.0	7.3	0.5	—	2.5	9.5	6.1
dc1	0.1	—	—	—	—	7.7	—
amazon0302	8.0	—	1.0	11.1	9.4	11.4	6.4
amazon0312	6.4	—	0.9	7.6	6.8	7.7	6.8
roadNet-PA	6.8	—	1.8	6.2	9.8	8.8	6.8
roadNet-CA	6.8	—	1.8	4.7	9.6	8.9	—
web-Google	2.9	—	0.5	—	2.3	3.3	—
wiki-Talk	0.4	—	0.2	—	0.2	3.5	—

complete and tie up their warps. The HYB format suffers somewhat less, because it can exploit multiple threads in the COO part; this is helped on the Kepler architecture by the fact that the atomic operations are much faster than in previous architectures. Still, the overall performance is much less than for most other matrices, and a proper design for this kind of sparsity pattern would have to employ other techniques. Some efforts that address the peculiarities of power-law matrices are the CSR-based ACSR format in [7] and the hybrid TILE-COMPOSITE format in [111], that we described in Sections 4.2 and 4.6, respectively; graph techniques may also be employed to improve communication patterns [59].

A graphical comparison among the formats is depicted in Figures 13(a) through 14(b) for platforms 1 and 4 using the *performance profiles* proposed in [32]. The performance profile is a normalized cumulative distribution of performance; adapting the definitions in [32] to our present situation, we define

Table 8: Detailed performance on platform 4: K40M

Matrix	Execution speed (GFLOPS)						
	CSR	JSR	HDI	ELL	HLL	HYB	SELL-P
cant	14.7	19.7	24.8	23.8	24.0	20.6	23.3
olafu	14.0	18.4	9.0	19.2	19.6	15.7	20.4
af_1_k101	17.3	12.5	24.4	28.8	28.1	27.0	26.8
Cube_Coup_dt0	15.4	18.8	—	24.9	28.1	27.3	26.8
ML_Laplace	23.4	21.8	26.7	○29.5	28.6	27.1	27.5
bcsstk17	12.8	12.8	3.8	7.5	12.8	8.6	13.2
mac_econ_fwd500	9.1	8.9	1.9	3.8	7.7	9.8	6.0
mhd4800a	11.7	8.3	0.9	11.0	10.2	4.0	8.8
cop20k_A	13.5	10.2	2.1	7.5	18.9	16.2	13.9
raefsky2	17.4	17.1	2.6	15.9	15.2	6.7	16.8
af23560	14.8	9.5	4.3	20.9	21.2	23.8	17.5
lung2	9.5	8.5	4.4	12.3	11.0	8.3	10.6
StocF-1465	13.2	12.2	4.2	—	20.2	22.2	16.3
PR02R	15.3	17.3	17.8	16.5	20.2	20.0	17.4
RM07R	20.8	22.4	9.8	9.9	18.6	17.7	12.8
FEM_3D_thermal1	14.2	10.8	3.8	20.6	18.3	21.3	17.2
FEM_3D_thermal2	16.9	11.5	28.2	24.1	23.1	24.6	21.8
thermal1	13.2	11.3	2.1	11.4	15.9	11.0	11.5
thermal2	14.6	11.9	2.2	◇ 11.8	16.0	14.3	12.0
thermomech_TK	12.3	11.3	1.4	8.2	14.3	10.1	—
thermomech_dK	11.4	12.5	1.8	16.7	19.2	18.7	13.7
nlpkkt80	16.9	—	31.2	26.7	26.9	27.2	26.0
nlpkkt120	16.9	—	32.3	26.2	26.1	26.9	25.0
pde060	15.3	13.1	13.3	23.0	22.4	23.9	18.0
pde080	15.8	13.4	29.8	23.7	23.2	24.5	18.8
pde100	16.1	13.5	30.8	23.7	23.2	24.4	19.2
webbase-1M	5.0	7.8	0.5	—	2.6	11.1	7.4
dc1	0.1	0.2	—	—	—	8.7	—
amazon0302	9.6	—	1.2	9.3	12.0	14.5	7.5
amazon0312	7.9	—	1.1	8.0	8.4	10.0	9.2
roadNet-PA	7.6	—	2.1	7.4	12.1	10.6	9.4
roadNet-CA	7.6	—	2.2	5.8	12.0	10.5	—
web-Google	3.3	—	0.7	—	⋆ 2.6	3.9	2.7
wiki-Talk	0.4	—	0.2	—	0.2	3.6	—

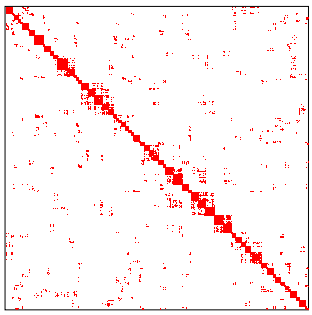
$S_{f,m}$ as the speed attained by using format $f \in \mathcal{F}$ on matrix $m \in \mathcal{M}$, and the *performance ratio*

$$r_{f,m} = \frac{\max\{S_{f,m} : f \in \mathcal{F}\}}{S_{f,m}}.$$

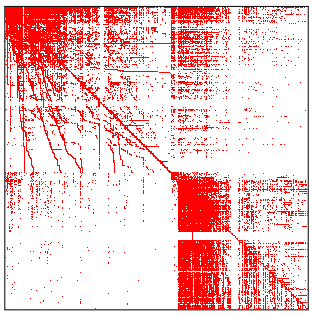
We also choose a value $r_X \geq r_{f,m}$ for all possible formats and matrices, and we assign this ratio to those cases where a format is unable to handle a matrix (e.g., due to memory usage). Denoting by n_M the size of the set $\mathcal{M}$, the performance profile is then defined as

$$\rho_f(\tau) = \frac{1}{n_M} \text{size}\left\{m \in \mathcal{M} : r_{f,m} \leq \tau\right\},$$

that is, the cumulative distribution of the probability that for format f the ratio $r_{f,m}$ is within a factor τ of the best possible value. If $\rho_f(\tau)$ levels asymptotically



(a) roadNet-CA



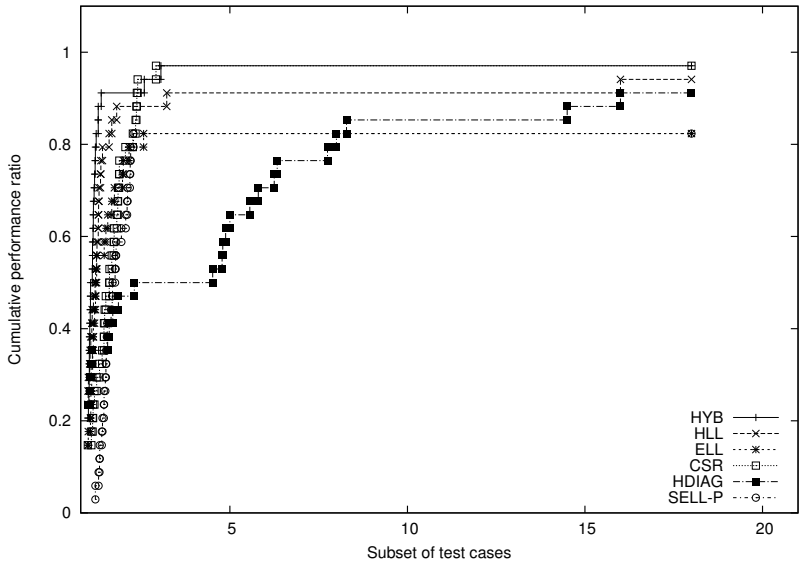
(b) webbase-1M

Figure 12: Sparsity patterns

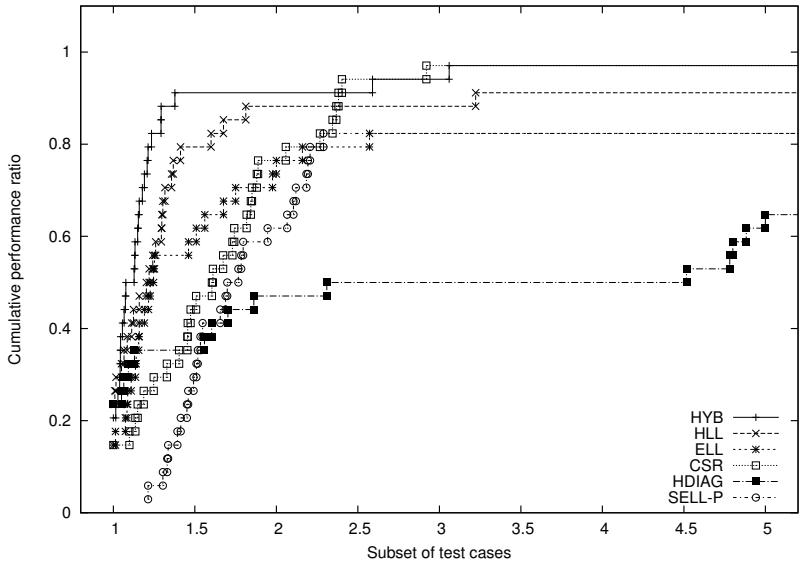
at less than 1, it means that there are some matrices that the format is unable to handle; thus, a curve that is "above" another one means that the corresponding format is "generally better" than the other. Note that the value $\rho_f(1)$ is the probability that format f is the *best* one for *some* matrix in the test set.

With these definitions in mind, we can observe that:

- The curves for all formats start with $\rho_f(1) > 0$, which means that *all* formats are *best* for *some* matrix;
- On both platforms, the HYB and CSR formats level at probability 1, which means that they can handle all test cases; this is also true of HLL on the K40M but not on the M2070, due to the memory footprint;
- From the zoomed view in Figures 13(b) and 14(b) the HYB and HLL formats have the fastest growth, which means that they are very good "general purpose" formats; on the M2070, HYB is above HLL, whereas on the K40M they are essentially equivalent;
- The CSR format curve has a much slower growth, but in the end it handles all cases, so it is a good default choice if we need to handle many different patterns in a reliable, if possibly suboptimal, way;
- SELL-P is also behaving effectively on the K40 but much less so on the M2070;
- JSR is hampered by the fact, already noted before, that the test program does not handle well the graph matrices;
- The curve for HDIA falls significantly below the others, confirming that it is a very specialized storage format, and should only be used in cases that are well suited to its structure.

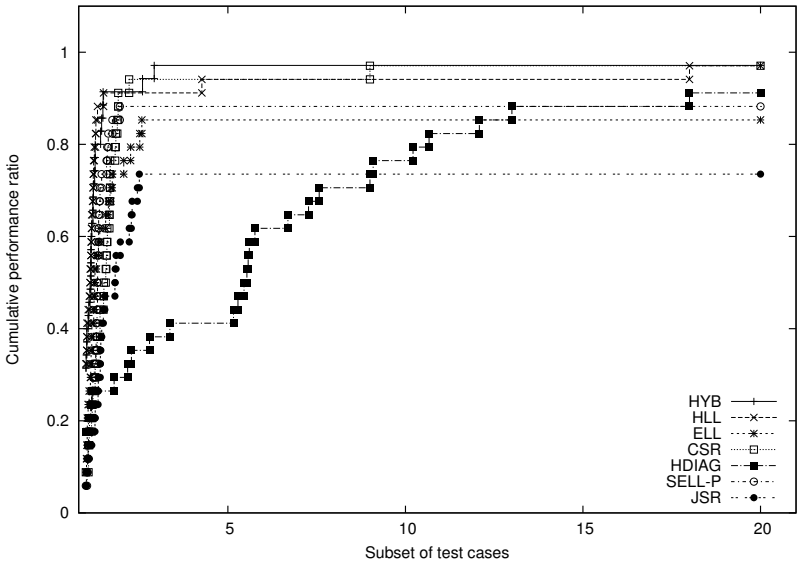


(a) Full view

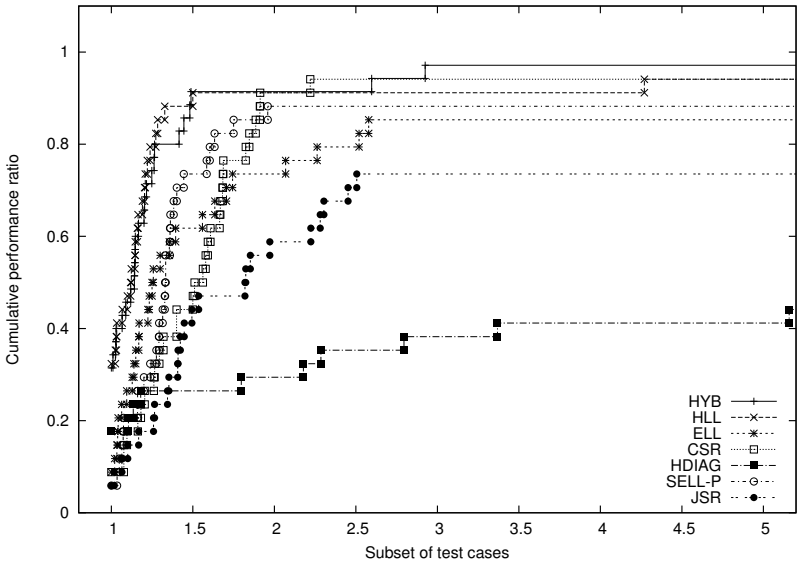


(b) Detailed view

Figure 13: Performance profile on platform 1: M2070



(a) Full view



(b) Detailed view

Figure 14: Performance profile on platform 4: K40M

5.3 Roofline Models

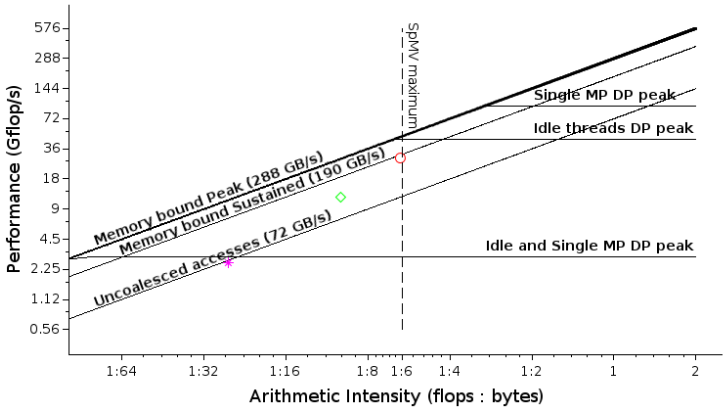


Figure 15: Roofline model of platform 4: K40M

How good are the performance numbers we have just discussed?

To enable an evaluation of the efficiency of the SpMV implementations we use the roofline model [102] to compute a realistic upper bound on the attainable performance; in Figure 15 we show the relevant graph for the K40M, that is, platform 4, with three matrix/format pairs highlighted (see below).

Global memory bandwidth takes a major role in SpMV performance because of its low arithmetic intensity, which is the ratio between the number of operations and the amount of memory being accessed. In the SpMV routine $y \leftarrow Ax$, the number of arithmetic operations on floating point values is $2NZ - M$, while the amount of memory being accessed, for double precision routines, is at least $8(NZ + M + N)$ bytes. For a square matrix with $NZ \gg M$, this ratio becomes $1/4$; this also represents an upper bound for the arithmetic intensity of the SpMV routine applied to any general-purpose matrix, since it would never execute more than $2NZ$ operations and it would never access less than $8NZ$ bytes. A refined estimate for the arithmetic intensity can be obtained by observing that most formats explicitly store one index per coefficient; therefore the memory is at least $12NZ + 8(M + N)$, and the ratio becomes $1/6$.

As we can see from Figure 15, the main performance bottleneck of SpMV is clearly due to memory bandwidth: the throughput of the ALUs cannot be fully exploited, independently of the quality of the implementation. A realistic estimate for peak throughput is obtained by multiplying the arithmetic intensity by the bandwidth, therefore a reasonable value would be $1/6|BW|$ (flops/sec). Considering for the K40M the sustained bandwidth measured in [5] at $190GB/s$, this gives a best possible performance of 32 GFLOPS.

There are other constraints, such as scattered accesses and work imbalance,

that may enter into our considerations:

Single MP DP peak This is the arithmetic throughput in double precision when the distribution of work is such that only one of the MultiProcessors in the device is active. This may happen if one block of threads has much more work to perform than the others, or equivalently, when there is a significant inter-block load imbalance. For the K40M, this value is 95 GFLOPS;

Idle threads DP peak This is the throughput in the case where in each warp only one thread out of 32 is active, or equivalently when there is significant intra-warp load imbalance; for the K40M this value is 45 GFLOPS;

Idle-Single MP DP peak This is the throughput when inter-block, intra-block and intra-warp imbalance all occur; in this case, the peak throughput is that of a single ALU, at about 3 GFLOPS.

These bounds are relevant for the power-law test matrices, where a few rows are much more populated than the average.

The three highlighted points are located using the measurements in Tables 8 and 9:

- ○ ML_Laplace with ELL;
- ◇ thermal2 with ELL;
- ⋆ web-Google with HLL.

For ML_Laplace we see that we are running very close to the maximum possible performance; thermal2 has a much heavier footprint, but it is also achieving a somewhat lower effective bandwidth. Finally, web-Google has a power-law data distribution, therefore its performance is close to the Idle-Single MP/DP peak.

5.4 SpMV Overheads

We now focus on space and time overheads that may affect the overall performance of the application using the SpMV kernel. We first analyze the matrix structure memory footprint of the considered formats and then discuss the conversion (transformation) overhead. These additional evaluation criteria are sometimes neglected by authors presenting a new format; as noted in [60], they are necessary to make a fully informed choice in the selection of a suitable format taking into account the overall application performance, not relying solely on the SpMV throughput.

In Table 9 we report the memory footprint of the various storage formats, measured in bytes per nonzero element. The entries highlighted with symbols are those used in the performance model of Sec. 5.3. As explained in [19], in our library implementation we have a set of objects that have a dual storage, on the CPU side and on the GPU side; the two are mirror images, and we only report the memory footprint of one image.

Table 9: Memory footprint (Bytes/nonzero)

Matrix name	CSR	HDI	ELL	HLL
cant	12.06	18.39	14.71	13.75
olafu	12.06	25.26	17.13	14.11
af_1_k101	12.11	19.62	12.28	12.28
Cube_Coup_dt0	12.07	—	14.02	12.41
ML_Laplace	12.05	17.81	∘12.20	12.19
bcsstk17	12.10	41.88	46.30	16.15
mac_econ_fwd500	12.65	239.24	86.93	44.96
mhd4800a	12.19	32.67	18.96	18.96
cop20k_A	12.18	214.40	45.27	14.66
raefsky2	12.04	27.26	14.46	12.87
af23560	12.19	19.23	12.66	12.62
lung2	12.89	74.82	23.11	21.68
StocF-1465	12.28	112.93	—	15.09
PR02R	12.08	25.73	21.88	15.79
RM07R	12.04	47.56	36.15	21.89
FEM_3D_thermal1	12.17	14.61	13.79	13.79
FEM_3D_thermal2	12.17	15.06	14.07	14.08
thermal1	12.58	193.12	20.14	16.06
thermal2	12.57	208.28	◇20.04	15.97
thermomech_TK	12.57	325.60	18.38	15.35
thermomech_dK	12.29	244.87	17.80	14.45
nlpkkt80	12.15	14.24	12.73	12.46
nlpkkt120	12.15	13.76	12.58	12.40
pde060	12.58	12.10	13.33	13.26
pde080	12.58	12.07	13.29	13.23
pde100	12.58	12.06	13.26	13.21
webbase-1M	13.29	204.68	—	40.60
dc1	12.61	236.70	—	80.37
amazon0302	12.85	331.24	14.43	14.46
amazon0312	12.50	318.29	16.03	15.81
roadNet-PA	13.42	217.11	41.04	20.67
roadNet-CA	13.43	206.98	54.15	20.74
web-Google	12.72	383.95	—	⋆ 52.05
wiki-Talk	13.91	376.24	—	166.96

The HYB format is opaque, and there is no API to retrieve its actual memory footprint, hence it is not listed in the table; moreover, since cuSPARSE provides a conversion routine from CSR, the CPU-side of HYB is stored in CSR.

Considering the amount of memory employed by the various storage formats, we see that in many cases HLL has a footprint that is quite close to that of CSR; this was indeed one of the design goals of HLL. For some matrices the ELL storage format is essentially equivalent to HLL, because the maximum and average row lengths are very close; this is true for instance of the af_1_k101 matrix. The pdeXX model matrices also have a very regular structure and similar memory footprint achieved by ELL, HLL, and CSR; however, these matrices also have a native diagonal structure that makes them natural candidates for HDI. Matrices without a natural (piecewise) diagonal structure have excessive memory overheads when stored in HDI.

The conversion of the input matrix from a basic format into a more sophis-

ticated representation introduces a preprocessing time that can be quite significant, even two orders of magnitude higher than the time needed for performing the SpMV. While this conversion overhead is usually amortized for applications such as iterative linear solvers that reuse the same sparse matrix over many invocations of the SpMV operator, it can offset the performance benefits of using the alternate format when the structure of the sparse matrix changes quite frequently (e.g., in graph applications) or a small number of iterations are needed in a solver. It is therefore very difficult to give general rules, and even to perform meaningful comparisons.

To ground our discussion with a concrete implementation, we have chosen to measure conversion overhead in the framework of our PSBLAS library. Of course we do not claim to cover all conceivable conversion strategies and implementations, and the reader should be aware that any conversion evaluation would be subject to change whenever different algorithms or different application contexts are involved.

As already mentioned, in our library we normally have a copy of data on the host side and on the device side; this means that the default way to convert from one format to another is to employ the CPU side conversion and then transfer the data to the GPU. This is not as bad as it sounds; indeed, some conversion algorithms require the use of auxiliary data structures that are not easily implemented on the device side. Moreover, if we are performing the conversion on the device, at some point we will have redundant copies of the data in the device memory; since the GPU does not have virtual memory, this can be a problem. As an example, in [26] we have an application where the matrices are built at the first time step and then at each time step we perform partial updates of the data structures on the device side, without regenerating the entire matrices.

In the present set of tests, we have matrices that are read by a benchmark program from file into the host main memory, and then have to be transferred in the device memory; therefore, data have to travel in any case from host to device. Moreover, to have a meaningful comparison among the conversion algorithms, we have to define a common initial state for the data structure.

We assume as our starting point a COO representation on the host (CPU) side with entries already sorted in row-major order. This reference format has been chosen because, as described in [39, 19], the COO representation is the most flexible when it comes to *building* the matrix, since the coefficients may be added in an arbitrary order, thus allowing the most freedom to the application programmer. Since our measurements are geared towards row-oriented formats, we apply a minimal preprocessing so that all format conversions start from the COO data already sorted in row-major order. Excluding the initial sorting phase from the timings reflects the fact that the test matrices come from very diverse applications, and each of them may well have a very different preferred setup order; including the sorting time we would be mixing completely different factors into our measurements. On the other hand, this means that our conversion times include the transfer of data from the host to the device over the PCI bus.

A sample of conversion timing data is reported in Table 10; all data have

been collected on platform 4 (Intel Core i7-4820, K40M). We explicitly note the following considerations:

- Conversion to CSR from an already sorted COO is extremely fast, since it only involves counting the entries in each row and applying a scan primitive to compute the offsets;
- Therefore, CSR conversion is the baseline for device-side, since it involves very little time beyond the absolute minimum needed to transfer the coefficient data from host to device memory across the PCI bus;
- Conversion to HYB is implemented by having a CSR format on the host, copying its data to the device, invoking the cuSPARSE conversion to HYB, and then releasing the temporary CSR data;
- For both ELL and HLL we have implemented the default CPU side conversion as well as methods split between the CPU and the GPU; the latter are the ones whose times are reported in the tables. Some preprocessing (essentially, counting offsets and maximum row occupancy) is performed on the CPU, then the coefficients and auxiliary data are copied onto the GPU where a CUDA kernel uses them to allocate and populate the ELL/HLL data structure;
- The HDI conversion at this time is only implemented on the host; the resulting data structure is then copied on the device side.

To present the timing data we first give the absolute timings measured in seconds; we then report the ratio of the conversion time to a single matrix-vector product to measure the conversion overhead (rounded to the next largest integer). Finally, we show the break-even point with respect to CSR, i.e., the number of iterations for which the accumulated time plus conversion time is shorter than with CSR. In some cases the CSR format is faster, and therefore the break-even is never attained. The values for HDI are quite large because the preprocessing takes place on the CPU side. These numbers should be treated carefully because they include the time it takes to transfer data from host to device; if the matrix can be reused and/or updated on the device, this would be reflected in a lower overhead.

The ELL, HLL, and HYB formats are almost always quite close in overhead, meaning that their conversion algorithms have a similar efficiency. Note that measuring the overhead in number of SpMV invocations, albeit quite natural and realistic, has the side effect of favoring CSR because its SpMV speed is usually *lower* than the others, hence the same conversion time is offset by a smaller number of iterations. Again, as in the speed measurements, the largest differences appear for the graph matrices.

5.5 Lessons Learned

The performance data discussed in this section allow us to state some rules of thumb:

- It is important to consider the conversion overhead: if the matrix (structure) is only used for a few products, it does not pay off to search for sophisticated data structures, CSR suffices. Having the same structure but different coefficients may call for a specialized coefficient update functionality, but the payoff is highly application dependent;
- It is possible to apply a global renumbering of the equations to minimize the matrix bandwidth; this may significantly affect the cache access patterns, but we do not discuss the issue in this paper for space reasons;
- Reordering only the rows of the matrix, as necessary in some formats (e.g. JAD), also entails an overhead because we have to reshuffle, at some point, either the output or the input vectors;
- The memory footprint can be a big problem, even for some formats like HLL; however:
- If the matrix comes from a PDE discretization, one should certainly try an ELLPACK-like format first, depending on the memory available, unless:
- If the matrix not only comes from a PDE, but also has a piecewise diagonal structure, then HDI is probably a very good idea;
- If the matrix comes from a graph problem, and has a power-law distribution in the number of entries, then the only option among the ones considered here is HYB, but even HYB is not very satisfactory; in this case, it may be appropriate to investigate more specialized formats [7, 111].

6 Conclusions

We presented a survey on the multiplication of a sparse matrix by a dense vector on GPGPUs, classifying and reviewing 71 storage formats that have been proposed in literature for the efficient implementation of this fundamental building block for numerous applications.

The importance of SpMV is testified not only by the large number of works we surveyed, but also by the continuously increasing number of publications that appear on the topic: during the review process of this paper we included 18 new publications. We evaluated the performance of a significant subset of the available storage formats considering different test matrices and GPGPU platforms. We considered multi-dimensional evaluation criteria, focusing not only the attainable SpMV throughput in FLOPS but also analyzing the matrix structure memory footprint and conversion overhead that may affect the overall performance of the application using the SpMV kernel.

Putting together the survey analysis and our own experience on the topic, we provide some possible directions for future work as well as some suggestions for authors that present their formats.

- To facilitate the performance comparison among different formats, we suggest to use normalized metrics, as we did in Section 5.4; some relevant proposals have been put forward in [60];
- Whenever allowed by the licensing, we would appreciate more authors to publish their implementation code. As reported in Section 4, authors publicly providing their implementation codes are in the minority (and sometimes the URL is not supplied in the paper): this complicates the task of providing a comprehensive performance comparison. Furthermore, even when the implementation code is available, it is not necessarily easy to integrate into existing software because of either lack of simple APIs or even source code, thus requiring a non negligible effort. For our part, our software is freely available at the URL `http://www.ce.uniroma2.it/psblas` and has been designed to make it as easy as possible to interface with.
- The ever increasing platform heterogeneity poses interesting research challenges to deal with. In the context of SpMV, possible approaches that can be explored to face heterogeneity include: (1) the definition of a single cross-platform storage format, as the CSR5 one for SIMD units presented in [67]; (2) the adoption of object-oriented design patterns (in particular the State pattern) that allow to switch among different storage formats at runtime, as proposed in [19]; (3) the study of partitioning strategies that aim to properly balance the load in hybrid GPU/CPU platforms, as presented in few already cited works [18, 51, 109].
- Specific sparsity patterns may require the adoption of ad-hoc storage formats for efficient computation: this is the case of matrices with block entries coming from PDE problems as well as large matrices with power-law distribution used by graph algorithms for big data analytics. Only few works have been devoted to this issue up to now.
- We have not touched on the problem of formats for storing sparse matrices on file, nor on the related issue of checkpointing in the context of massively parallel applications, see e.g., [61].

In conclusion, let us note that sparse matrices remain a central, vibrant and challenging research topic today as they have been in the past decades, and we expect to see many more developments in the path towards ever more powerful computing platforms.

Acknowledgements

We wish to thank the Associate Editor and the anonymous referees, whose insightful comments have been instrumental in improving the quality of this paper.

We gratefully acknowledge the support received from CINECA for project IsC14_
HyPSBLAS, under the ISCRA grant programme for 2014, and from Amazon with the AWS in Education grant programme 2014; this work was also partly supported by INdAM. We wish to thank Dr. John Linford of Paratools Inc. and Dr. Pasqua D'Ambra of CNR Italy, for their help in getting access to some of the test platforms.

We also acknowledge Dr. Giles for having informed us know that their code in [87] is now part of NVIDIA's cuSPARSE library and Dr. Mukunoki for having provided us their code in [79].

Table 10: Conversion overhead

Conversion time (seconds)					
Matrix name	CSR	HDI	ELL	HLL	HYB
Cube_Coup_dt0	0.411	—	0.588	0.585	0.592
ML_Laplace	0.092	0.384	0.131	0.131	0.131
cop20k_A	0.009	0.346	0.015	0.013	0.015
FEM_3D_thermal2	0.011	0.042	0.016	0.016	0.017
thermal2	0.027	1.008	0.036	0.034	0.041
thermomech_TK	0.003	0.131	0.004	0.004	0.005
nlpkkt80	0.090	0.327	0.126	0.125	0.129
pde100	0.022	0.079	0.028	0.028	0.031
webbase-1M	0.011	0.326	—	0.024	0.030
dc1	0.003	0.132	—	0.166	0.095
amazon0302	0.005	0.227	0.006	0.006	0.007
roadNet-CA	0.019	0.618	0.030	0.026	0.037
web-Google	0.017	1.239	—	0.039	0.033
wiki-Talk	0.022	1.475	—	0.303	0.117
Conversion time (Number of SpMV)					
Matrix name	CSR	HDI	ELL	HLL	HYB
Cube_Coup_dt0	25	—	58	65	64
ML_Laplace	39	186	70	68	65
cop20k_A	23	138	21	47	46
FEM_3D_thermal2	28	169	57	55	59
thermal2	23	127	26	33	35
thermomech_TK	25	125	23	39	34
nlpkkt80	27	179	59	59	62
pde100	25	176	48	47	56
webbase-1M	10	25	—	11	54
dc1	1	3	—	4	537
amazon0302	18	111	26	32	44
roadNet-CA	14	123	16	28	36
web-Google	6	84	—	10	13
wiki-Talk	1	24	—	7	43
Conversion time (break-even point)					
Matrix name	CSR	HDI	ELL	HLL	HYB
Cube_Coup_dt0	0	—	28	24	26
ML_Laplace	0	984	79	91	122
cop20k_A	0	∞	∞	37	95
FEM_3D_thermal2	0	183	40	45	40
thermal2	0	∞	∞	71	∞
thermomech_TK	0	∞	∞	65	∞
nlpkkt80	0	153	30	29	31
pde100	0	139	24	23	34
webbase-1M	0	∞	—	∞	28
dc1	0	∞	—	∞	4
amazon0302	0	∞	900	37	33
roadNet-CA	0	∞	∞	13	46
web-Google	0	∞	—	∞	39
wiki-Talk	0	∞	—	∞	5

References

[1] W. Abu-Sufah and A. Abdel-Karim. An effective approach for implementing sparse matrix-vector multiplication on graphics processing units. In *Proc. of 14th IEEE Int'l Conf. on High Performance Computing and Communication*, HPCC '12, pages 453–460, June 2012.

[2] W. Abu-Sufah and A. Abdel-Karim. Auto-tuning of sparse matrix-vector multiplication on graphics processors. In *Supercomputing*, volume 7905 of *LNCS*, pages 151–164. Springer-Verlag, 2013.

[3] Abal-Kassim Cheik Ahamed and Frederic Magoules. Fast sparse matrix-vector multiplication on graphics processing unit for finite element analysis. In *Proc. of IEEE 14th Int'l Conference on High Performance Computing and Communications*, HPCC '12, pages 1307–1314, 2012.

[4] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, 3rd edition, 1999.

[5] Hartwig Anzt, Moritz Kreutzer, Eduardo Ponce, Gregory D. Peterson, Gerhard Wellein, and Jack Dongarra. Optimization and performance evaluation of the IDR iterative Krylov solver on GPUs. *Int. J. High Perform. Comput. Appl.*, May 2016.

[6] Hartwig Anzt, Stanimire Tomov, and Jack Dongarra. Implementing a sparse matrix vector product for the SELL-C/SELL-C-σ formats on NVIDIA GPUs. Technical Report EECS-14-727, University of Tennessee, 2014.

[7] Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on GPUs for graph applications. In *Proc. of Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 781–792, 2014.

[8] Arash Ashari, Naser Sedaghati, John Eisenlohr, and P. Sadayappan. A model-driven blocking strategy for load balanced sparse matrix-vector multiplication on GPUs. *J. Parallel Distrib. Comput.*, 76:3–15, February 2015.

[9] D. Barbieri, V. Cardellini, and S. Filippone. Generalized GEMM applications on GPGPUs: experiments and applications. In *Parallel Computing: from Multicores and GPU's to Petascale*, ParCo '09, pages 307–314. IOS Press, 2010.

[10] Davide Barbieri, Valeria Cardellini, Alessandro Fanfarillo, and Salvatore Filippone. Three storage formats for sparse matrices on GPGPUs. Technical Report DICII RR-15.6, Università di Roma Tor Vergata, 2015. `http://hdl.handle.net/2108/113393`.

[11] M. M. Baskaran and R. Bordawekar. Optimizing sparse matrix-vector multiplication on GPUs. Technical Report RC24704, IBM Research, 2009.

[12] S. Baxter. Modern GPU library, 2013. `http://nvlabs.github.io/moderngpu/`.

[13] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. Technical Report NVR-2008-004, NVIDIA Corp., 2008.

[14] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proc. of Int'l Conf. on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 18:1–18:11. ACM, 2009.

[15] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröoder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003.

[16] Luc Buatois, Guillaume Caumon, and Bruno Levy. Concurrent Number Cruncher: A GPU implementation of a general sparse linear solver. *Int. J. Parallel Emerg. Distrib. Syst.*, 24(3):205–223, 2009.

[17] Wei Cao, Lu Yao, Zongzhe Li, Yongxian Wang, and Zhenghua Wang. Implementing sparse matrix-vector multiplication using CUDA based on a hybrid sparse matrix format. In *Proc. of 2010 In'tl Conf. on Computer Application and System Modeling*, volume 11 of *ICCASM '10*, pages 161–165. IEEE, October 2010.

[18] V. Cardellini, A. Fanfarillo, and S. Filippone. Heterogeneous sparse matrix computations on hybrid GPU/CPU platforms. In *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, volume 25 of *Advances in Parallel Computing*, pages 203–212. IOS Press, 2014.

[19] Valeria Cardellini, Ssalvatore Filippone, and Damian Rouson. Design patterns for sparse-matrix computations on hybrid CPU/GPU platforms. *Sci. Program.*, 22(1):1–19, 2014.

[20] Ali Cevahir, Akira Nukada, and Satoshi Matsuoka. Fast conjugate gradients with multiple GPUs. In *Computational Science - ICCS 2009*, volume 5544 of *LNCS*, pages 893–903. Springer, 2009.

[21] J. Choi, J. Demmel, J. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers. LAPACK Working Note #95, University of Tennessee, 1995.

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

[23] P Colella. Defining software requirements for scientific computing, 2004. http://view.eecs.berkeley.edu/w/images/temp/6/6e/20061003235551!DARPAHPCS.ppt.

[24] CUSP : A C++ Templated Sparse Matrix Library, 2016. http://cusplibrary.github.io.

[25] M. Daga and J. L. Greathouse. Structural agnostic SpMV: Adapting CSR-Adaptive for irregular matrices. In *Proc. of IEEE 22nd Int'l Conf. on High Performance Computing*, HiPC '15, pages 64–74, 2015.

[26] Pasqua D'Ambra and Salvatore Filippone. A parallel generalized relaxation method for high-performance image segmentation on GPUs. *J. Comput. Appl. Math.*, 293:35–44, 2016.

[27] H.-V. Dang and B. Schmidt. CUDA-enabled sparse matrix-vector multiplication on GPUs using atomic operations. *Parallel Comput.*, 39(11):737–750, 2013.

[28] T. Davis. Wilkinson's sparse matrix definition. *NA Digest*, 07(12):379–401, March 2007.

[29] T. A. Davis. Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw.*, 30(2):196–199, 2004.

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

[31] M.M. Dehnavi, D.M. Fernandez, and D. Giannacopoulos. Finite-element sparse matrix vector multiplication on graphic processing units. *IEEE Trans. Magnetics*, 46(8):2982–2985, 2010.

[32] D. Elizabeth Dolan and J. Jorge Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, 2002.

[33] I.S. Duff, M. Marrone, G. Radicati, and C. Vittoli. Level 3 basic linear algebra subprograms for sparse matrices: a user level interface. *ACM Trans. Math. Softw.*, 23(3):379–401, 1997.

[34] A. Dziekonski, A. Lamecki, and M. Mrozowski. A memory efficient and fast sparse matrix vector product on a GPU. *Progress in Electromagnetics Research*, 116:49–63, 2011.

[35] EESI Working Group 4.3. Working group report on numerical libraries, solvers and algorithms. Technical report, European Exascale Software Initiative, 2011. http://www.eesi-project.eu/.

[36] Ahmed H. El Zein and Alistair P. Rendell. Generating optimal CUDA sparse matrix vector product implementations for evolving GPU hardware. *Concurr. Comput.: Pract. Exper.*, 24(1):3–13, 2012.

[37] 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 *Proc. of 17th Int'l Conf. on Parallel and Distributed Systems*, ICPADS '11, pages 165–172. IEEE Computer Society, 2011.

[38] X. Feng, H. Jin, R. Zheng, Z. Shao, and L. Zhu. A segment-based sparse matrix vector multiplication on CUDA. *Concurr. Comput.: Pract. Exper.*, 26(1):271–286, 2014.

[39] S. Filippone and A. Buttari. Object-oriented techniques for sparse matrix computations in Fortran 2003. *ACM Trans. Math. Softw.*, 38(4):23:1–23:20, 2012.

[40] Jeswin Godwin, Justin Holewinski, and P. Sadayappan. High-performance sparse matrix-vector multiplication on GPUs for structured grid computations. In *Proc. of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 47–56. ACM, 2012.

[41] J.L. Greathouse and M. Daga. Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format. In *Proc. of Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 769–780. IEEE, November 2014.

[42] A. Greenbaum. *Iterative Methods for Solving Linear Systems.* SIAM, Philadelphia, PA, 1997.

[43] Dominik Grewe and Anton Lokhmotov. Automatically generating and tuning GPU code for sparse matrix-vector multiplication from a high-level representation. In *Proc. of 4th Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 12:1–12:8. ACM, 2011.

[44] Dahai Guo and William Gropp. Adaptive thread distributions for SpMV on a GPU. In *Proc. of Extreme Scaling Workshop*, BW-XSEDE '12, pages 2:1–2:5, Champaign, IL, 2012. University of Illinois at Urbana-Champaign.

[45] P. Guo, L. Wang, and P. Chen. A performance modeling and optimization analysis tool for sparse matrix-vector multiplication on GPUs. *IEEE Trans. Parallel Distrib. Syst.*, 25(5):1112–1123, 2014.

[46] Ping Guo and Liqiang Wang. Auto-tuning CUDA parameters for sparse matrix-vector multiplication on GPUs. In *Proc. of 2010 Int'l. Conf. on Computational and Information Sciences*, ICCIS '10, pages 1154–1157. IEEE Computer Society, December 2010.

[47] Ping Guo and Liqiang Wang. Accurate cross-architecture performance modeling for sparse matrix-vector multiplication (SpMV) on GPUs. *Concurr. Comput.: Pract. Exper.*, 27(13):3281–3294, 2015.

[48] M. Heller and Tomás Oberhuber. Improved row-grouped CSR format for storing of sparse matrices on GPU. In *Proc. of Algoritmy 2012*, pages 282–290, 2012.

[49] M. R. Hugues and S. G. Petiton. Sparse matrix formats evaluation and optimization on a GPU. In *Proc. of 12th IEEE Int'l Conf. on High Performance Computing and Communications*, HPCC '10, pages 122–129. IEEE Computer Society, September 2010.

[50] W. W. Hwu. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 2011.

[51] Sivaramakrisha Bharadwaj Indarapu, Manoj Maramreddy, and Kishore Kothapalli. Architecture- and workload- aware heterogeneous algorithms for sparse matrix vector multiplication. In *Proc. of 19th IEEE Int'l Conf. on Parallel and Distributed Systems*, ICPADS '13, December 2013.

[52] V. Karakasis, G. Goumas, and N. Koziris. Perfomance models for blocked sparse matrix-vector multiplication kernels. In *Proc. of 38th Int'l Conf. on Parallel Processing*, ICPP '09, pages 356–364. IEEE Computer Society, September 2009.

[53] C. T. Kelley. *Iterative Methods for Linear and Nonlinear Equations*. SIAM, Philadelphia, 1995.

[54] D. R Kincaid, T. C. Oppe, and D. M. Young. ITPACKV 2D User's Guide, May 1989. `http://rene.ma.utexas.edu/CNA/ITPACK/manuals/userv2d/`.

[55] Z. Koza, M. Matyka, S. Szkoda, and L. Miroslaw. Compressed multirow storage format for sparse matrices on graphics processing units. *SIAM J. Sci. Comput.*, 36(2):C219–C239, 2014.

[56] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. Basermann, and A. R. Bishop. Sparse matrix-vector multiplication on GPGPU clusters: A new storage format and a scalable implementation. In *Proc. of 26th IEEE Int'l Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IPDPSW '12, pages 1696–1702, 2012.

[57] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units. *SIAM J. Sci. Comput.*, 36(5):C401–C423, 2014.

[58] Yuji Kubota and Daisuke Takahashi. Optimization of sparse matrix-vector multiplication by auto selecting storage schemes on GPU. In *Computational Science and Its Applications - ICCSA 2011*, volume 6783 of *LNCS*, pages 547–561. Springer, 2011.

[59] Verena Kuhlemann and Panayot S. Vassilievski. Improving the communication pattern in matrix-vector operations for large scale-free graphs by disaggregation. *SIAM J. Sci. Comput.*, 35(5):S465–S486, 2013.

[60] D. Langr and P. Tvrdik. Evaluation criteria for sparse matrix storage formats. *IEEE Trans. Parallel Distrib. Syst.*, 27(2):428–440, 2016.

[61] D. Langr, P. Tvrdik, and I. Simecek. Storing sparse matrices in files in the adaptive-blocking hierarchical storage format. In *Proc. of 2013 Federated Conference on Computer Science and Information Systems*, FedCSIS '13, pages 479–486. IEEE, 2013.

[62] Randall J. LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations*. SIAM, Philadelphia, PA, 2007.

[63] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. SMAT: An input adaptive auto-tuner for sparse matrix-vector multiplication. In *Proc. of 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 117–126, 2013.

[64] K. Li, W. Yang, and K. Li. Performance analysis and optimization for SpMV on GPU using probabilistic modeling. *IEEE Trans. Parallel Distrib. Syst.*, 26(1):196–205, 2015.

[65] R. Li and Y. Saad. GPU-accelerated preconditioned iterative linear solvers. *J. Supercomput.*, 63(2):443–466, 2013.

[66] HUI Liu, SONG Yu, ZHANGXIN Chen, BEN Hsieh, and LEI Shao. Sparse matrix-vector multiplication on NVIDIA GPU. *Int. J. Numerical Analysis and Modeling, Series B*, 3(2):185–191, 2012.

[67] W. Liu and B. Vinter. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proc. of 29th Int'l ACM Conf. on Supercomputing*, ICS '15, 2015.

[68] Yongchao Liu and B. Schmidt. LightSpMV: Faster CSR-based sparse matrix-vector multiplication on CUDA-enabled GPUs. In *Proc. of 26th Int'l Conf. on Application-specific Systems, Architectures and Processors*, ASAP '15, pages 82–89, July 2015.

[69] David Luebke, Mark Harris, Naga Govindaraju, Aaron Lefohn, Mike Houston, John Owens, Mark Segal, Matthew Papakipos, and Ian Buck. GPGPU: general-purpose computation on graphics hardware. In *Proc. of 2006 ACM/IEEE Conf. on Supercomputing*, SC '06, 2006.

[70] M. Maggioni and T. Berger-Wolf. AdELL: An adaptive warp-balancing ELL format for efficient sparse matrix-vector multiplication on GPUs. In *Proc. of 42nd Int'l Conf. on Parallel Processing*, ICPP '13, pages 11–20. IEEE Computer Society, October 2013.

[71] M. Maggioni and T. Berger-Wolf. CoAdELL: Adaptivity and compression for improving sparse matrix-vector multiplication on GPUs. In *Proc. of 2014 IEEE Int'l Parallel Distributed Processing Symposium Workshops*, IPDPSW '14, pages 933–940, May 2014.

[72] M. Maggioni, T. Berger-Wolf, and Jie Liang. GPU-based steady-state solution of the chemical master equation. In *Proc. of IEEE 27th Int'l Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IPDPSW '13, pages 579–588, May 2013.

[73] Marco Maggioni and Tanya Berger-Wolf. Optimization techniques for sparse matrix-vector multiplication on GPUs. *J. Parallel Distrib. Comput.*, 93–94:66–86, 2016.

[74] Matrix algebra on GPU and multicore architectures, September 2016. `http://icl.cs.utk.edu/magma/`.

[75] A. Maringanti, V. Athavale, and S. B. Patkar. Acceleration of conjugate gradient method for circuit simulation using CUDA. In *Proc. of 2009 Int'l Conf. on High Performance Computing*, HiPC '09, pages 438–444. IEEE, December 2009.

[76] K.K. Matam and K. Kothapalli. Accelerating sparse matrix vector multiplication in iterative methods using GPU. In *Proc. of 40th Int'l Conf. on Parallel Processing*, ICPP '11, pages 612–621. IEEE Computer Society, September 2011.

[77] A. Monakov and A. Avetisyan. Implementing blocked sparse matrix-vector multiplication on NVIDIA GPUs. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 5657 of *LNCS*, pages 289–297. Springer-Verlag, 2009.

[78] A. Monakov, A. Lokhmotov, and A. Avetisyan. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In *High Performance Embedded Architectures and Compilers*, volume 5952 of *LNCS*, pages 111–125. Springer-Verlag, 2010.

[79] 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*, volume 7975 of *LNCS*, pages 211–223. Springer, 2013.

[80] B. Neelima, S. R. Prakash, and Ram Mohana Reddy. New sparse matrix storage format to improve the performance of total SpMV time. *Scalable Comput.: Pract. Exper.*, 13(2):159–171, 2012.

[81] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6:40–53, March 2008.

[82] NVIDIA Corp. CUDA cuSPARSE library, 2015. `http://developer.nvidia.com/cusparse`.

[83] Tomás Oberhuber, Atsushi Suzuki, and Jan Vacata. New row-grouped CSR format for storing the sparse matrices on GPU with implementation in CUDA. *Acta Technica*, 56:447–466, 2011.

[84] S. V. Patankar. *Numerical Heat Transfer and Fluid Flow*. Series in Computational Methods in Mechanics and Thermal Sciences. Hemisphere Publishing Corp., NY, first edition, 1980.

[85] J. C. Pichel, F. F. Rivera, M. Fernández, and A. Rodríguez. Optimization of sparse matrix-vector multiplication using reordering techniques on GPUs. *Microprocess. Microsyst.*, 36(2):65–77, 2012.

[86] A. Quarteroni and A. Valli. *Numerical Approximation of Partial Differential Equations*. Springer-Verlag, Berlin, 1994.

[87] I. Reguly and M. Giles. Efficient sparse matrix-vector multiplication on cache-based GPUs. In *Proc. of Innovative Parallel Computing*, InPar '12, pages 1–12. IEEE, May 2012.

[88] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2nd edition, 2003.

[89] J. Sanders and E. Kandrot. *CUDA by example: An introduction to general-purpose GPU programming*. Addison-Wesley, Boston, MA, USA, first edition, 2010.

[90] Naser Sedaghati, Te Mu, Louis-Noel Pouchet, Srinivasan Parthasarathy, and P. Sadayappan. Automatic selection of sparse matrix representation on GPUs. In *Proc. of 29th ACM Int'l Conf. on Supercomputing*, ICS '15, pages 99–108, 2015.

[91] M. Shah and V. Patel. An efficient sparse matrix multiplication for skewed matrix on GPU. In *Proc. of 14th IEEE Int'l Conf. on High Performance Comput. and Comm.*, pages 1301–1306, June 2012.

[92] Bor-Yiing Su and Kurt Keutzer. clSpMV: a cross-platform OpenCL SpMV framework on GPUs. In *Proc. of 26th ACM Int'l Conf. on Supercomputing*, ICS '12, pages 353–364, 2012.

[93] Xiangzheng Sun, Yunquan Zhang, Ting Wang, Xianyi Zhang, Liang Yuan, and Li Rao. Optimizing SpMV for diagonal sparse matrices on GPU. In *Proc. of 40th Int'l Conf. on Parallel Processing*, ICPP '11, pages 492–501. IEEE Computer Society, September 2011.

[94] W. Tang, W. Tan, R. Goh, S. Turner, and W. Wong. A family of bit-representation-optimized formats for fast sparse matrix-vector multiplication on the GPU. *IEEE Trans. Parallel Distrib. Syst.*, 29(9):2373–2385, 2015.

[95] Wai Teng Tang, Wen Jun Tan, Rajarshi Ray, Yi Wen Wong, Weiguang Chen, Shyh-hao Kuo, Rick Siow Mong Goh, Stephen John Turner, and Weng-Fai Wong. Accelerating sparse matrix-vector multiplication on GPUs using bit-representation-optimized schemes. In *Proc. of Int'l Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 26:1–26:12. ACM, 2013.

[96] F. Vázquez, J. J. Fernández, and E. M. Garzón. A new approach for sparse matrix vector product on NVIDIA GPUs. *Concurr. Comput.: Pract. Exper.*, 23(8):815–826, 2011.

[97] 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(8):408–420, 2012.

[98] Mickeal Verschoor and Andrei C. Jalba. Analysis and performance estimation of the conjugate gradient method on multiple GPUs. *Parallel Comput.*, 38(10-11):552–575, 2012.

[99] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proc. of 2008 ACM/IEEE Conf. on Supercomputing*, SC '08, pages 31:1–31:11, 2008.

[100] Zhuowei Wang, Xianbin Xu, Wuqing Zhao, Yuping Zhang, and Shuibing He. Optimizing sparse matrix-vector multiplication on CUDA. In *Proc. of 2nd Int'l Conf. on Education Technology and Computer*, volume 4 of *ICETC '10*, pages 109–113. IEEE, June 2010.

[101] Daniel Weber, Jan Bender, Markus Schnoes, André Stork, and Dieter Fellner. Efficient GPU data structures and methods to solve sparse linear systems in dynamics applications. *Computer Graphics Forum*, 32(1):16–26, 2013.

[102] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009.

[103] Sam Williams, Nathan Bell, Jee Whan Choi, Michael Garland, Leonid Oliker, and Richard Vuduc. Sparse matrix-vector multiplication on multicore and accelerators. In *Scientific Computing on Multicore and Accelerators*, pages 83–109. CRC Press, Boca Raton, FL, December 2010.

[104] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Comput.*, 35(3):178–194, 2009.

[105] J. Wong, E. Kuhl, and E. Darve. A new sparse matrix vector multiplication graphics processing unit algorithm designed for finite element

problems. *Int'l J. Numerical Methods in Engineering*, 102(12):1784–1814, 2015.

[106] Weizhi Xu, Hao Zhang, Shuai Jiao, Da Wang, Fenglong Song, and Zhiyong Liu. Optimizing sparse matrix vector multiplication using cache blocking method on Fermi GPU. In *Proc. of 13th ACIS Int'l Conf. on Software Engineering, Artificial Intelligence, Networking and Parallel Distributed Computing*, SNPD '12, pages 231–235. IEEE Computer Society, August 2012.

[107] S. Yan, C. Li, Y. Zhang, and H. Zhou. yaSpMV: yet another SpMV framework on GPUs. In *Proc. of 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 107–118, February 2014.

[108] Mei Yang, Cheng Sun, Zhimin Li, and Dayong Cao. An improved sparse matrix-vector multiplication kernel for solving modified equation in large scale power flow calculation on CUDA. In *Proc. of 7th Int'l Power Electronics and Motion Control Conf.*, volume 3 of *IPEMC '12*, pages 2028–2031. IEEE, June 2012.

[109] W. Yang, K. Li, Z. Mo, and K. Li. Performance optimization using partitioned SpMV on GPUs and multicore CPUs. *IEEE Trans. Comput.*, 64(9):2623–2636, 2015.

[110] Wangdong Yang, Kenli Li, Yan Liu, Lin Shi, and Lanjun Wan. Optimization of quasi-diagonal matrix-vector multiplication on GPU. *Int. J. High Perform. Comput. Appl.*, 28(2):183–195, 2014.

[111] Xintian Yang, Srinivasan Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on GPUs: Implications for graph mining. *Proc. VLDB Endow.*, 4(4):231–242, 2011.

[112] H. Yoshizawa and D. Takahashi. Automatic tuning of sparse matrix-vector multiplication for CRS format on GPUs. In *Proc. of IEEE 15th Int'l Conf. on Computational Science and Engineering*, CSE '12, pages 130–136, December 2012.

[113] Liang Yuan, Yunquan Zhang, Xiangzheng Sun, and Ting Wang. Optimizing sparse matrix vector multiplication using diagonal storage matrix format. In *Proc. of 12th IEEE Int'l Conf. on High Performance Computing and Communications*, HPCC '10, pages 585–590, September 2010.

[114] Cong Zheng, Shuo Gu, Tong-Xiang Gu, Bing Yang, and Xing-Ping Liu. BiELL: A bisection ELLPACK based storage format for optimizing SpMV on GPUs. *J. Parallel Distrib. Comput.*, 74(7):2639–2647, 2014.

Sparse matrix-vector multiplication on GPGPUs

Filippone, Salvatore

2017-03-01

Filippone S, Cardellini V, Barbieri D, Fanfarillo A. (2017) Sparse matrix-vector multiplication on GPGPUs. ACM Transactions on Mathematical Software, Volume 43, Issue 4, March 2017, Article 30

http://dx.doi.org/10.1145/3017994