CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2016)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3874

SPECIAL ISSUE PAPER

Performance optimization of Sparse Matrix-Vector Multiplication for multi-component PDE-based applications using GPUs

Ahmad Abdelfattah[1,*,†], Hatem Ltaief[2], David Keyes[2] and Jack Dongarra[1,3,4]

[1]*Innovative Computing Laboratory, University of Tennessee, Knoxville, USA*
[2]*Extreme Computing Research Center, King Abdullah University of Science and Technology, Kingdom of Saudia Arabia*
[3]*Oak Ridge National Laboratory, USA*
[4]*University of Manchester, UK*

SUMMARY

Simulations of many multi-component PDE-based applications, such as petroleum reservoirs or reacting flows, are dominated by the solution, on each time step and within each Newton step, of large sparse linear systems. The standard solver is a preconditioned Krylov method. Along with application of the preconditioner, memory-bound Sparse Matrix-Vector Multiplication (SpMV) is the most time-consuming operation in such solvers. Multi-species models produce Jacobians with a dense block structure, where the block size can be as large as a few dozen. Failing to exploit this dense block structure vastly underutilizes hardware capable of delivering high performance on dense BLAS operations. This paper presents a GPU-accelerated SpMV kernel for block-sparse matrices. Dense matrix-vector multiplications within the sparse-block structure leverage optimization techniques from the KBLAS library, a high performance library for dense BLAS kernels. The design ideas of KBLAS can be applied to block-sparse matrices. Furthermore, a technique is proposed to balance the workload among thread blocks when there are large variations in the lengths of nonzero rows. Multi-GPU performance is highlighted. The proposed SpMV kernel outperforms existing state-of-the-art implementations using matrices with real structures from different applications. Copyright © 2016 John Wiley & Sons, Ltd.

Received 13 December 2015; Revised 22 April 2016; Accepted 22 April 2016

KEY WORDS: sparse matrix-vector multiplication; GPU optimizations; block sparse matrices

1. INTRODUCTION

Many simulations motivating the development of high performance linear algebra libraries on emerging architectures have a matrix structure of sparsely distributed blocks that are densely populated, inherited from a multi-component PDE structure. For instance, reservoir modeling, the simulation of flow through porous media, is a nonlinear, multi-species, multi-phase problem, which tracks many components per discretization cell. Efficient extraction of the hydrocarbons remaining after the conventional phase, which is capable of recovering typically about 35% of the reservoir, is one of greatest environmental challenges facing mankind. Production is often accompanied by contaminated water (or other fluids) injected to displace the hydrocarbons. Furthermore, the enhanced recovery phase, ideally up to 70% of the total, has a poorer return on energy investment. Optimizing reservoir development while reducing uncertainty requires simulating many forward scenarios. The bottleneck in these and many other applications is often the solution of large, sparse spatially structured or unstructured linear systems within each Newton step on each implicit step of the time integration. Because the cost of solving these systems grows superlinearly in the number of

*Correspondence to: Ahmad Abdelfattah, Innovative Computing Laboratory, University of Tennessee, Knoxville, USA.
†E-mail: ahmad@icl.utk.edu

discrete cells due to elliptic ill-conditioning and polynomially in the block size at each cell, while other computational costs are essentially linear, the cost of the linear solves grows to an arbitrarily high percentage of the overall computational effort as the grid is refined and more components are resolved; 80% or more is not unusual in practice in the reservoir modeling industry. The Sparse Matrix-Vector multiplication (SpMV) is the innermost computational kernel in typical reservoir models; therefore, overall computational efficiency directly reflects the performance of this kernel. For contemporary petroleum reservoir simulations, the block size typically ranges from three to a few dozen, and still larger blocks are relevant within adaptively model-refined regions of the domain. Because of the low native arithmetic intensity of the SpMV kernel, it is essential to fully exploit the matrix structure to hide data motion with useful computations.

We leverage optimization techniques from the KBLAS library [1] in the context of the SpMV kernel for block-sparse matrices. We show how design ideas such as register blocking and double buffering can be used not only for relatively large dense matrices, but also at the smaller scale represented by sparse matrices arising from PDE applications with multiple components. While these optimizations are important for high performance dense kernel executions, they are even more critical when dealing with sparse linear algebra operations, due to irregular memory accesses and low compute-intensity kernels. The new SpMV kernel outperforms existing state-of-the-art implementations on GPUs. A multi-GPU SpMV interface allows simulation of larger problem sizes, while increasing the level of concurrency. The paper extends the effort previously introduced in [2] in the following respects:

(1) Increased level of detail for the kernel design, with illustrative figures for every proposed SpMV kernel.
(2) Instead of matrices with synthetic structure, the performance reported in this paper is based on matrices with structures arising from real applications. We use a number of general sparse matrices from the University of Florida Sparse Matrix Collection [3], and promote every nonzero entry to a square block of a given size. This gives insight on the performance of the proposed kernel on more realistic structures.
(3) We propose a modification to the SpMV kernel that achieves better performance than the kernel proposed in [2], when there is large variation in the number of nonzero in the matrix rows. Based on the histogram of the nonzero row lengths of these matrices, we show that the performance of the original kernel [2] may be unacceptable and propose a modification that enables a much better performance. The multi-GPU scaling of the new version is also highlighted.

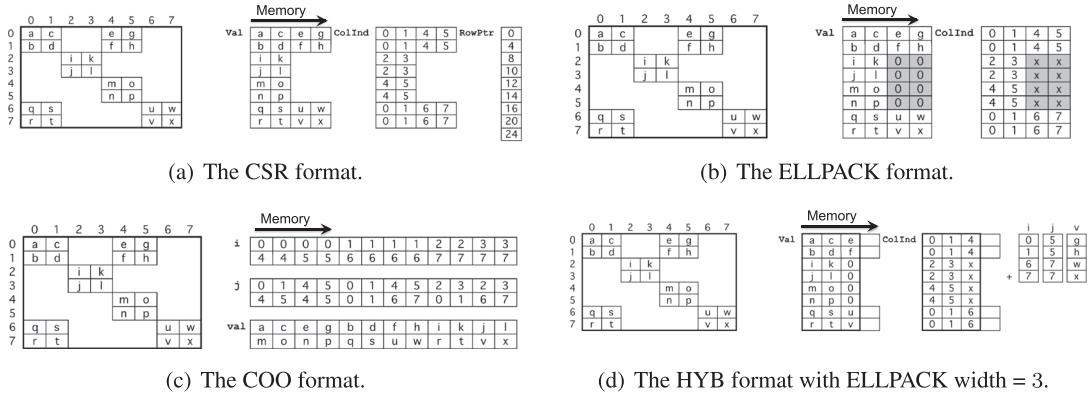
The rest of the paper is organized as follows. Section 2 provides a literature review for SpMV using both CPUs and accelerators. Section 3 discusses multi-component PDE-based applications, which are conventionally ordered into block-sparse matrices. Section 4 presents a uniform design strategy inherited from KBLAS [1], and shows how it can be applied to an SpMV kernel. In Section 5, we describe the design of the proposed SpMV kernels. Section 6 presents performance results. We conclude in Section 7.

2. RELATED WORK

Because of its importance and wide use, the literature is rich in contributions for GPU-accelerated SpMV. Bell and Garland [4] proposed SpMV implementations for several formats, including Compressed Sparse Row (CSR), ELLPACK [5], and the Coordinate (COO) format. They also proposed HYB, which is a hybrid format that combines both the ELLPACK format and the COO format, in an effort to reduce the padding overhead of ELLPACK. Most of these implementations (probably more optimized) are available in the cuSPARSE library [6], as are the baselines against which most researchers compare their techniques. The four formats (CSR, ELLPACK, COO, and HYB) are shown in Figure 1.

Perhaps the ELLPACK format [5] is the most convenient format for GPUs, because a sparse matrix A is stored as a dense matrix (in column major format) with dimensions $m \times \text{nnz}_{max}$, where m

DOI: 10.1002/cpe



(a) The CSR format.

(b) The ELLPACK format.

(c) The COO format.

(d) The HYB format with ELLPACK width = 3.

Figure 1. Representation of a block-sparse matrix by different formats.

is the number of rows of A and nnz_{max} is the maximum number of non-zeros found in the rows of A (Figure 1(b)). Another dense matrix is required to store the integer column indices of the non-zeros. The regularity of ELLPACK format is obtained at the cost of introducing zero padding overhead, when there is a variation in the row lengths of A. The overhead is reflected in extra memory reads plus extra computation.

Many researchers have proposed remedies to the ELLPACK overhead. Monakov *et al.* [7] proposed a sliced version of the ELLPACK format, where each slice is stored in a separate ELLPACK format. The slice size can be fixed or variable, and the zero padding can be even reduced by reordering the rows according to their lengths. Vázquez *et al.* [8] proposed the ELLPACK-R format that adds auxiliary information to avoid the extra computation. They introduced an extra integer array that keeps the non-zero length of each row in order to skip computing zeros. Choi *et al.* [9] proposed a parameterized blocked version of the ELLPACK format that proves to be competitive for block-sparse matrices, although it is restricted to certain block sizes, and targets mainly Fermi generation GPUs. They also proposed an autotuning framework that uses a performance model and matrix-dependent parameters to identify the best storage parameters for the proposed format. Kreutzer *et al.* [10] proposed the packed Jagged Diagonal Storage (pJDS), which is very similar to the sliced ELLPACK format proposed by Monakov *et al.* [7]. They show up to 1.3× speedup against the ELLPACK-R SpMV with up to 70% saving in memory requirements. They also generalized the sliced ELLPACK format to the SELL-C-σ format [11], in an effort to provide a unified sparse storage format across different architectures. The authors show that this format is suitable for multicore processors as well as GPUs, including the Intel Xeon Phi. The proposed implementation does not show a significant loss of performance when compared against hardware-specific-formats. The SELL-C-σ has been improved and optimized for GPUs by Anzt *et al.* [12], by introducing some zero padding to satisfy the memory constraints of the GPU architecture, hence called the SELL-P format.

Ashari *et al.* [13] proposed an adaptive algorithm for SpMV using the CSR format (called ACSR), where additional metadata are used with the standard CSR format that help achieve better GPU utilization. ACSR groups rows with similar lengths into *bins* so that a bin-specific kernel is launched. Kernels that process different bins are launched concurrently using *dynamic parallelism*, a feature available on NVIDIA GPUs starting compute capability 3.5 [14]. Although the performance of ACSR is better than other similar implementations by a small margin, its preprocessing step is much cheaper than other formats. ACSR is mainly proposed for adaptive graph applications, where the structure of the graph adjacency matrix changes frequently, thus making the preprocessing step a serious bottleneck. A similar approach was proposed by Greathouse and Daga [15] for AMD GPUs.

We are mainly interested in the Blocked Sparse Row (BSR) format, which is the blocked version of the CSR format. It was first introduced for CPU architectures by Im *et al.* [16, 17]. The BSR format targets sparse matrices that are naturally blocked, as shown in Figure 2. It uses one integer per block to store its column index, as well as an integer to denote the start of every block row. In cases specific to structured grid problems, Godwin *et al.* [18] proposed a format called Column

DOI: 10.1002/cpe

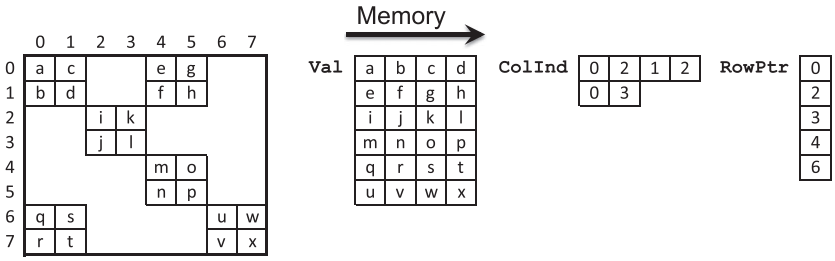


Figure 2. The BSR format.

Diagonal Storage (CDS), which assigns only one integer for a group of blocks located at the same diagonal/off-diagonal. The work by Choi *et al.* [9] suggested GPU specific optimizations for the BSR format that were not enough to outperform the cuSPARSE HYB kernel [4]. They concluded that the BSR will be dominated by a reduction step that is affected by the number of blocks per block row. This paper revisits the BSR formats and proposes some optimization techniques for a wide range of block sizes.

While most researchers propose formats that inherit properties of the CSR or ELLPACK formats (or both), there are some efforts that investigated enhancing the SpMV operation using the COO format. For example Dang and Schmidt [19] investigated a sliced version of the COO format (called SCOO), where they reported more than 50% speedup over the respective COO implementation by cuSPARSE. Yan *et al.* [20] proposed a Blocked Compressed Common Coordinate (BCCOO) format, which is a variant of the COO format that uses bit flags instead of full integers to store row information. It also processes the matrix into vertical slices to improve the cache hit rate for the vector access.

Among all the aforementioned sparse formats, the cuSPARSE HYB format along with the SELL-P format usually achieve the best performance on the GPU across several matrices, as long as the matrix structure does not change during the simulation.

3. PDE-BASED APPLICATIONS WITH MULTI-COMPONENTS

Numerous applications result in sparse matrices of dense blocks, where the first nontrivial block size is two (e.g., streamfunction and vorticity in fluid dynamics) and the block size ranges up to hundreds in realistic contemporary applications that drive high performance computing (e.g., detailed kinetics models of hydrocarbon-air combustion). In the applications expressed as PDEs that motivate this work, the number of components is related to the number of fields defined over the domain. The blocks are square because each field (e.g., density, momentum, internal energy, concentration of a given species in a given phase in a given charge state) has its own conservation equation.

If the conservation equations were decoupled, all blocks would be diagonal and the data structures designed for this paper would not be relevant for high performance. However, most systems of conservation laws (Equation (1)) couple the fields defined at each point through possibly several types of physical interdependencies.

$$\frac{\partial(\rho\phi_k)}{\partial t} + \nabla\cdot(\rho\mathbf{v}\phi_k) - \nabla\cdot(\mu_k\nabla\phi_k) = F_k(\phi_1,\phi_2,\cdots,\phi_K), \quad k = 1,2,\cdots,K. \tag{1}$$

In the typical convection-diffusion-reaction system shown, the convection terms couple the momenta to all convected components. The momenta are products of density (ρ) and velocities ($\mathbf{v}$), and the density is a function of the mass fractions and thermodynamic state of all of the species ($\phi_1,\phi_2,\cdots,\phi_K$) in the system. The gradient operator acting on the density couples degrees of freedom across the grid points in the stencil, so the typical off-diagonal component of the off-diagonal blocks is nonzero. The diffusion terms couple the degrees of freedom to each other because the diffusion coefficients (μ_k) are also complex functions of the mass fractions and thermodynamic state

DOI: 10.1002/cpe

at each point. Again, the gradient operator couples the degrees of freedom across the grid points in the stencil, so that the off-diagonal blocks are best regarded as fully dense. The structure of the reaction terms for the creation and consumption of each component (F_k) may lead to some exploitable sparsity within the diagonal blocks because not all components react with all others. However, the diagonal blocks are often factored as part of a block preconditioner to pre-scale the system and the blocks are best regarded as full in this case.

Equation (1) is a simplified schema of systems described by first principles in, for example, [21] for porous media applications or [22] for reacting flows. In turn, such systems may be regarded as embedded in multiphysics applications for which computational modelers increasingly prefer fully implicit solvers [23] for reasons of numerical efficiency, stability, and/or robustness. Past generations of modelers lacking powerful high performance solvers have tended to employ operator splitting to solve such systems in a series of steps that leave behind first-order temporal splitting errors and potentially destabilizing mechanisms. Splitting also weakens temporal locality and arithmetic intensity. Contemporary high performance solver software allows such users to more fully exploit the inexpensive flops of a GPU and reduce expensive memory thrashing.

Often, a natural synergism is exploited in the overall Newton-Krylov solution framework, wherein the matrix-vector products with the Jacobian are approximated by directional differences of the conservation law residual function of which Newton attempts to find the root [24], the so-called Jacobian-free Newton Krylov method. However, when the residual functions are highly complex, especially when they require table lookups for constitutive properties, and when the Jacobian will be sufficiently reused, it is preferable to remove such function calls from the inner Krylov loop by computing with explicitly stored Jacobian elements, whether the elements are themselves precomputed analytically or by residual differences. This is the context of the current contribution.

4. A UNIFORM DESIGN STRATEGY

This section introduces the main design principles of KBLAS [1] as abstract ideas, which are then projected into the detailed design of the SpMV kernels discussed in Section 5.

4.1. Hierarchical register blocking

Matrix Block and Block Size.[‡] An input matrix is always subdivided into square or rectangular blocks. In this work, a *matrix block* is always square, with a size of $bs \times bs$. Both terms *matrix block* and *block* are equivalent and refer to the basic unit used in matrix subdivision. KBLAS uses bs as a tuning parameter. However, in block-sparse matrices, the input matrix comes naturally blocked, and the value of bs is predetermined by the BSR format. It might be too small to provide sufficient parallelism, or too large to be processed as a single block. This is why we introduce another terminology for the amount of work that can be processed at one time using the same group of CUDA threads.

Working Set. A *working set* denotes the minimum amount of work assigned to a thread block (TB) at a time. It is generally different from a *matrix block*. A *working set* has the dimensions $nb \times w$, where nb is the height. Both nb and w are controllable through tuning parameters. There are three cases that relate the size of the *working set* to the size of the *matrix block*.

(1) *Case 1*: A *working set* is equivalent to one *matrix block*. Because we focus on square blocks, this leads to $nb = w = bs$.
(2) *Case 2*: A *working set* spans multiple adjacent *matrix blocks* in the same block row. This also leads to $nb = bs$, but w is multiples of bs.
(3) *Case 3*: A *working set* is part of a *matrix block*. This case is used when the *matrix block* is so large that it can be subdivided into multiple *working sets*.

[‡]In this paper, we use block size and block dimension interchangeably.

DOI: 10.1002/cpe

KBLAS uses cases 1 and 3 for dense matrices, where bs is not bound to the number of unknowns. In fact, bs and nb are used in a recursive blocking technique for dense matrices, and are often set to relatively large values to saturate TBs with computation. In block sparse matrices, however, the value of bs is predetermined by the number of components, and only nb and w can be controlled according to any of the cases mentioned earlier.

In most cases, a TB processes one *working set* at a time. However, in some cases, a TB is allowed to process multiple *working sets* concurrently. A *working set* should fit into a fast memory level, such as shared memory or registers. Since we focus on the SpMV kernel, which lacks data reuse of the matrix, there is no need to incorporate shared memory in storing *working sets*. Our strategy proposes that a *working set* should be stored in the register file of a streaming multiprocessor (SM). This ensures the fastest data access when computation is carried out. In addition, excluding shared memory from storing *working sets* reduces chances of synchronization points, and eliminates unnecessary shared load/store instructions.

Thread Arrays and Thread Groups. A *working set* is processed by a *thread array*. In general, a TB consists of one or more *thread arrays*. A *thread array* is always structured as a 2D thread configuration whose height must be equal to the height of the *working set*. A *thread array*, therefore, has $nb \times N_{tg}$ threads, where N_{tg} is called the number of *thread groups*. N_{tg} is a tuning parameter.

As shown in Figure 3, a *working set* is subdivided into vertical slices. Each vertical slice is assigned to one *thread group*. Because each *thread group* has nb threads, it is recommended to choose a value of nb that is equivalent to one or more warps. However, the strategy supports any value for nb. Therefore, we use the term *thread group* instead of the term *warp*, because the former is more generic.

If the TB is designed to process one *working set* at a time, then the TB consist of one *thread array*. If it is designed to process multiple *working sets* concurrently, then the TB consists of multiple independent *thread arrays*, each of which consists of $nb \times N_{tg}$ threads. The number of *thread arrays* inside a TB is denoted as N_{ta}. It is exposed as a tuning parameter in some cases when bs is very small. Otherwise, the value of N_{ta} is 1.

Elements per Thread (*ept*). Going a level further, within a vertical slice of the *working set*, each thread is responsible of a single row of the vertical slice. Eventually, each thread stores, in registers, a small segment of a single row of the matrix. The number of registers required per thread to store such segment is called *elements per thread (ept)*. In fact, *ept* is a crucial design parameter that affects the kernel performance. Figure 3 shows the hierarchical register blocking of a 16×16 working set among 4 thread groups. The value of *ept* is 4. Apparently, the width of the *working set* w is equal to $N_{tg} \times ept$.

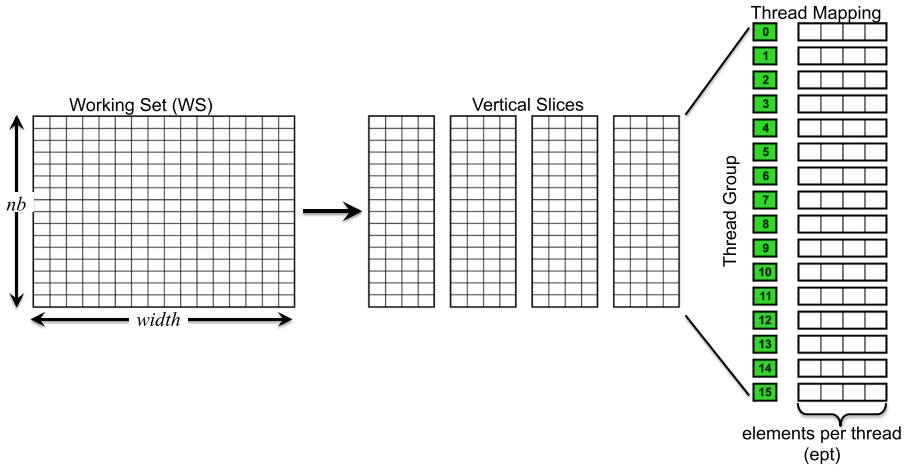


Figure 3. Hierarchical register blocking.

DOI: 10.1002/cpe

4.2. Double buffering

The proposed design strategy incorporates a double buffering scheme in order to hide the computation with data prefetching. This means that each TB requires a storage large enough to fit at least two *working sets*. Considering Figure 3, each thread needs two independent buffers, each one is *ept* in length ($ept = 4$ in the figure). It is necessary to ensure that the register pressure does not result into register spilling into the DRAM. This can be achieved through careful choices of the size of the *working set*, as well as *ept*. All computation on a working set is done on the register level, except for a final reduction step that is performed on shared memory among thread groups.

4.3. Instruction level parallelism (ILP)

In order to reduce chances of stalling thread execution, each *thread group* should execute instructions that are as independent from each other as possible. According to the register blocking technique shown in Figure 3, ILP can be achieved by choosing $ept > 1$, so that a compute step is not close to its respective load step in the instruction stream of the *thread group*, leading to less instruction dependency. In most cases, *ept* is larger than 1.

4.4. Warp latency hiding

The CUDA runtime can switch execution to other warps on the same SM, if the current warp gets stalled. It is then possible to reduce the latency of a warp stall by increasing the number of warps within a TB. Tuning parameters such as nb, N_{tg}, and N_{ta} are usually set to ensure multiple warps per TB.

4.5. Computation-driven TB mapping

TBs are programmed to process *working sets* whose partial results can be accumulated to each other, which is the case for working sets that belong to the same block row. This enables accumulation to happen at the register level as TBs move from one *working set* to another. It also minimizes the role of shared memory to a final reduction step among *thread groups* before writing the final result to the global DRAM.

4.6. Collaboration among TBs

KBLAS uses multiple TBs per block row or block column in order to achieve better occupancy for relatively small dense matrices [1]. The original SpMV design proposed by the authors [2] uses one TB per block row. While our results show good performance results for most matrices, we study some cases where the performance drops significantly due to huge variations in the row lengths. In Section 5, we extend the original SpMV design to allow multiple TBs to work on the same block row, which leads to more balanced workload among TBs.

5. IMPLEMENTATION DETAILS

This section discusses how the design concepts of KBLAS [1], which are described in Section 4, can be applied to the proposed SpMV kernel. The routine name BSRMV is used to denote an SpMV operation on block-sparse matrices using the BSR format.

5.1. Dividing block size range

From a user's point of view, one API is provided to perform the operation. However, it is not possible to have one CUDA kernel that can efficiently handle all block sizes. We introduce three different kernels to divide the spectrum of the block size. While the three kernels inherit the design strategy described in Section 4, they differ in the way they restrict some properties of the uniform strategy while relaxing others, according to the targeted block size. Based on experiments performed on a Kepler GPU, the proposed kernels are assigned the following ranges of the block size bs:

DOI: 10.1002/cpe

- *Kernel 1 (K1)*: can process small values of bs, strictly 2 through 5.
- *Kernel 2 (K2)*: can process medium *blocks* starting from 5 up to 45.
- *Kernel 3 (K3)*: is assigned for large *blocks* starting from 45 and beyond.

We point out that both *K2* or *K3* can be used for block sizes 5×5 and up. However, *K2* reaches a hardware contraint (in terms of number of threads and register pressure) if bs is too large. This is where *K3* should take over, as it is designed for large block sizes. In any kernel, all accesses to the input vector x are directed to the read-only data cache on the GPU. The memory references to x are often a limiting factors to performance. It is a common practice, found in many previous contributions, to direct accesses to x through a separate cache, in order to increase the chances of data reuse.

5.2. *Small blocks (*K1*)*

Kernel *K1* builds upon the fact that a single GPU warp can read one or more *blocks* in one memory transaction. Given that NVIDIA GPUs use a fixed warp size of 32, this kernel cannot be applied to *blocks* larger than 5×5. The grid design of *K1* consists of 1D array of TBs. Each TB is assigned $Nbrows_{tb}$ consecutive block rows, leading to a grid configuration $(B_x, 1)$, where $B_x = \left\lceil \frac{\text{\#block rows of the matrix}}{Nbrows_{tb}} \right\rceil$ As we point out later on, $Nbrows_{tb}$ is a function of two tuning parameters.

Kernel *K1* uses a *working set* that spans multiple adjacent *blocks* in the same block row. This implies that $nb = bs$, and w is multiples of bs. Each TB consists of N_{ta} independent *thread arrays*, where N_{ta} is greater than 1. Each *thread array* is strictly a warp that is truncated to be fully divisible by bs^2. The truncated warp is restructured to a $nb \times N_{tg}$ configuration, where $N_{tg} = \left\lfloor \frac{32}{bs^2} \right\rfloor \times bs$. The value of ept is strictly 1. Each truncated warp processes $Nbrows_{ta}$ consecutive block rows, where $Nbrows_{ta}$ is a tuning parameter. This implies that $Nbrows_{tb} = Nbrows_{ta} \times N_{ta}$. A TB is launched with a (T_x, T_y) configuration, where $T_x = 32$, and $T_y = N_{ta}$. Figure 4 shows the hierarchy of the overall kernel design. Truncated warps within the same TB are completely independent of each other. Therefore, the design of *K1* does not require any synchronization points, even if reduction through shared memory is needed. As an example, consider the case when $bs = 3$, as shown in Figure 5. Only 27 threads remain active, and the truncated warp is able to read three *blocks* at one memory transaction.

The double buffering technique is incorporated, although its effectiveness is dependent on the length of the block row. Let $nnzb_{warp} = \left\lfloor \frac{32}{bs^2} \right\rfloor$ denote the number of *blocks* that a warp can read at one memory request. If a block row contains more than $nnzb_{warp}$ *blocks*, then there is an opportunity to prefetch more data while the current $nnzb_{warp}$ *blocks* are being processed. Latency hiding among warps can be controlled through tuning the value of N_{ta} and $Nbrows_{ta}$. Both parameters are independent. There are no restrictions on these values except hardware restrictions (e.g., maximum number of threads per TB).

5.3. *Medium blocks (*K2*)*

Kernel *K2* has the same high level design of *K1* shown in Figure 4, but the low level details differ. A *working set* is equivalent to one *block*. This implies $nb = w = bs$. A TB consists of N_{ta} *thread arrays*. In contrast to *K1*, *K2* does not restrict a *thread array* to a warp. A *thread array* alternatively consists of any $nb \times N_{tg}$ configuration, provided that $N_{tg} \leqslant nb$. Each *thread array* can process $Nbrows_{ta}$ consecutive block rows, where $Nbrows_{ta} > 1$. A TB is launched with a (T_x, T_y) configuration, where $T_x = nb \times N_{tg}$, and $T_y = N_{ta}$. Every *thread array* is restructured into N_{tg} *thread groups*, each one having nb threads.

Figure 6 shows an example when $bs = nb = 7$ and $N_{tg} = 4$. This leads to $T_x = 28$. The register blocking strategy within a 7×7 block is a variation over the hierarchical register blocking technique mentioned in Section 4.1. However, the value of *ept* is not homogeneous across all *thread groups*. We extend the register blocking technique to allow *thread groups* to process vertical

DOI: 10.1002/cpe

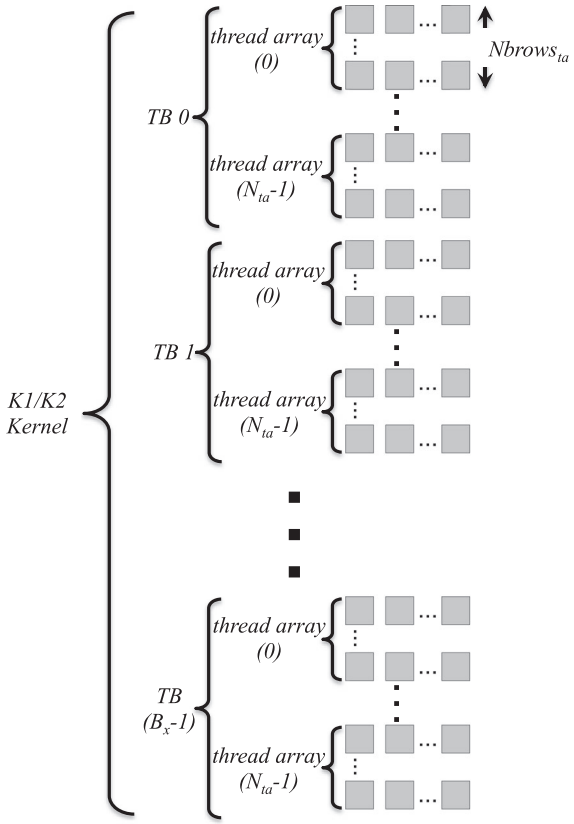


Figure 4. *K1* structure (also applies to *K2*).

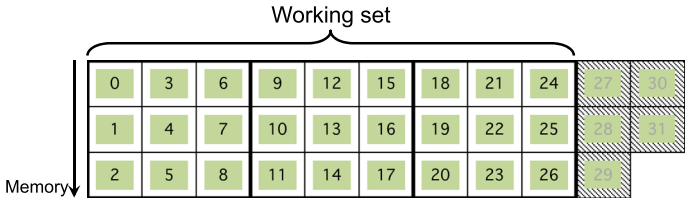


Figure 5. Thread mapping of a truncated warp over three 3×3 blocks.

slices not necessary equal in width. In general, given a block size bs, and a number of *thread groups* $N_{tg} \leqslant bs$, we define the following quantities:

$$ept_{max} = \left\lceil \frac{bs}{N_{tg}} \right\rceil \tag{2a}$$

$$threshold = (N_{tg} - (bs \bmod N_{tg})) \bmod N_{tg} \tag{2b}$$

We apply these two equations on Figure 6. Each *thread group* keeps at maximum $ept_{max} = 2$ registers per *block*. This also includes *thread group* 0 in the figure. In general, only *thread groups* $\geqslant$ *threshold* use ept_{max} registers. Other *thread groups* use ept_{max}-1 registers. According to Figure 6, *threshold* is equal to 1. The double buffering technique (Section 4.2) is applied on the level of one *block*. So another 7×7 block is prefetched while the current one is being computed.

We emphasize that *K2* allows multiple warps to be involved in the computation of a single *block*. For example, a 17×17 block can be processed using 68 active threads (from three warps). Threads are restructured into 4 *thread groups*, meaning that $bs = bs = nb = 17$, and $N_{tg} = 4$. In this case

DOI: 10.1002/cpe

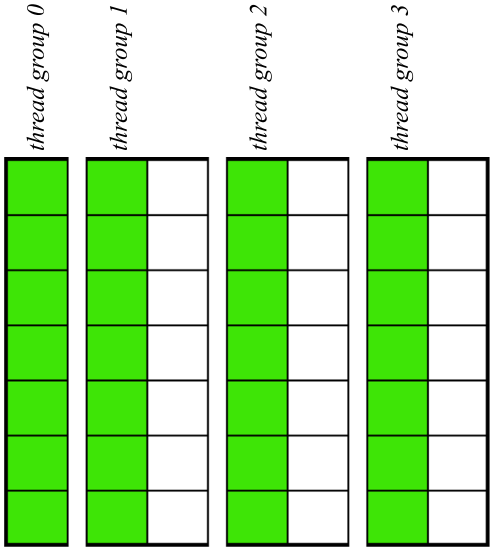


Figure 6. Example of *K2* register blocking on a 7×7 block.

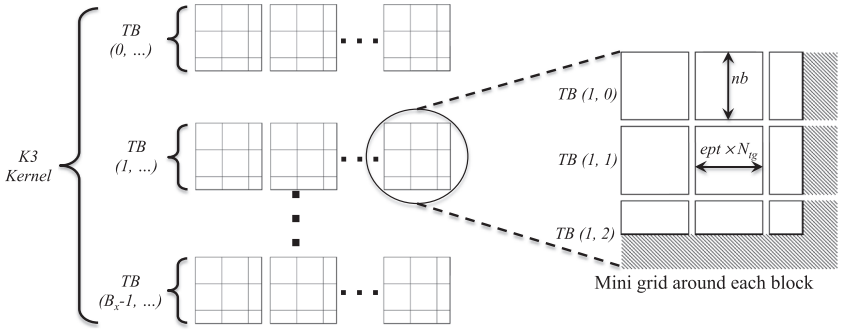


Figure 7. Structure of the *K3* kernel.

$ept_{max} = 5$ and $threshold = 3$. There are chances to achieve ILP per thread as ept_{max} gets bigger. In addition, latency hiding opportunities increase within the same block as more warps are involved in computation.

*5.4. Large blocks (*K3*)*

Kernel *K3* is designed to target large values of bs, including extreme cases when bs can be order of hundreds, for example. Neither *K1* nor *K2* can handle such cases because the hardware constraints might lead to excessive register pressure, running out of shared memory, or even launching a number of threads beyond the capacity of the SM. The *matrix block* is too large to be processed using a single *thread array*, or even a single TB. Kernel *K3* is designed to handle such cases. It is the only kernel that incorporates multiple TBs to process a *matrix block*. Each square *block* of the sparse matrix represents multiple *working sets*. Similarly, a *working set* has the dimensions $nb \times w$, where $nb \leqslant bs$. The width of the *working set* w is equal to $N_{tg} \times ept$, which are discussed shortly in this section. The grid is configured as a 2D array of size (B_x, B_y), where B_x is computed similarly to the past two kernels, and $B_y = \left\lceil \frac{bs}{nb} \right\rceil$. All TBs that have the same b_x value behave like a mini-grid around every *matrix block*. Figure 7 shows the structure of the *K3* grid, with an example where $B_y = 3$. Every three TBs sharing the same b_x value process the same block row. The mini-grid is designed similarly to a dense GEMV kernel.

DOI: 10.1002/cpe

A TB consists of one *thread array*, which means that N_{ta} is equal to 1. A *thread array* consists of $nb \times N_{tg}$ threads. The number of *thread groups* N_{tg} is a tuning parameter. Each *thread group* processes a vertical slice of the *working set*. The width of each vertical slice is equal to *ept*, which is exposed as a tuning parameter. The width of the *working set* is, therefore, equal to $ept \times N_{tg}$. A TB is launched with a (T_x, T_y) configuration, where $T_x = nb$, and $T_y = N_{tg}$.

5.5. Load balancing

The original design of the SpMV kernel [2] uses exactly one TB to compute the product of a block of rows with the input vector. A TB completely traverses all the rows assigned to it. This property might lead to load imbalance among thread blocks, if there are large variations among the row lengths of the input matrix. The performance results section 6 highlights some examples of this type of matrix.

We propose a solution to this problem, and improve the performance of the SpMV kernel for such matrix structures. The solution is to restructure the matrix, by breaking down relatively long rows into multiple independent rows. Given a maximum limit λ of any row length, the new matrix have rows whose lengths cannot exceed λ. Choosing a relatively low value of λ (e.g., 16 or 32), the workload among most TBs is acceptably balanced. A consequence of the restructuring is that the output of the SpMV operation is not the final output y. Figure 8 shows an example for *balanced BSR* layout, with $\lambda = 3$.

In order to represent the new matrix, the row pointer array `RowPtr` is replaced by a new one that reflects the new structure. Neither the array of non-zeros nor the array of column indices needs to be touched. However, it is necessary to have an auxiliary array, which we call *Segment Pointer* (`segPtr`), in order to retrieve the output vector y from $\hat{y}$. The length of the `segPtr` array is equal to the number of block rows of the original matrix + 1. It stores, for each block row in the original matrix, an integer that points to the first entry in $\hat{y}$ that holds a partial result for such block row. For example, the corresponding `segPtr` array for $\hat{y}$ in Figure 8 is $\{0, 2, 3, 4, 8, 10, 11, 14\}$. A reduction operation is required to sum together all the elements in $\hat{y}$ that belong to the same entry in y. We developed a reduction kernel that produces y given $\hat{y}$ and the `segPtr` array. The computation time of the reduction kernel is essentially negligible with respect to the SpMV operation.

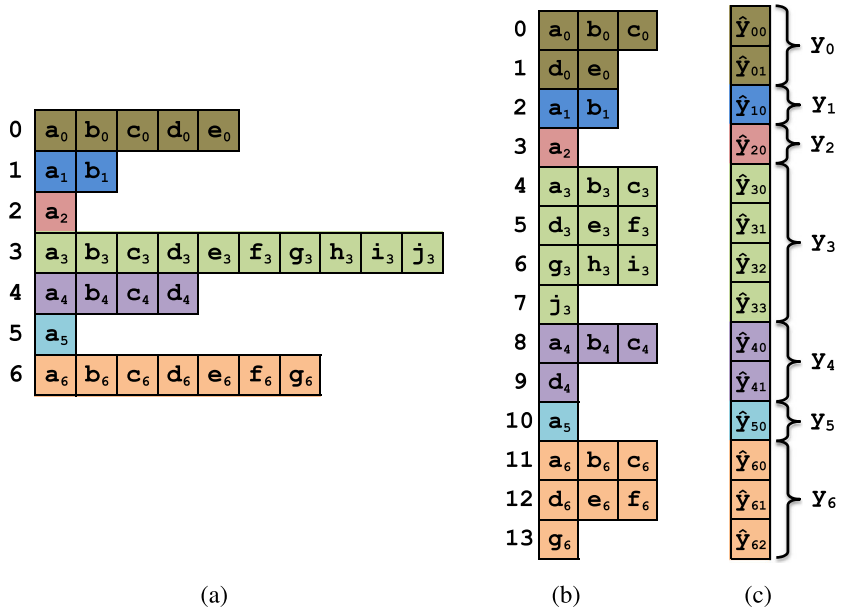


Figure 8. Balanced BSR layout ($\lambda = 3$) and the required postprocessing reduction step. (a) Matrix layout, (b) New matrix layout, and (c) Reduction on the output vector.

DOI: 10.1002/cpe

Table I. Tuning parameters for the BSRMV kernel.

bs	working set	block row assigned to	Tuning parameters	Restrictions
small	multiple blocks	warp	$N_{ta}, Nbrows_{ta}$	None
medium	one block	thread array	$N_{ta}, N_{tg}, Nbrows_{ta}$	$N_{tg} \leqslant bs$
large	part of a block	1 or more TBs	nb, N_{tg}, ept	$N_{tg} \times ept \leqslant nb \leqslant bs$

5.6. Multi-GPU kernels

The methodology for multi-GPU execution is a simple extension. Block rows of the matrix, preferably reordered according to their lengths, will be distributed among GPUs in a 1D cyclic manner. The `RowPtr` array of the BSR format has to be locally updated so that each GPU ends up having a submatrix that is described using the BSR format. Using this methodology, the single GPU kernel can be used on each submatrix. Since each GPU will compute the final result of certain segments of the output vector, a post-processing step is required to collect these segments and put them in place to produce the final vector.

However, this distribution, which has been used in [2], does not work well for all matrices, unless the row lengths are balanced, as proposed in Section 5.5. We will show the multi-GPU performance before and after incorporating the balanced BSR format for the input matrix. After the matrix is distributed among GPUs, the multi-GPU SpMV has the following steps:

(1) Each GPU does a local BSRMV operation.
(2) Each GPU sends its local result into a reference GPU (e.g., GPU 0).
(3) GPU 0 launches a *shuffle* kernel to reverse back the 1D cyclic distribution of $\hat{y}$.
(4) If balanced BSR layout is used, the reduction kernel mentioned in Section 5.5 is launched.

5.7. Tuning parameters

Table I summarizes the tuning parameters of the three proposed kernels. It also shows the granularity level used to process a block row, which is the major difference among the three kernels. As the block size gets bigger, we move from a single warp to a general *thread array*, and eventually to one or more TBs. It also shows how a working set evolves from multiple blocks, to a single block, and finally to a sub-block.

In order to tune the three kernels according to each GPU model, we adopt a simple brute-force approach. This is feasible in our case because the search space for the tuning parameters of each kernel is relatively small. Thanks to the restrictions mentioned in Table I, and to our experience with KBLAS [1], we are able to rule out combinations that either violate hardware-specific limitations or are unlikely to produce high performance. Eventually, it was feasible to try out all the remaining combinations for each kernel in a reasonable time.

6. PERFORMANCE RESULTS

6.1. System setup

All experiments are conducted on a machine equipped with two 8-core Intel Sandy Bridge CPUs (Intel Xeon E5-2670, running at 2.6 GHz), and 3 Kepler generation GPUs (Tesla K40c, running at 745 MHz, with ECC on). We use CUDA Toolkit 7.0. For economy of space, performance results are shown for double precision arithmetic. No surprises are attached to other precisions or complex formats. From now on, the proposed SpMV kernels will be referred to as KSPARSE and balanced KSPARSE.

6.2. Matrix test suite

To show performance results on matrices from real applications, we conducted two types of performance tests. The first one uses matrices from The University of Florida (UFL) Sparse Matrix

DOI: 10.1002/cpe

Table II. Properties of the selected matrices from real applications.

Name	Size	Non-zeros	Description
airfoil_2d	14,214	259,688	Computational fluid dynamics
bauru5727	40,366	145,019	Eigenvalue/model reduction problem
cage10	11,397	150,645	Directed weighted graph
cavity21	4,562	131,735	Subsequent computational fluid dynamics
coater2	9,540	207,308	Computational fluid dynamics
hvdc1	24,842	158,426	Power network problem
lhr07	7,337	154,660	Chemical process simulation
rajat22	39,899	195,429	Circuit simulation
shermanACb	18,510	145,149	2D/3D problem
spe5Ref_dpdp	2,058,000	113,464,400	Oil reservoir simulation

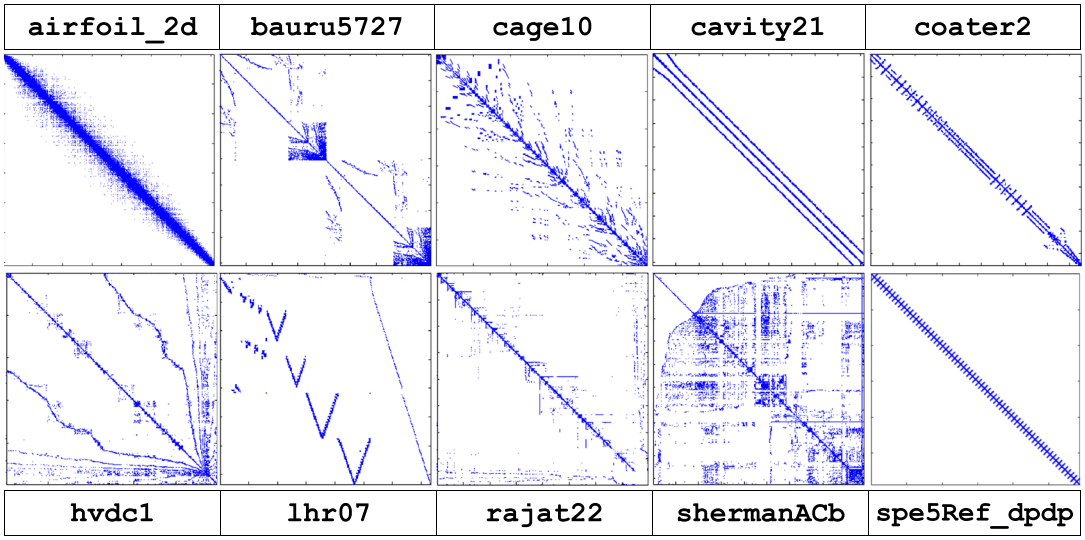


Figure 9. Structures of the selected matrices from real applications.

Collection [3]. Such matrices are not necessarily block-sparse; however, we are interested in their structures, as inherited from spatial discretization. In this regard, performance tests for a certain block size bs replaces each non-zero of the matrix by a $bs \times bs$ square block. Such approach enables us to test the performance of the proposed kernels against a wide range of sparsity patterns. The second test uses a real block-sparse matrix used in reservoir simulation. The matrix is originally based on the SPE5 benchmark [25], and has a dense block structure of size 7×7. Table II shows some properties of the selected matrices. All matrices are square and non-symmetric. Figure 9 shows the structures of these matrices. Not all of these matrices are relevant to simulations involving PDEs, but they are included for generality of applicability.

6.3. Performance on matrices from The UFL sparse matrix collection

Figures 10–18 show the double precision performance of the SpMV operation for the aforementioned matrices (one figure per matrix). We show the performance of the following libraries:

- cuSPARSE using the BSR format [6].
- cuSPARSE using the HYB format [6] [4].
- MAGMA using the SELL-P format [12].
- KSPARSE using BSR format [2].
- Balanced KSPARSE using BSR plus the load balancing technique, with $\lambda = 16$. The performance of this kernel includes the reduction kernel that produces the final output vector.

DOI: 10.1002/cpe

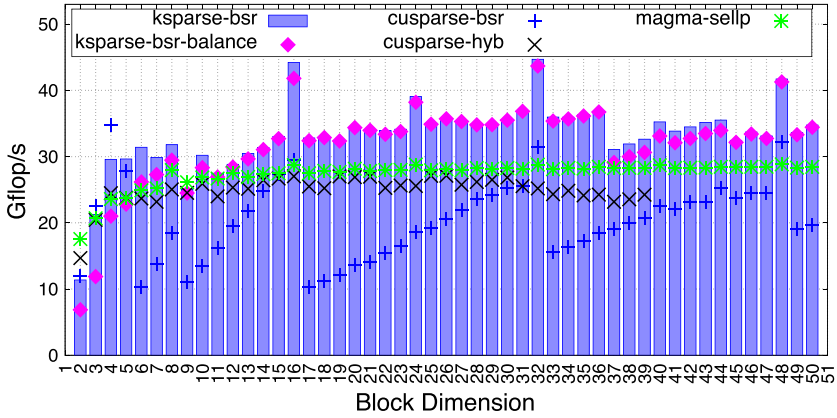


Figure 10. Performance of DBSRMV for the **airfoil_2d** matrix.

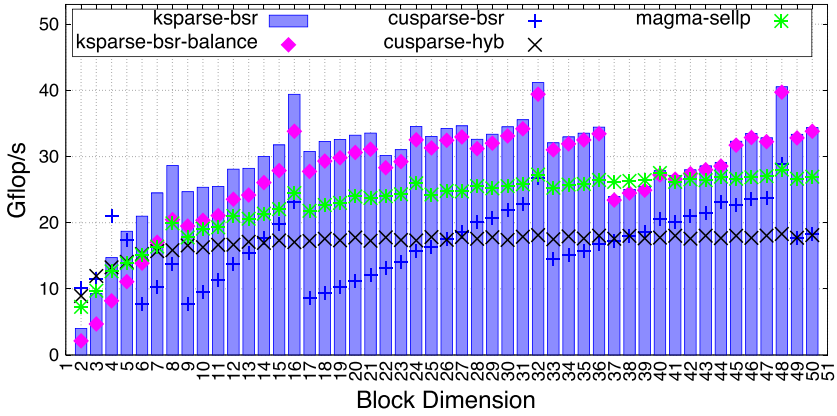


Figure 11. Performance of DBSRMV for the **bauru5727** matrix.

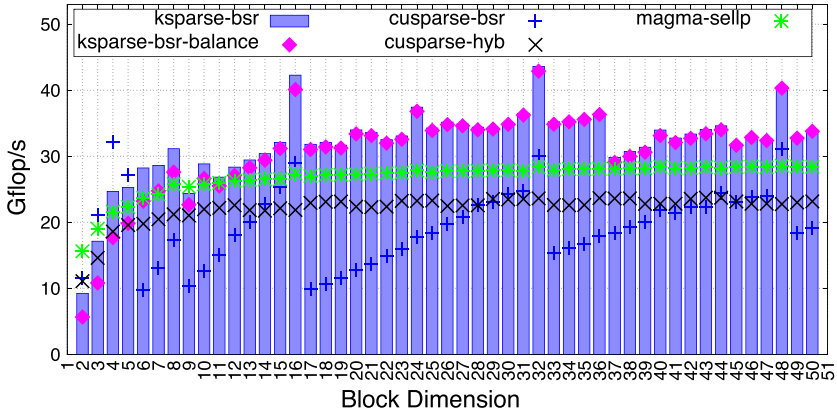


Figure 12. Performance of DBSRMV for the **cage10** matrix.

The performance graphs for seven of these matrices matrices share a common behavior. The proposed KSPARSE kernel outperforms all others after a block size around seven. The balanced version of KSPARSE scores no clear advantage over the unbalanced kernel, which means that the structure of such matrices provides an acceptable balanced workload, and that there is no need to restructure the matrix, because it leads to an extra overhead in terms of the reduction kernel. Table III shows speedups against cuSPARSE-BSR, cuSPARSE-HYB, and MAGMA-SELLP. We point out

DOI: 10.1002/cpe

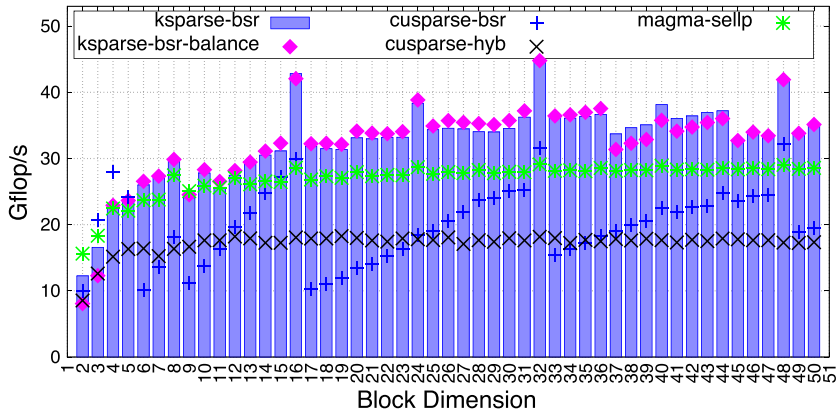


Figure 13. Performance of DBSRMV for the **cavity21** matrix.

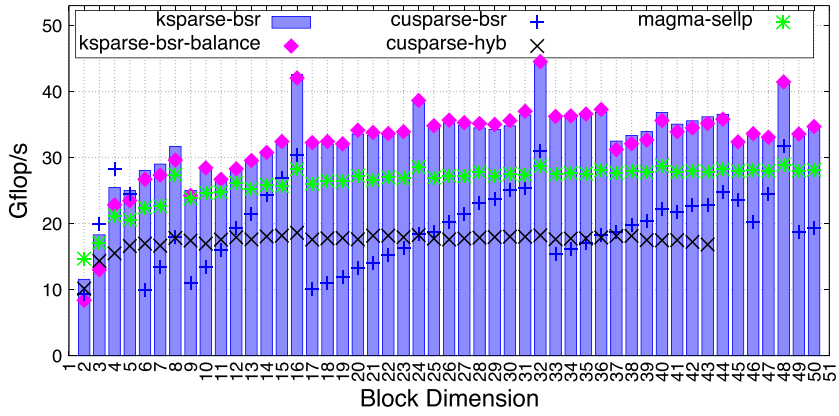


Figure 14. Performance of DBSRMV for the **coater2** matrix.

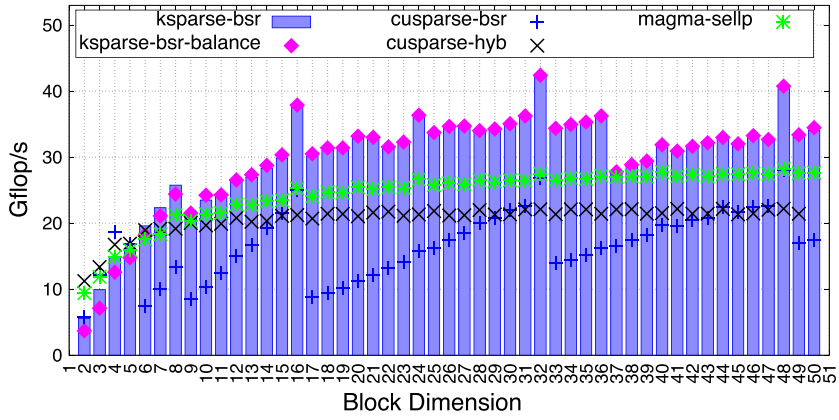


Figure 15. Performance of DBSRMV for the **hvdc1** matrix.

that KSPARSE is at least 3× faster than cuSPARSE using the same sparse storage format, thanks to the optimization techniques discussed in Sections 4 and 5.

Now, we highlight two matrices that show the importance of the balanced KSPARSE kernel. The performance graphs for matrices **rajat22** and **shermanACb** are shown in Figures 17 and 18, respectively. The performance of the balanced KSPARSE kernel is up to 6.18× faster than KSPARSE for the **rajat22** matrix, and up to 15× faster for the **shermanACb** matrix. The reason behind such huge

DOI: 10.1002/cpe

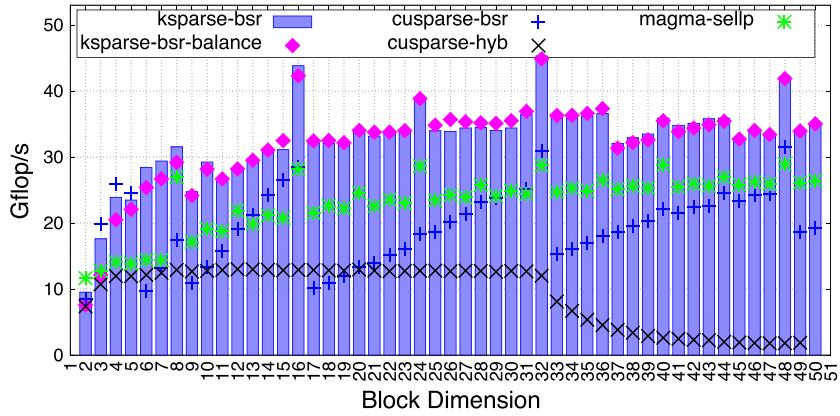


Figure 16. Performance of DBSRMV for the **lhr07** matrix.

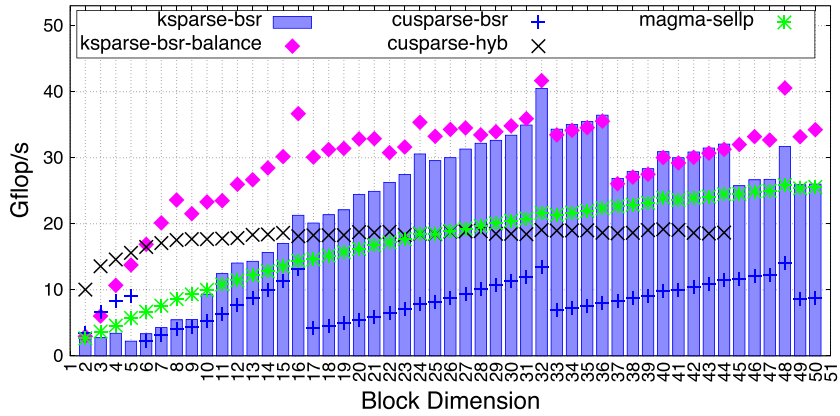


Figure 17. Performance of DBSRMV for the **rajat22** matrix.

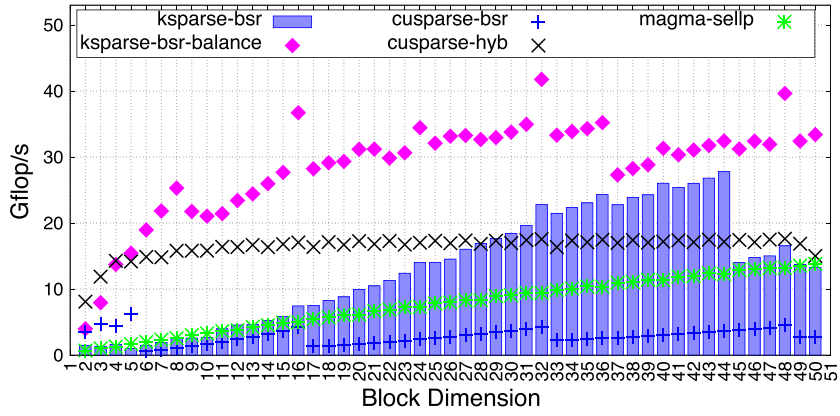


Figure 18. Performance of DBSRMV for the **shermanACb** matrix.

speedups can be explained by inspecting the histograms of the row lengths of these two matrices. Figure 19 shows such histogram for the **shermanACb** matrix, which is also similar in behavior to the histogram of the **rajat22** matrix (not shown). Both matrices have very large number of short rows, and very few number of extremely long rows. Such structures create a high load imbalance for the original KSPARSE kernel [2], due to the fact that it assigns an entire row to the same group of threads. By using the balanced KSPARSE kernel, we can maintain a balanced workload regardless of the matrix structure. With respect to cuSPARSE-BSR, cuSPARSE-HYB and MAGMA-SELLP,

DOI: 10.1002/cpe

Table III. Speedup of KSPARSE against other implementations.

Matrix / speedup against	cuSPARSE-BSR	cuSPARSE-HYB	MAGMA-SELLP
airfoil_2d (Figure 10)	3.17×	1.77×	1.55×
bauru5727 (Figure 11)	3.54×	2.31×	1.61×
cage10 (Figure 12)	3.23×	1.93×	1.54×
cavity21 (Figure 13)	3.09×	2.48×	1.54×
coater2 (Figure 14)	3.16×	2.44×	1.55×
hvdc1 (Figure 15)	3.45×	1.92×	1.55×
lhr07 (Figure 16)	3.16×	23.13×	2.05×

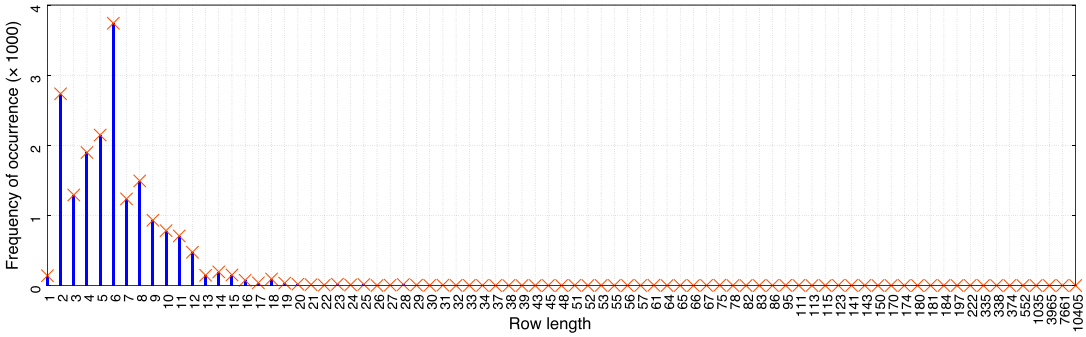
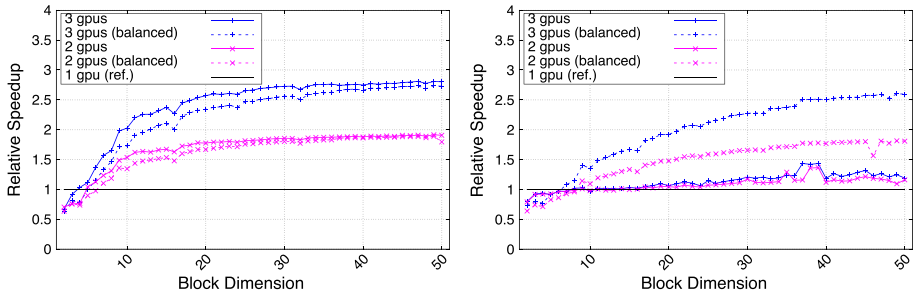


Figure 19. Row lengths histogram of the **shermanACb** matrix.



(a) The **coater2** matrix. (b) The **shermanACb** matrix.

Figure 20. Multi-GPU scaling for the **coater2** and **shermanACb** matrices.

the balanced KSPARSE kernel scores speedups up to 8.86×, 2.26×, and 3.11× for the **rajat22** matrix, and up to 30.25×, 2.38×, and 10.50× for the **shermanACb** matrix, respectively.

Considering multi-GPU performance, we highlight only two matrices, **coater2** as an example of matrices with relatively balanced row lengths, and **shermanACb** as an example of matrices with huge variations in row lengths. Figure 20 shows the relative speedup for such matrices. The results include the communication time needed to send all local results to the reference GPU, the time of the shuffle kernel, and the time of the reduction step if balanced KSPARSE is used.

Considering the **coater2** matrix (Figure 20(a)), inter-GPU communication causes performance across multiple GPUs to drop below unity for small block sizes. The amount of computation in such case is not enough to saturate the GPU with enough work. As the block size gets larger, this effect consistently diminishes. The balanced KSPARSE kernel does not scale as good as the original one, since the single GPU performance of the balanced version is not better than the unbalanced kernel. However, the gap between the two kernels shrinks as the overhead of the reduction kernel gets smaller.

On the other hand, Figure 20(b) shows the impact of the balanced kernel on the multi-GPU performance. The performance of the original kernel on 3 GPUs is under 50% speedup with respect

DOI: 10.1002/cpe

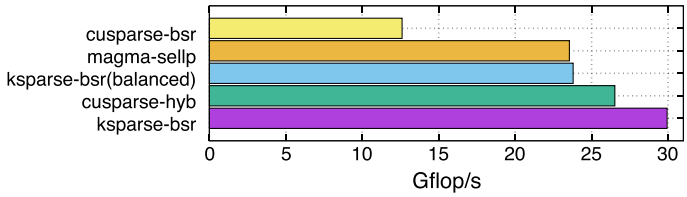


Figure 21. Single GPU performance for the **spe5Ref_dpdp** matrix.

to the single GPU performance. This is due to the 1D cyclic distribution of block rows, without consideration of their variation in length. The balanced kernel achieves much better scaling on 2 and 3 GPUs. The multi-GPU performance of the balanced kernel has three sources of overhead, (i) communication among GPUs, which is more costly than the original kernel because $\hat{y}$ is longer than y, (ii) the shuffle kernel to put all entries of $\hat{y}$ in the right place, and (iii) the postprocessing reduction kernel.

6.4. Performance on matrices from reservoir simulation

Figure 21 shows the single GPU performance on the **spe5Ref_dpdp** matrix. KSPARSE is 2.4× faster than cuSPARSE-BSR, 1.13× faster thank cuSPARSE-HYB, and 1.27× faster than MAGMA-SELLP. The balanced KSPARSE kernel does not score performance gain against KSPARSE, which means that the row lengths are balanced to some extent, which can be seen from Figure 9. In addition, we conducted a multi-GPU performance experiment, where the 2-GPU performance is 1.93× faster than the single GPU test. Similarly, a speedup of 2.83× was achieved on a 3-GPU run.

7. CONCLUSION AND FUTURE WORK

This paper introduces a high performance SpMV kernel for block-sparse matrices and shows how the design ideas of the KBLAS library, which target dense matrices, can still be applied on a smaller scale to relatively small square blocks in sparse matrices. The paper also presents a new technique to alleviate load imbalance due to large variations in the row lengths of some matrices. Both single and multi-GPU performances of the proposed kernels are highlighted on matrices with realistic structures. In most test cases, the proposed design achieves many-fold speedups against state-of-the-art SpMV kernels.

Future directions include the integration of the proposed kernel into sparse iterative solvers, in order to solve real systems arising from multi-component applications, and enhancing the performance of some auxiliary kernels (shuffle and reduce) for small block sizes. Another direction of interest is to try GPU-specific features, like dynamic parallelism, in order to assign work dynamically based on the length of each block row of the matrix.

ACKNOWLEDGEMENTS

This work is partly supported by Saudi Aramco, through research project RGC/3/1438. The authors would like also to thank NVIDIA for their support and generous hardware donations as well as Pascal Hénon from TOTAL S.A. for fruitful technical discussions.

REFERENCES

1. Abdelfattah A, Keyes D, Ltaief H. Kblas: An optimized library for dense matrix-vector multiplication on gpu accelerators. *ACM Transactions on Mathematical Software* 2015, to appear.
2. Abdelfattah A, Ltaief H, Keyes D. High performance multi-gpu spmv for multi-component pde-based applications. In *Euro-Par: Parallel Processing*, vol. 9233, Lecture Notes in Computer Science. Springer Berlin Heidelberg: Vienna, 2015; 601–612.
3. Davis TA, Hu Y. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software* 2011; **38**(1):1:1–1:25.

DOI: 10.1002/cpe

4. Bell N, Garland M. Implementing sparse matrix-vector multiplication on throughput-oriented processors. *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, Portland, Oregon, 2009; 18:1–18:11.
5. Kincaid D, Oppe T, Young D. Itpackv 2d user's guide, 1989. (Available from: http://www.ma.utexas.edu/CNA/ITPACK/manuals/userv2d/) [Accessed on 19 April 2016].
6. The nvidia cuda sparse matrix library (cusparse). (Available from: http://developer.nvidia.com/cuSPARSE) [Accessed on 19 April 2016], owner = abdelfattah.
7. Monakov A, Lokhmotov A, Avetisyan A. Automatically tuning sparse matrix-vector multiplication for gpu architectures. *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*, Pisa, Italy, 2010; 111–125.
8. Vázquez F, Fernández JJ, Garzón EM. A new approach for sparse matrix vector product on nvidia gpus. *Concurrency and Computation: Practice and Experience* 2011; **23**(8):815–826.
9. Choi JW, Singh A, Vuduc RW. Model-driven autotuning of sparse matrix-vector multiply on gpus. *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, Bangalore, India, 2010; 115–126.
10. Kreutzer M, Hager G, Wellein G, Fehske H, Basermann A, Bishop AR. Sparse matrix-vector multiplication on gpgpu clusters: A new storage format and a scalable implementation. *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Shanghai, China, 2012; 1696–1702.
11. Kreutzer M, Hager G, Wellein G, Fehske H, Bishop A. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units. *SIAM Journal on Scientific Computing* 2014; **36**(5):C401–C423.
12. Anzt H, Tomov S, Dongarra J. Implementing a sparse matrix vector product for the sell-c/sell-c-σ formats on nvidia gpus. *Technical Report*, Innovative Computing Laboratory (ICL): Knoxville, Tennessee, 2014. (Available from: http://www.icl.utk.edu/sites/icl/files/publications/2014/icl-utk-772-2014.pdf) [Accessed on 19 April 2016].
13. Ashari A, Sedaghati N, Eisenlohr J, Parthasarathy S, Sadayappan P. Fast sparse matrix-vector multiplication on gpus for graph applications. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, New Orleans, Louisana, 2014; 781–792.
14. Nvidia kepler gk110 architecture whitepaper. (Available from: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf) [Accessed on 19 April 2016].
15. Greathouse JL, Daga M. Efficient sparse matrix-vector multiplication on gpus using the csr storage format. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, New Orleans, Louisana, 2014; 769–780.
16. Im EJ, Yelick K. Optimizing sparse matrix computations for register reuse in sparsity. In *International Conference on Computational Science (ICCS)*, vol. 2073, Lecture Notes in Computer Science. Springer Berlin Heidelberg: San Francisco, 2001; 127–136.
17. Im EJ, Yelick K, Vuduc R. Sparsity: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications* 2004; **18**(1):135–158.
18. Godwin J, Holewinski J, Sadayappan P. High-performance sparse matrix-vector multiplication on gpus for structured grid computations. *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, ACM, New York, NY, USA, 2012; 47–56.
19. Dang HV, Schmidt B. Cuda-enabled sparse matrix vector multiplication on gpus using atomic operations. *Parallel Computing* 2013; **39**(11):737–750.
20. Yan S, Li C, Zhang Y, Zhou H. yaspmv: Yet another spmv framework on gpus. *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, Orlando, Florida, 2014; 107–118.
21. Chen Z, Huan G, Ma Y. *Computational Methods for Multiphase Flows in Porous Media*. Society for Industrial and Applied Mathematics, 2006. (Available from: http://epubs.siam.org/doi/abs/10.1137/1.9780898718942).
22. Williams FA. *Combustion Theory : The Fundamental Theory of Chemically Reacting Flow Systems*, Combustion Science and Engineering Series. Benjamin/Cummings: Menlo Park, California, 1985.
23. Keyes DE, McInnes LC, Woodward C, Gropp W, Myra E, Pernice M, Bell J, Brown J, Clo A, Connors J, Constantinescu E, Estep D, Evans K, Farhat C, Hakim A, Hammond G, Hansen G, Hill J, Isaac T, Jiao X, Jordan K, Kaushik D, Kaxiras E, Koniges A, Lee K, Lott A, Lu Q, Magerlein J, Maxwell R, McCourt M, Mehl M, Pawlowski R, Randles AP, Reynolds D, Rivire B, Rde U, Scheibe T, Shadid J, Sheehan B, Shephard M, Siegel A, Smith B, Tang X, Wilson C, Wohlmuth B. Multiphysics simulations: Challenges and opportunities. *International Journal of High Performance Computing Applications* 2013; **27**(1):4–83.
24. Knoll DA, Keyes DE. Jacobian-free Newton–Krylov methods: a survey of approaches and applications. *Journal of Computational Physics* 2004; **193**(2):357–397.
25. Killough JE, Kossack CA. Fifth comparative solution project: Evaluation of miscible flood simulators. *SPE Symposium on Reservoir Simulation*, San Antonio, Texas, 1987.

DOI: 10.1002/cpe