J Supercomput (2017) 73:3761–3795
DOI 10.1007/s11227-017-1972-3

SpMV and BiCG-Stab optimization for a class of hepta-diagonal-sparse matrices on GPU

Mayez A. Al-Mouhamed[1] · **Ayaz H. Khan**[2]

Published online: 24 March 2017

Abstract The abundant data parallelism available in many-core GPUs has been a key interest to improve accuracy in scientific and engineering simulation. In many cases, most of the simulation time is spent in linear solver involving sparse matrix–vector multiply. In forward petroleum oil and gas reservoir simulation, the application of a stencil relationship to structured grid leads to a family of generalized hepta-diagonal solver matrices with some regularity and structural uniqueness. We present a customized storage scheme that takes advantage of generalized hepta-diagonal sparsity pattern and stencil regularity by optimizing both storage and matrix–vector computation. We also present an in-kernel optimization for implementing sparse matrix–vector multiply (SpMV) and biconjugate gradient stabilized (BiCG-Stab) solver. In-kernel is intended to avoid the multiple kernels invocation associated with the use of the numerical library operators. To keep in-kernel, a lock-free inter-block synchronization is used in which completing thread blocks are assigned some independent computations to avoid repeatedly polling the global memory. Other optimizations enable combining reductions and collective write operations to memory. The in-kernel optimization is particularly useful for the iterative structure of BiCG-Stab for preserving vector data locality and to avoid saving vector data back to memory and reloading on each kernel exit and re-entry. Evaluation uses generalized hepta-diagonal matrices that derives from a range of forward reservoir simulation's structured grids. Results show the prof-

✉ Mayez A. Al-Mouhamed
mayez@kfupm.edu.sa
http://faculty.kfupm.edu.sa/COE/mayez/

Ayaz H. Khan
ay.khan@qu.edu.sa

1 Computer Engineering Department, King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia

2 Computer Science Department, Qassim University, Buraydah, Saudi Arabia

itability of proposed generalized hepta-diagonal custom storage scheme over standard library storage like compressed sparse row, hybrid sparse, and diagonal formats. Using proposed optimizations, SpMV and BiCG-Stab have been noticeably accelerated compared to other implementations using multiple kernel exit–re-entry when the solver is implemented by invoking numerical library operators.

Keywords Reservoir simulation · BiCG-Stab · SpMV · GPU · CUDA Optimization · Structured grid · Synchronization · Coalesced memory

1 Introduction

Modern scientific and engineering problems dominate a wide range of daily applications with higher accuracy that translate into a much larger scale of simulation such as high-energy physics, scientific simulation, data mining, climate forecast, and earthquake prediction. Graphics processing units (GPUs) become one of the most significant devices in high-performance computing since past few years [32]. Compute Unified Device Architecture (CUDA) developed by NVIDIA is available as general-purpose programming framework for GPUs and easy to learn [2] as an extension to C/C++ programming.

The GPU tremendous computing power is extremely useful for accelerating many science applications which are based on discrete numerical simulation. One important class is the structured grid computation (SGC). Hence, there is a great interest to develop systematic optimization approaches for accelerating SGCs using emerging GPUs. Generally, an SGC simulation consists of repeatedly solving a large system of linear equations using direct and *Iterative Numerical Algebra Algorithms* (INAAs). Direct methods are more reliable, more accurate but require large memory space and generally lacks scalability in many cores due to their sequential nature. The nearest-neighbor stencil used in structured grids dictate the matrix to be sparse with problem-dependent distribution of elements. Hence, INAAs have abundant data parallelism with large sparse systems of linear equations. Among the most popular iterative methods to solve large sparse linear system is the Krylov subspace solvers [33]. The BiCG-Stab technique is one of the example of these solvers. The most frequent operators are the matrix–vector multiplication, vector inner product, and vector addition and scaling [2]. The techniques used to solve these large linear systems need considerable time, during which a small portion of code is repeatedly invoked for most of the simulation time [33]. GPU acceleration of the above operations [22] primarily aims at enhancing the simulation accuracy. Several bottlenecks which may limit performance of the GPUs are the low efficiency due to the sparse solver matrix which increases the overhead with irregular accesses to the memory which imposes serious limitations on memory bandwidth [1].

Generally scientists use math operators and even complete solvers which are available within a number of numerical libraries (NLs) to alleviate the complexity of many-core programming. The academia as well as the GPU manufacturers have developed efficient NLs with optimized math operators to ease the process of simulating approximate solutions to nonlinear partial differential equations (PDEs). Generally,

NLs provide fast, accurate and portable parallel simulation programs [27,34] involving dense and sparse linear algebra, a variety of sparse matrix storage formats, iterative solvers, and sometimes tools to define data structures for science simulation [7,17]. Below is a list of some NLs that have support to CUDA and GPUs:

- CUSP [31] for generic parallel algorithms with sparse linear algebra, maximal independent set graph, and polynomial relaxation, pre-conditioners for algebraic multi-grid and approximate inverse methods.
- MAGMA [34] for dense matrix algebra for hybrid systems,
- Thrust [18] for object-oriented data structure to support parallel algorithms,
- cuBLAS and cuSparse [27] support for all standard Basic Linear Algebra Subroutines (BLAS). Application Programming Interface (API) for batched generic matrix-multiplication, LU factorization, inverse matrix, device API for CUDA kernels, sparse matrix formats, sparse triangular solve, and incomplete factorization pre-conditioners.
- PETSc [7] has a suite of data structures and routines for vectors and matrices, sparse storage formats, Krylov pre-conditioners, parallel Newton-based nonlinear and time stepping ordinary differential equation solvers.
- Trilinos [17] is a suit of packages such as BLAS, linear solvers and pre-conditioners, optimization solvers, eigen-solvers, load balancing utilities, abstract Interfaces and adapters, mesh generation and management utilities and discretization utilities.

Offloading all computation of matrix and vectors to the GPU is considered a straightforward method to utilize the GPU accelerator in implementing the solvers such as Krylov subspace using the functions available in the libraries such as cuSparse [30] and CUSP [10]. Although this approach provide good improvement in the performance comparing to CPU-based instead of GPU, sometimes the high capability of these accelerators is not exploited due to the limitations caused by the operations of the linear algebra presented by these libraries [4,5].

The SpMV where the multiplication occurs between large sparse matrices with dense vectors is considered one of the most important computational and time-consuming kernels for a wide range of engineering and scientific applications from structural mechanics to quantum physics to fluid dynamics. Thus, to achieve higher performance on this operation, it is important to choose an sparse matrix storage format and utilize the underlying GPU architecture carefully. Section 2 summarizes the works related to SpMV implementations using different sparse matrix formats and optimizations targeting GPUs.

To decrease overhead and bottlenecks mentioned above, many studies have proposed the various storage formats for sparse matrices to optimize the access of the memory pattern and reduce the storage space needed to store them in GPU memory such as COOrdinate (COO) format, Compressed Sparse Row (CSR) format, Block Sparse Row (BSR) format, ELLPACK format, HYBrid (HYB) format, and DIAgonal (DIA) format. Each of these formats has different storage requirements, computational properties, and methods for accessing and manipulating elements of the matrix. Section 4 gives a brief overview about most of these basic formats and shows the storage complexity for each of them.

OpenACC is a technology to automate the process of parallelization supporting various architectures including GPUs. All key optimizations are left to the compiler, and parallelization can be accomplished by preceding a code block with the kernels directive written in standard C/C++ programming language. The compiler provides the most warranty of correctness for the parallelized code. The code efficiency is not guaranteed to be accomplished as the parallelization process is completely automated. The compiler lacks the domain knowledge of the application. In the PGIs accelerator model [21], the user has little control over optimizations applied during transformations. The performance of directive-based programming can be as good as the compiler and auto-parallelization technology could withstand. This is particularly the case for the newly emerging GPU-based parallel architectures as they have many architectural parameters to tune. As a result, manual tuning and optimization are always preferred to gain the most performance [16].

In this paper, a structured grid is used in the simulation of a petroleum forward reservoir model. The stencil relationship allow identifying a family of generalized hepta-diagonal (GH) matrices. GH is the main solver matrix, and it is inefficient to store it as is due to its sparse pattern. A customized storage scheme is proposed for GH to optimize the storage requirements and implied arithmetic operations in SpMV. A set of optimizations is also proposed for the BiCG-Stab solver. The enhancements aims at designing the solver algorithm as one single kernel. For this a custom global synchronization is used after combining inner products and global write to memory. Hence, data locality is enhanced altogether with reuse of cached data. In the evaluation we compare our proposed storage scheme, optimized SpMV, and BiCG-Stab to library based implementations using some standard sparse matrix storage formats.

2 Literature survey

It is well known that the most Krylov-space iterative solvers to find solution for a large sparse system of linear equations spend most of their time in the SpMV operator. Hence, optimization of SpMV attracted many researchers with the aim of finding rules for selecting a storage scheme that minimizes the memory requirements depending on the degree of sparsity and regularity in the data structure. Memory efficiency allows handling large problems that may contribute in enhancing the simulation accuracy.

Bell and Garland [8,9] first presented a comprehensive analysis for performance of SpMV on general-purpose GPUs using different storage formats. Analysis shows that the main bottleneck of the COO and CSR is memory bandwidth due to explicit storage for the row and column indices. Hence, the computation to communication ratio is low due to frequent access to global memory of the GPU. Unlike COO and CSR, DIA and ELL formats store both row and columns indices implicitly. Structured-sparse matrices result from common stencil grids in 1-, 2-, and 3-D discretizations. Performance analysis indicates that for unstructured matrices, CSR or HYB obtained the best performance and for structured matrices DIA storage format is most recommended especially when nonzero elements (NZs) are limited to a small number of matrix diagonals.

Bordawekar et al. [11] presented a number of optimization methods over CSR storage format based on address mapping and data access with a single warp of threads assigned to work on two rows. The performance of CSR kernel is compared with Bell and Garland's CSR and HYB kernels [8] using a set of 19 unstructured matrices. Their kernel outperformed Bell and Garland's CSR kernel on almost all matrices and outperformed HYB on some matrices with up to 15% speedup.

Buatois et al. [12] proposed blocked-CSR (BCSR) and investigated the performance of BCSR on G80 series of NVIDIA graphic cards comparing with CSR. They tested BCSR using different sizes for the block and they found that on any device BCSR-4×4 (shortly referred to BSR-4) is significantly faster than 2×2, and 2×2 is significantly faster than CSR.

Ahamed et al. [2] proposed vector CSR storage format for SpMV kernel based on Bell & Garland kernel [8] using Alinea library. They used several types of matrices from the University of Florida repository [13] to compare the performance of their kernel with kernels from cuSparse and CUSP libraries. They found that their implementation outperforms cuSparse and CUSP libraries for double-precision computation.

A hybrid processing method (HPM) for SpMV [19] is proposed by enhancing the standard compress sparse row (CSR) with NZ values. HPM consists of sorting the size of the rows in a roughly descendant order and store the results with the corresponding row index in a separate 2D array in addition to the arrays used in the standard CSR scheme. HPM aims at solving the problem of thread load balancing that affects the CSR scheme by assigning equal number of operations among threads. The performance of HPM and hybrid SpMV kernels are compared with kernels that used scalar CSR and vector CSR formats. Results show that HPM outperforms both of the above schemes.

A static method is proposed [26] to predict the optimal storage format depending on the input matrix and CPU–GPU time to transfer matrix data under a subset of storage formats and a storage preprocessing overhead. The overhead is due to the number of accessed data structures, iterations, and number of NZs per row. The Sparse Matrix Collection of Florida University is used in the evaluation. Evaluation shows that CSR format is preferred for sparse square and non-square matrices if the number of rows is less than the number of columns, and otherwise ELL is selected. A Bit-Level Single-Index (BLSI) representation is proposed to reduce the memory foot print and preprocessing overhead when the prediction model selects the HYP or ELL. Once the sparse matrix storage scheme is selected or designed, the next important issue is to enhance the SpMV execution model in view of the sparse data structure. Optimizing SpMV explores ways of mapping computations to threads to avoid a sporadic indexing and to favor streaming memory accesses. Another important optimization is to enhance load balancing among the threads and reduce the synchronization overheads. In the following we explore contributions to the above area.

A method to improve the performance of the SpMV kernel is proposed [24] based on the analysis of sparse matrix features such as the average number of NZs in a row, the NZ standard deviation, etc. The focus is on load balancing of thread work. A combination of CSR and ELL formats is used. To address the issue of load balance when using this combination, a number of warps are assigned to a row and the CSR format is selected when the number of NZs exceed some empirically determined

threshold, and otherwise, the ELL format is used. Other thresholds are used on the minimum and maximum number of NZs for warps assigned to the CSR and ELL formats, respectively. In the evaluation, a set of 14 unstructured sparse matrices with varying degree of sparsity are used. Proposed storage improves the performance by a factor of 25% on average compared with the best results of HYB proposed by Bell and Garland [8].

Greathouse et al. [15] showed that standard CSR-based SpMV on GPUs has poor performance due to irregular memory access patterns, load imbalance, and reduced parallelism. Simple assignment of results to threads causes imbalance among the threads and un-coalesced memory access. These are responsible of resource under utilization. In the CSR-Vector storage, one result is assigned to a warp which allows coalesced access to memory. However, poor resource occupancy is noticed due to mismatch between the number of NZs in a row and the fixed number of threads in a warp. For this non-CSR storages have been proposed which causes some engineering hurdle as most software use CSR and the need for large runtime overhead and storage for inter-format conversion. In the CSR-Adaptive storage, a warp is assigned a fixed number of NZs which are streamed into shared memory using coalesced access and dynamically assign different number of rows to each warp. Evaluation shows that CSR-adaptive achieves an average speedup of 14.7 over existing CSR-based algorithms and 2.3 over clSpMV cocktail, which uses an assortment of matrix formats.

Implementing the SpMV using the common coordinate COO suffers from load imbalance and high memory bandwidth. A blocked common coordinate storage (BCCOO) proposed [36] to alleviate the bandwidth problem by using bit flags to store the row indices in a blocked common coordinate format. Additionally, the matrix is partition into vertical slices to enhance the cache hit rate when accessing the SpMV operand vector. Finally, the vector scan approach is improved to reduce load imbalance. The SpMV is implemented using OpenCL. A variety of diagonal-sparse, structured-sparse, and dense matrices are used in the evaluation. Proposed storage best performance is measured as between 6×10^{-3} and 15×10^{-3} of peak flops for diagonally dominant sparse matrices and up to 9×10^{-3} of peak flops for GTX 680 (3.09 TFLOPS). These results compares favorably to those obtained using cuSparse and CUSP libraries.

The acceleration of Krylov subspace iterative methods [6] for GPUs proposed based on (1) minimizing the number of kernel launches, (2) reducing GPU-host communication, and (3) enhancing the operator routines. Furthermore, the merging of cuBLAS operators is used to reduce the number of kernel calls. As a result, the number of kernels launches is reduced to eight for the BiCG-Stab compared to 16 cuBLAS function invocations. For the inner product, each thread strides over a complete row, handling one element in every part. At the end, the partial sums computed by the threads are collected using fan in. For the sparse matrix–vector multiply (SpMV), each SM is allocated a column of data, which is broken down into parts with thread block size. Each thread in a block strides over the entire column processing one element in each part. The partial sum computed by different threads are collected using a fan in summation. The cache limitation is overcome by computing data in chunks of vectors depending on block size and precision format. The custom-designed kernel is developed by merging multiple arithmetic operations, and keeping data in shared memory whenever

possible. In the evaluation NxN sparse matrices from the University of Florida Matrix Collection are used for testing the implementation of SpMV and BiCG-Stab using the CSR storage format. Using merged operators, BiCG-Stab is speeded up by a factor of 2 compared to cuBLAS reference implementation for N within the range of 10^5–10^6. A scalable flop performance is shown for the inner product compared to non-scalable implementations from NVIDIA and MAGMA for a vector of 10^5 elements.

An object-oriented toolkit (PETSc) [22] proposed for the automatic generation of optimized CUDA kernels. The tools takes as input some user defined high-level description of numerical solutions for nonlinear PDEs such as the iterative Newton–Krylov methods. The tool uses knowledge of finite difference discretization with stencil-based sparse matrix and generate optimized sparse matrix data structure. CUDA code is further optimized using auto-tuning. In the evaluation, performance of generated kernel is compared to that of CUSP library using the CSR, DIA, ELLPACK sparse matrix storage formats. For SpMV, the generated solutions achieve a speedup of 1.5 over the above library when the number of components is above 6, i.e., independent variables associated with each grid point. For vector inner product and vector-norm, auto-tuning may double the speedup of generated kernel, which shows significant speedup over hand-tuned code, CUSP and cuSPARSE libraries.

For general sparse matrix data layout, an adaptive multi-level blocking (AMB) [25] has been proposed to improve the computational performance of SpMV kernel by reducing the memory traffic with division and blocking optimization techniques and reusability of input vector element in the cache. An auto-tuning mechanism based on estimating the memory traffic and predicting the SpMV computation performance is used to find the optimal parameters. With the AMB format, each segment (column wise and row wise) comprises of sufficient number of NZs to be computed by different threads in the parallelized SpMV CUDA kernel. For irregular sparse matrix data layout, AdELL+ is proposed [23] as an extension to ELL-based adaptive format. A parametrized warp-balancing heuristic scheme is used to balance the warps in CUDA kernel execution. The SpMV kernel for the proposed matrix format has also been developed with several optimizations such as warp granularity, blocking, delta compression, and nonzero unrolling to enhance the efficiency of memory footprint and memory hierarchy in GPUs. The optimal parameters for the SpMV kernel related to underlying architecture is obtained via an online auto-tuning mechanism that hides the preprocessing overhead with useful SpMV computations.

3 GPU architecture

Modern GPUs are throughput-oriented many-core processors that offer very high peak computational throughput. It is attached as a co-processor in the computer system connected to host environment (system motherboard) via peripheral component interconnect (PCI Express 16E) to communicate with the CPU. GPUs are organized into an array of highly threaded streaming multiprocessors (SMs). Each SM has a number of streaming processors (SPs) that share control logic and instruction cache. The GPU used for the experimentation here is NVIDIA Tesla K20x (Kepler architecture) (see Table 1) that comes with a 6 GB of graphics double data rate (GDDR) DRAM referred

Table 1 KFUPM's GPU specification

Property	Value
GPU model	NVIDIA Tesla K20Xm
CUDA driver version/runtime version	6.5/6.0
Compute capability	CUDA 3.5
Total amount of global memory	6144 MB
(14) Multiprocessors, (192) CUDA cores/MP	2688 CUDA cores
Total amount of shared memory per block	49,152 bytes
Total number of registers available per block	65,536
Warp size	32
Maximum number of threads per multiprocessor	2048
Maximum number of threads per block	1024
Max dimension size of a thread block (x, y, z)	(1024, 1024, 64)
Max dimension size of a grid size (x, y, z)	(2,147,483,647, 65,535, 65,535)

to as global memory (GM) that is visible to all threads in all blocks. Each SM has a shared memory (ShM) which is on-chip, readable and writable, and visible to all threads running within SM and as fast as register access. However, ShM is very small in size compared to GM. Each SM schedules one warp at a time with zero overhead warp scheduling. The warp is the unit of thread scheduling in SMs. Each warp consists of 32 threads of consecutive thread ids. In the case of higher dimensional kernels, warps will be retrieved from blocks according to the row-major numbering. As warps executes in SIMD fashion, if there is a high latency exception such as loading data from GM or storing results to GM then the whole warp must be suspended and its context is preserved. A DMA operation is initiated by the SM whenever it finds one or more threads within a warp to perform such a long latency memory transfer operations (accessing global memory) and schedule another warp (ready to execute) to the SPs.

GPUs have their own memory hierarchy with unified memory request paths for loads and stores. This memory hierarchy has been developed from Level-0 (no hierarchy) to Level-3 (L1, L2, Global) with the advancement in the GPU generations (Fermi, Kepler, Maxwell, Pascal). Unlike, CPU cache memory levels, GPU cache memories are read-only memories and doesn't have cache coherency protocols implemented. For example, in kepler (K20X) architecture, each SM has 64 KB of on-chip memory that can be configured to be used as shared memory and L1 cache. L1 cache in kepler can store only local memory accesses such as register spills and stack data, global memory loads are cached in L2 [28]. In addition to L2 cache, global loads can also be cached in compiler directed 48 KB read-only data cache. The read-only path can be managed automatically by the compiler or explicitly by the programmer.

4 Sparse matrix storage schemes

As a basic sparse matrix format, the COOrdinate (COO) format represented by three arrays (Fig. 1b), one is a dense array (Data) which store the nonzero elements (NZs)

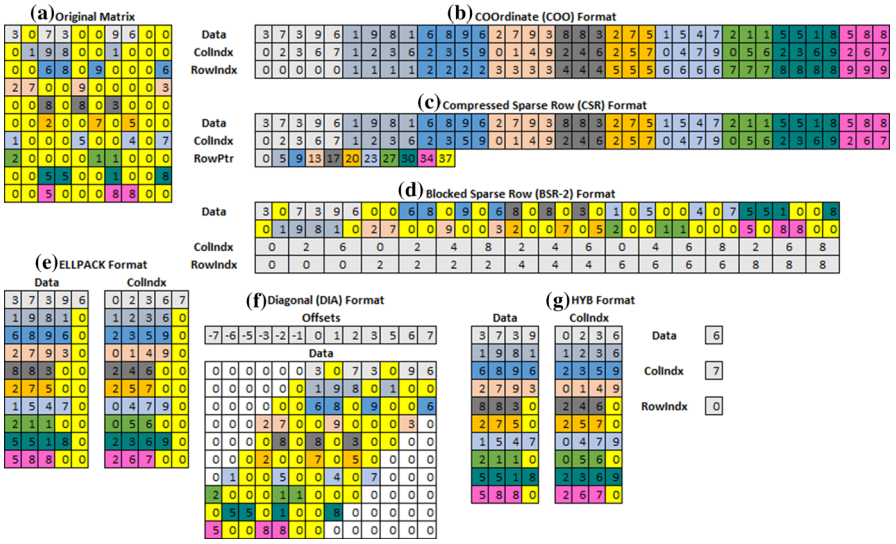

Fig. 1 Examples of common sparse matrix formats. **a** Original matrix, **b** COOrdinate (COO) format, **c** compressed sparse row (CSR) format, **d** blocked sparse row (BSR-2) format, **e** ELLPACK format, **f** DIAgonal (DIA) format, **g** HYB format

of the sparse matrix. The other arrays are a column indices array (ColIndx) and a row indices array (RowIndx). The storage complexity for COO format is O(3NZs).

Compressed Sparse Row Format (CSR) permits indexed access to rows. Similar to COO, it consists of one dense array (Data) to store NZs of the sparse matrix. Two other vectors are used to store the column index (ColIndx) of each element with NZ value and the offset points to start of each row (RowPtr) in the dense array (Data) of the sparse matrix, see Fig. 1c for an example of CSR matrix format. The storage complexity for CSR format is $O(2NZs + Nr + 1)$, where Nr is the number of rows. The CSR format aims at minimizing the storage for matrices with arbitrary sparsity. CSR continuously stores the NZs in each row of the sparse matrix. CSR uses one level of indirection to recover the NZ indices at runtime.

The Blocked Sparse Row format (BSR-k) aims at exploiting the blocked structure for matrices with arbitrary sparsity. Each NZ element of vector in BSR is mapped to a NZ square block of k dimensions. It assumes that the number of NZs in each row is multiple of block size. Additional zeros are stored in a block to satisfy this condition [14]. Similar to CSR storage of NZs, BSR stores blocks of data by expanding a kxk block in a row-major fashion with fixed size and possible padding with zeros (Fig. 1d). This reduces address calculation in recovering the NZ indices at runtime.

In the ELLPACK, a dense matrix (Data) with dimensions of $N \times N_{\max}$ is used to store the NZs, where $N_{\max}$ is the NZs in the longest row. Another matrix (ColIndx) with same dimension is used to store column index of each element, see Fig. 1e for an example of ELLPACK format. The Data and ColIndx matrices are padded with zeros in unused positions. The storage complexity for ELLPACK format is $O(2(N+N_{\max}))$. The ELL storage format is well suited to vector architectures. An MxN block having

at most k NZs per row is stored as an Mxk block with possible padding with zeros. The column indices are also stored.

The DIAgonal (DIA) storage format is specially utilized for sparse matrices contains the NZs along diagonals and it is represented by two arrays (Fig. 1f), one for NZs and the other to store the positions of the diagonals in the sparse matrix. The complexity of DIA format is $O((N_{\text{diag}}.xN) + N_{\text{diag}})$ where N_{diag} is number of the diagonals in the sparse matrix. The DIA format aims at efficiently storing sparse matrices having dominant NZs along the matrix diagonals. DIA stores NZ diagonals as rows of contiguous elements, starting from the lowest diagonal up to the highest diagonal. For each diagonal, the offset from the central diagonal is also stored.

The hybrid format (HYB) aims at storing the typical number of NZs per row in the ELL data structure and the remaining entries of exceptional rows in the COO format; see Fig. 1g for an example of HYB format. Sometimes, the largest number of NZs k per row is a structure-dependent parameter. In this case, k columns per row are used. One general approach is to add a Kth column if at least one-third of the matrix rows contain K or more NZs. When the number of rows with K or more NZ values is below the one-third of the matrix, the remaining NZ values are placed into COO.

5 Forward reservoir simulation

A petroleum forward reservoir simulation (FRS) represents a model of fluid flow and mass transfer in porous media. The objective of FRS simulation is to provide comprehensive information on flow variables and well responses. Generally, FRS is a 3D flow process involving a multi-phase process in a porous medium, i.e., a solid containing void pores dispersed in a regular or random manner. For example, a two-phase process may consists of water and oil. In general, the medium has a number of state variables such as the permeability, which is the rock capacity to transmit fluid, porosity, oil pressure, water saturation as well as a number of interacting forces such as gravity and capillary.

5.1 FRS-structured grid

A multi-phase oil reservoir may extend over a large field and may include tens of wells. The FRS model is based on linearizing a set of nonlinear PDE equations using the Newton–Raphson approach. Discretizing the PDEs using the finite-volume method allows representing the simulated area using a 3D structured grid of fixed volume cells. A two-phase FRS may consists of two components, which are the oil and water. The state of a grid cell is determined by two independent variables: the pressure (p) and the water saturation (s). The objective of the reservoir modeling and simulation is to provide approximate solutions for p and s at discrete time points for all grid cells over a given time frame. Computing the differences of the model function at the neighboring grid cells is based on one of several possible stencils. A standard approach for solving the flow variables is to apply a stencil relationship at each cell such that $s(t)$ at any cell depends on $p(t-1)$ and $s(t-1)$ for a set of neighboring grid cells. Each cell

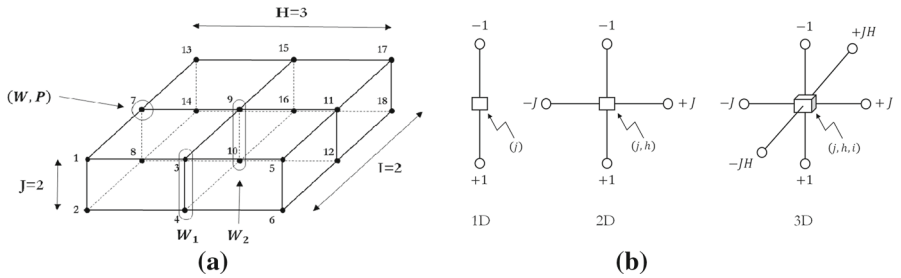

Fig. 2 Structure grid and stencil computations. **a** A $2 \times 3 \times 3$ structured grid with two wells, **b** stencil defined over 1D, 2D, and 3D grids

interacts with 2, 4, or 6 nearest neighbors depending on whether a 1-D, 2-D, or 3-D grid is used. Hence, the above grids are characterized by a three-point, five-point or a seven-point stencils, respectively. In general, oil may have k components, for which each grid cell has a set of k independent variables.

Figure 2a shows a two-phase 3-D FRS-structured grid as a set of $J \times H \times I$ cells. Each cell (j, h, i) is located at its three coordinates: (J) as the vertical axis, (H) as the horizontal axis, and (I) as the onward axis. The two-phase grid has 18 $(2 \times 3 \times 3)$ cells distributed over two plans, each has nine cells. Each cell is characterized by two components, which are w and p. The above grid can be represented by a vector U where each (j, h, i) cell is mapped to vector $U(m)$, where $m = j + J \times h + J \times H \times i$. The cell numbering shown in Fig. 2a illustrates this mapping. The unfolding process allows representing the stencil constraints for all the structured grid cells including all possible cell interactions by using a $m \times m$ matrix. Figure 2b shows the above stencils for the case of 1D, 2D, or 3D grids with the corresponding offsets between a central cell and its neighbors when the grid is unfolded into a 1D representation.

In addition, we may define a number of wells as part of the FRS model. Each well is represented by a set of cells arranged along the vertical axis. Figure 2a shows two wells $W1$ and $W2$, each consisting of two cells. The state at each well's cell can be determined using direct measurements. For this the well information must be taken into account in the FRS simulation.

5.2 Generalized sparse, blocked, hepta-diagonal matrix

Generally, FRS uses a 3D grid which is a set of $J \times H \times I$ cells, where each cell is located at its coordinate (j, h, i). In a k-phase process, the state of each cell is represented by k independent variables or residuals $(p, s, \ldots)$. Hence, the total number of grid state variables is $M = J \times H \times I \times k$. In general, cell (j, h, i) interacts with its six immediate neighbors which are located at: $(j-1, h, i)$, $(j+1, h, i)$, $(j, h-1, i)$, $(j, h+1, i)$, $(j, h, i-1)$, and $(j, h, i+1)$. A cell interacts with each of its neighbors by means of a block sub-matrix (B) of $k \times k$ variables. Bs represents a variational relationship or Jacobian matrix between the interacting cells. The solver matrix A is built to represent the stencil relations among all the grid cells, e.g., all the inter-cells blocks. Hence, the solver matrix is a block-based matrix that represents the Jacobian of cell interactions. Hence, the solver matrix has a size of M^2.

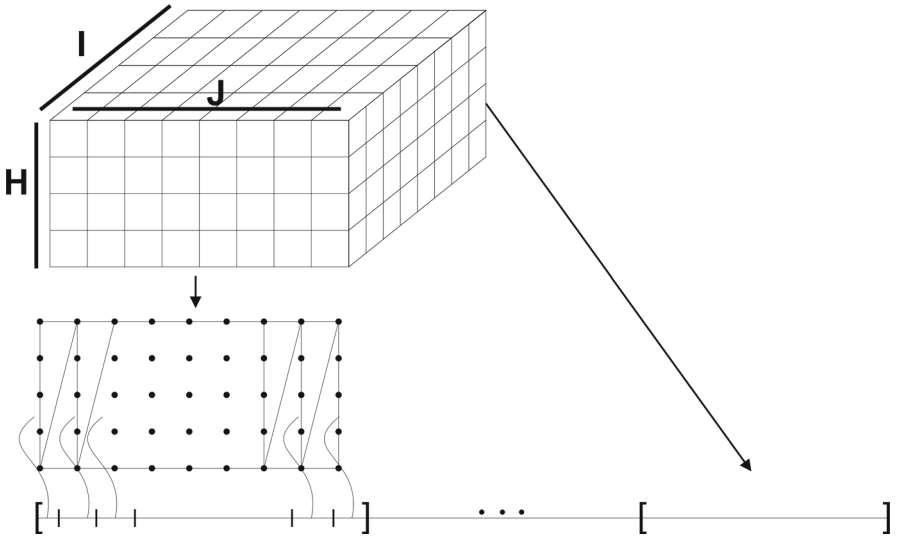

Fig. 3 A $5 \times 11 \times 8$ structured grid where cells unfolded into a vector representation

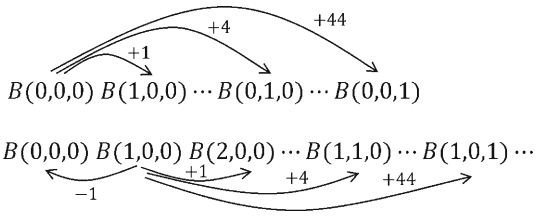

Fig. 4 The first and second *block-rows* of solver matrix with *block* offsets

Figure 3 shows a structured grid where $J \times H \times I$ consists of $4 \times 11 \times 8$ or 440 cells distributed over four plans each has 88 cells. The figure shows the unfolding of the grid cells and their mapped onto the 1D vector. Note that each cell is represented by k components which makes the vector size $M = J \times H \times I \times k$. Any pair of cells X and Y that are related by some stencil will be associated a B block of $k \times k$ elements originated at row X and column Y of matrix A.

Using a seven-point stencil, cell (j, h, i) located at $m = j + J \times h + J \times H \times i$ has seven NZ blocks centered at block $B(j, h, i)$. In other words, each row (j, h, i) of the solver matrix generally has seven blocks of NZs, of which three blocks are to left the central block $B(j, h, i)$ and three other blocks are to the right:

$$\begin{aligned} & B(j, h, i-1)..B(j, h-1, i) \\ & ..B(j-1, h, i)B(j, h, i)B(j+1, h, i) \\ & ..B(j, h+1, i)..B(j, h, i+1) \end{aligned} \tag{1}$$

where the offsets from left to right of the central block are $J \times H$, $-J$, -1, $+1$, $+J$, and $+J \times H$ blocks, respectively. Figure 4 shows the blocks corresponding to the first and second cells. Grid boundary conditions lead to truncate the negative offsets.

Recall the two-phase, $4 \times 11 \times 8$, 3-D structured grid. Figure 4 shows the first and second block-rows (first and second cells) of the solver matrix with offsets from the central block corresponding to the stencil relationship. The first and second block-row have four and five blocks due to stencil truncation at the grid boundary, respectively, where the block offsets are $-1, 1, 4$, and 44. Referring to the central block, the relative column position of the neighboring blocks in each row are the same in each row. Thus knowing the location of the central block in a row allows finding the column numbering of all the other neighboring blocks in that row.

In the case of 3-D grid, each row of the resulting matrix has 7-block of $k \times k$ NZ elements corresponding to the stencil relationship. The resulting matrix is (1) sparse as each row has only seven $k \times k$ blocks of NZs and (2) blocked, hepta-diagonal as the blocks with same offset to central are arranged along each of the seven matrix diagonal. Hence, the above matrix is called the *Hepta-diagonal (H) Matrix*, i.e., a blocked matrix with seven NZ block-diagonals.

The reader may refer to [38] for more detailed description on the PDEs of a reservoir simulation model that is jointly used with the finite volume discretization approach to derive a stencil relationship. The stencil is applied at each grid cell to produce a set of algebraic equations whose representation is compacted into a hepta-matrix.

5.3 Incorporating the knowledge at the wells

An oil reservoir represented by a structured grid having M cells, Nw wells, and k components is associated a solver matrix GH. GH has four sections, which are (1) cell–cell interaction is represented by a Mk $\times$ Mk block-based Hepta-diagonal sub-matrix H, (2) the well–cell interaction represented by Nw $\times$ Mk sub-matrix W, (3) the cell-well represented by Mk $\times$ Nw sub-matrix WT (by transposing W), and (4) a Nw $\times$ Nw diagonal matrix in the right bottom corner. The solver matrix consists of assembling the above four matrices into a $[M \times (k + \text{Nw})]^2$ solver matrix, which is called the *generalized hepta-diagonal (GH) Matrix*.

Figure 5a shows GH for the oil reservoir represented by the structured grid displayed in Fig. 2a with 18 grid cells ($M = 18$), two wells (Nw $= 2$), and each cell has two components ($k = 2$). The grid cells are arranged in linear order where a cell located at $m = (j, h, i) = j + 2 \times h + 6 \times i$. Since each cell has 2 components in the above grid, the blocked grid structure is shown using 2×2 blocks with the same color for each block. GH is a 38×38 matrix that consists of four parts: (1) H as a 36×36 sub-matrix, (2) W as a 2×36 sub-matrix, (3) WT as a 36×2 sub-matrix, and (4) a 2×2 diagonal sub-matrix. Each square cell represents one element, where a yellow cell represents a zero elements in GH. Cell $m = (j, h, i)$ is represented by its six neighboring blocks (stencil) in addition to the cell's central block which is colored in red. The six neighboring blocks that distributes on the left and right of each central cell are at offsets (in blocks) -6 (light green) and $+6$ (dark green), -2 (dark orange) and $+2$ (light orange), and -1 (light blue) and $+1$ (dark blue), respectively. The number of standard NZ elements in GH is 7JHI.

Each well is associated with an additional row in the solver matrix to account for the knowledge of the state at each well's cell. A well's row has NZ values only at the

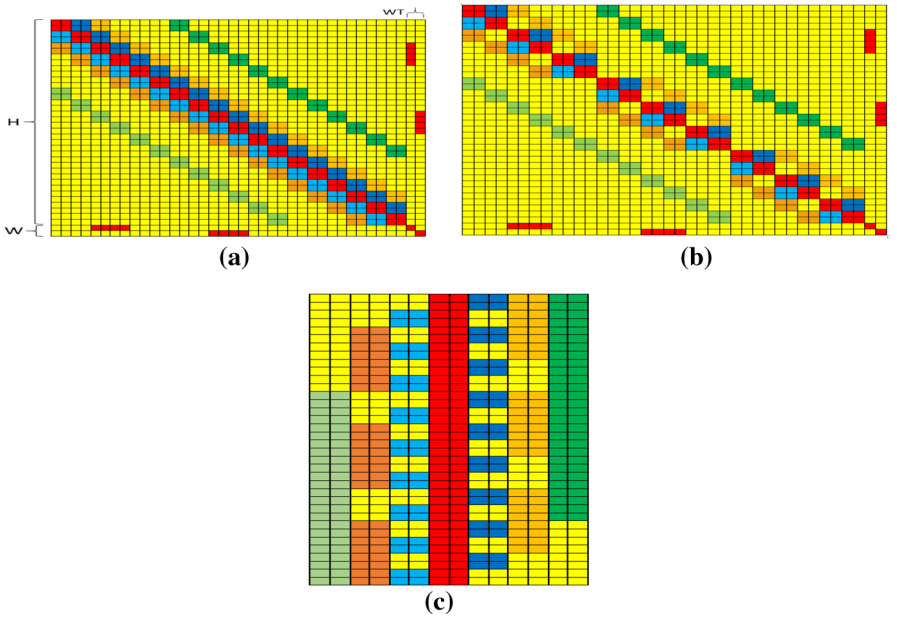

Fig. 5 Generalized Hepta (GH) representations. **a** For the FRS system with its sub-matrices H, W, Wt, and D, **b** after accounting for the stencil truncation conditions, **c** block-diagonal representation of the hepta-matrix

location of each of its cells. Grid cells must account for their intersection with the well's cells, which defines the sub-matrix W and its transpose WT.

The seven-point stencil is truncated when the stencil is applied to a cell located on each of the six grid sides. Accounting for the truncation at the generation of GH is implemented by checking whether the current cell belongs to one of the six grid plans. A neighboring block $B(x, y, z)$ for some central block $B(i, h, j)$ is truncated when one or more of its coordinates is located at any of the grid boundaries. The implication on GH is that many blocks become zero within the seven diagonal of each row $m = (j, h, i)$ satisfying any of the above conditions. Figure 5b shows an updated GH after accounting for the stencil truncation. The number of NZ elements in GH after accounting for the stencil truncation is $(7JHI - 2HJ - 2IJ - 2IH)k^2$. The ratio of NZs (or blocks) to overall number of blocks in GH is $R = (7JHI - 2HJ - 2IJ - 2IH)/(JHI)^2$ which is close to $7/JHI$ for a large grid. Hence, Storing GH is quite inefficient because the NZs occupies a very small fraction of GH storage.

Note that the data values of each NZ block is defined after the PDE model that governs the grid simulation and its associated initial conditions. These details are omitted from this presentation which focuses on the GH sparse pattern and regularity.

5.4 Exploiting the regularity in the hepta-matrix

The FRS computational algorithm consists of repeatedly (1) solving a large system of linear equations GH. $X = b$ to find the solution $X(t)$, (2) save $X(t)$ and update the

Jacobian GH and vector b as function of the new state $X(t)$ and FRS model. The above process repeats for a given simulation time. The saved solution $X(t)$ represents the cell states, which can be exploited to predicts how the oil and water are moving over the production time. The FRS computational performance depends to a large extent on optimizing GH storage as well as the computational efficiency of the sparse linear solver (LAS). The solver system GH. $X = b$ can be implemented by using a direct or an iterative approach. Example of a direct approach is the LU factorization with partial pivoting, which produces accurate solution. Accuracy is not a major requirement in the FRS Newton–Raphson linearization. In addition, direct methods generally have limited data parallelism and many global synchronization, which makes them less interesting for many-core implementation. On the other hand, Krylov INAAs like biconjugate gradient stabilized(BiCG-Stab) have interesting converging properties in addition to abundant data parallelism. It can be used to solve a large sparse system of linear equations, where the solver matrix is not necessarily symmetric or definite positive. It is well known that INAAs most of the computing time is spend on sparse matrix–vector multiply (SpMV) operations, which justifies the focus on storage optimization, operation on NZs, and the efficient use of memory hierarchy in view of the sparse data structure. BiCG-Stab repeatedly invokes SpMV, vector addition and scaling, and vector inner product. For this the sparse matrix storage scheme has significant impact on performance and storage requirements. For the above reasons, we focus on the following GPU optimizations:

1. Minimizing GH storage requirements by exploiting the regularity pattern of NZs and the blocked data structure in the Hepta-diagonal matrix H and the well matrices.
2. Optimizing the regeneration of the SpMV original indices for NZs in H, W, and WT for efficiently implementing the SpMV and all needed vector operations.
3. Maximizing bandwidth in retrieving the sparse matrix data from lower level of the memory (GM) by using memory coalescing and avoiding conflicts in shared memory (ShM).
4. The use of tiling technique to be able of handling very large matrices and vectors retrieval, storage, and arithmetic operations using relatively very small ShM at the compute units (SMs).
5. Optimize global synchronization across all computing thread blocks following each reduce operation (inner product).
6. Optimize the GPU resource occupancy using auto-tuning by searching for the best combination of some architectural parameters.

For the above reasons, the design of custom storage schemes and kernel optimization of the corresponding SpMV are two critical steps toward the design of efficient INAAs for parallel reservoir simulation.

6 Optimized GH storage and SPMV

Optimizing the storage of the generalized hepta-matrix GH for the SpMV operation consists of designing separate storages for the hepta-matrix H and the two well matrices W and WT due to their different data structures. A balanced storage computation

is needed for overall SpMV optimization for finding a minimal storage that incur the least arithmetic operations and avoid costly global operations on the GPU. In the following we present a methodology for optimizing H and the well matrices (W and WT) to balance storage and computation in SpMV.

6.1 Synthesizing optimized storage for the hepta-matrix H

The SpMV operation consists of computing GH. $X = V$, which can be seen as the inner product of a row of NZs by the corresponding elements from X. A compact representation of H is to allocate one block-row for each cell, in which the central block and its six standard neighboring blocks are stored. The result is a rectangular matrix with JHJk rows and $7k$ columns. Each block-row consists of $7 \times k^2$ blocks corresponding to a central cell and its six stencil related cells, where the column offset with respect to the central cell are -3, -2, -1, 1, 2, and 3. The above block column offsets map to the blocks at offset -6, -2, -1, 1, 2, and 6 in the H matrix. Hence, the synthesized storage of matrix H is a $7JHIk^2$ block-diagonal rectangular matrix called *Bdia* as each of the block-diagonal from H is converted into the block column format.

To generate an instance of Bdia storage, the main diagonal m is extracted by scanning the grid through iterating over the value of j, h, and i dimensions, respectively. This diagonal will be used as the main diagonal of Bdia storage format (i.e., offsets equals to 0) for all other rows.

The SpMV operation consists of computing $\text{Bdia}(H) \cdot X = V$. Hence, we need to find the original index of each NZ block. To simplify the presentation we describe the SpMV by referring to the mth row-block as a set of k consecutive rows (row mk to $(m + 1)k$), which corresponds to cell m. The mth central cell has its blocks located at the diagonal, i.e., the column offsets of the block corresponding to the central cell ($\text{BDIA}(m, m)$) is also m. The offsets with respect to the central cell of the neighboring blocks (NB) can be easily found because these blocks have the same regular distribution for each row. At the mth row-block, the column numbering (u) of the seven blocks must be in the set $U = [-3, -2, -1, 0$ (central block), 1, 2, 3] which maps to the blocks at columns $m + h(u)$ in H, where $h(u)$ is in the set Uh $= [-6, -2, -1, 0, 1, 2, 6]$. The function $h(u)$ is invariant for all row blocks due to the regularity of the distribution of the neighboring cells (stencil related cells) with respect to central cell. The SpMV operation mainly consists of computing $s+ = \text{Bdia}(m, u) * X(m + h(u))$ for each of the seven blocks, i.e., for the mth block-row and each value of u in U.

The total storage of Bdia is $JHIk^2$ and its number of NZs is $(7JHI - 2HJ - 2IJ - 2IH)(k^2)$. The ratio of NZs (or blocks) to overall block is $R = (7JHI - 2HJ - 2IJ - 2IH)/JHI$ which is close to 1 for grids that are large enough. For example, the ratio $R = 0.9732$ when $J = H = I = 32$, which is an indicator of the efficiency of Bdia storage scheme.

At runtime, a number of *thread blocks* (TBs) are assigned to each streaming multiprocessor (SM). SM breaks each TB into warps, each consists of a bundle of 32 threads which are run in SIMD mode. Hence, the mth SpMV result is assigned to the mth computing thread, i.e., the mth block of results from V is assigned to the mth

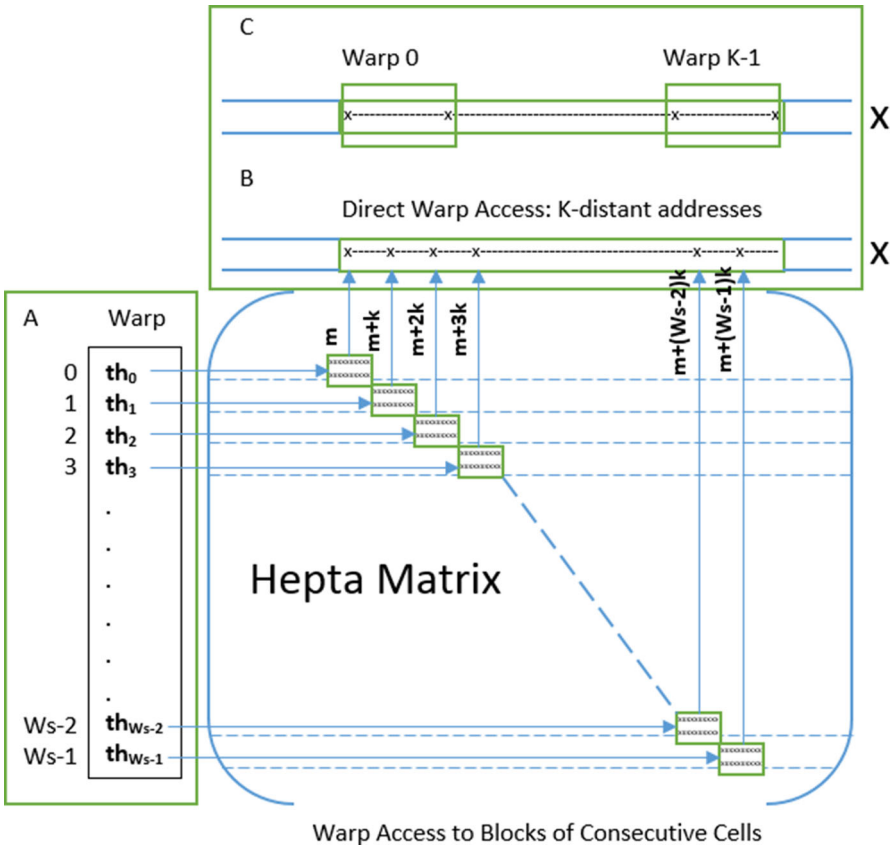

Fig. 6 Non-coalescent access due to k-distant addresses (B) and cooperative data pre-fetching prior to inner product loop (C)

thread th(m). Thread th(m) computes a block of results $V(m)$ as is the sum$s+=$ Bdia$(m,u)*X(m+h(u))$ for each value of u in U. Since thread th(m) accesses Bdia(m,u), the threads of each warp access a sub-column of Bdia blocks. Hence, coalesced access to GM is granted only if the matrix is stored according to column-major ordering as opposed to row-major ordering. Note that multi-dimensional arrays must necessarily be converted to 1D representation when stored in GM.

In addition, thread th(m) must also access $X(m+h(u))$ for each value of $h(u)$ in Uh. In the H matrix, the central block as well as all the neighboring blocks are shifted by one block column from one block-row to the next, i.e., a shift of k if one refers to the real ordering of elements. Hence, successive threads th(m) and th$(m+1)$ access elements $X(m+u)$ and $X(m+u+1)$ that are distant by k data elements when accessing elements of vector X. Figure 6 shows the access by a warp to blocks corresponding to a set of Ws successive grid cells with $k=3$, where Ws is the wrap size. The column addresses are distant by k. Hence, access to X by the above SpMV loop cannot be coalesced for arbitrary values of k as shown in Fig. 6 for the

vector X (left). The alternative is to let the threads of each warp cooperatively load, in coalesced fashion, the range of values from X that is needed to compute each block of results from V. This is shown in Fig. 6 for the vector X (right) which is accessed k times by a warp. Denote by th(m), th$(m + \text{Ws} - 1)$ are the threads of a warp, the range $R(m, u) = (\text{start}, \text{end})$ of X values to process a block must satisfy start $= m + h(u)$ and end $=$ start $+$ Ws $- 1$. Hence, processing each block by a warp requires cooperative thread loading from GM of $k *$ Ws values from vector X, storing the data in shared memory, and computing $s+ = \text{Bdia}(m, u) * \text{ShM}(m + h(u))$ by each warp thread for each block. Access to ShM is conflict-free because the consecutive threads in each warp access consecutive data elements from X. Note that for each warp, the range $R(m, u)$ of needed data from X to process a block may overlap with the data from X to process another block for small values of k. The overlap is avoided if loading the X values into ShM is done in three phases of contiguous data blocks, where the block offset are (-6), $(-2, -1, 0, 1, 2)$ and (6), e.g., when Ws $\leq 4k$. For Kepler Ws $= 16$, the above condition is satisfied when $k \geq 4$.

6.2 Synthesizing storage for the well matrices

In the following, we present our approach for the design of a storage for the two well matrices so that to minimize their storage as well as the overhead in retrieving and computing SpMV operations on these matrices.

Recall the well matrix W, GH includes two well matrices W and WT. Since the wells do not overlap in their locations, there is only one NZ in each column of W. An array Wa is used for storing the sequences of consecutive NZs of each well one after the other so that the sequence of a given well can be retrieved by indexing Wa with the well number w. Three arrays a, b and c are needed to store (1) $a(w)$ as the pointer to the first NZ in Wa, (2) $b(w)$ as the column order of $a(w)$ and (3) $c(w)$ as the number of consecutive NZs for well w. Hence, computing the inner product (s) of a well row by a vector X requires indexing Wa to retrieve the well NZs and their orders. Each well row requires reducing $s+ = \text{Wa}[a(w) + i] \times X[b[w] + i]$ for $i \leq c(w)$. Based on the above data structure, retrieving each element requires two integer operations. The total storage of W requires Nw $\times (k + 3)$ words.

Minimizing the storage of WT is a different from the above as its data structure is the transpose of W but with different data values. Recall that each grid cell (j, h, i) maps to 1D representation at offset $m = j + h \times J + i \times J \times H$. The access of WT is needed for the mth thread that is computing the inner product of $[\text{row}(H), t]$ by a vector X, where row(H) is the mth row of H and t is the NZs in the mth row of WT (if any). Note that $t = 0$ in most cases. In the case, t is NZ, the problem is to minimize the retrieval of its value only when computing the stencil at the mth cell, which depends (neighborhood) on a well cell. The problem is to retrieve an NZ (if any) from row m of WT and to find its column order. The wells are distributed over the upper grid plan (H, I) with some minimum distance between each other. Hence, the plan (H, I) can be partition into a grid, each grid square may contains at most one well. Given $m = F(j, h, i)$, the mth thread computes a pointer ptr $= (h/u, i/v)$ which points to

a well partition number, where u and v are the number of partitions along the H and I axes, respectively. Practically, the range of values for ptr is relatively small because an oil field cannot have no more than 128 wells. This help finding the values of u and v. Given m, the potential well number is obtained by indexing an array (d) such that $d(\text{ptr}) = w$ or zero, e.g., identify a well or there is no well. This indicates that cell m falls into the well partition w. Now, term t is computed as $\text{Wa}[a(w) + j] \times X[M + w]$ if $a(w)$ is NZ, where $(a(w) + j)$ points to the NZ corresponding to the jth cell of well $a(w)$ and $V[M + w]$ points to the corresponding element in vector X.

In conclusion, we note the following observations: (1) both sub-matrices W and WT use the same compact storage (Wa) for all NZ data and (2) finding whether there is one NZ in row m requires two integer operations for calculating a pointer value and two integer additions for indexing.

6.3 Customizing the solver implementation

In most linear algebra solvers, the arithmetic operators used are the vector inner product, vector addition and vector scaling, and sparse matrix–vector multiply (SpMV). In most INAAs like the BiCG, completing thread blocks must ensure all other blocks have completed current iteration before starting the next iteration due to data dependency. Each row operation produces some result that is used in the subsequent row operations. In the BiCG-Stab (see Algorithm 1), the thread blocks cooperatively compute some intermediate vector data in each iteration that is used as input operand for the next iteration. Exact implementation of the solver algorithm requires no TB should start the next iteration until all other TBs have completed and a global solution is ready in GM.

Algorithm 1 The BiCG-Stab Solver

1: $x =$ some initial guess
2: $r = b - Ax$
3: $\hat{r}$ = r or a guess ($\hat{r}^t.r \neq 0$)
4: $\rho = \omega = \alpha = 1$
5: $v = p = 0$
6: **while** $r.r \geq \epsilon$ **do**
7: $\quad \rho = r_0.r$
8: $\quad \beta = \frac{\rho.\alpha}{\rho_p.\omega}$
9: $\quad p = r + \beta(p - \omega.v)$
10: $\quad v = A.p$
11: $\quad \alpha = \frac{\rho}{r_0.v}$
12: $\quad s = r - \alpha.v$
13: $\quad t = A.s$
14: $\quad \omega = \frac{s.t}{t.t}$
15: $\quad x = x + \alpha p + \omega s$
16: $\quad r = s - \omega.t; res = r.r$
17: **end while**

The BiCG-Stab algorithm has five inner products and requires two cooperative vector saving on GM, which requires seven inter-block synchronization. CUDA does

not provide a global synchronization scheme other than the *Kernel Exit and Re-entry* (KER). Unfortunately, implementing the above algorithm using standard KER global synchronization is responsible for significant performance degradation due to the loss of any thread block data locality. Similarly, implementing the solver using library calls leads to use library operators as separate function calls which implicitly use the KER scheme for each operator. Our approach is based on enhancing the implementation of INAAs by improving spatial and temporal locality of data accesses in code regions. In the following we discuss the need for some INAAs domain-specific optimizations:

1. *Minimizing data transfer overhead between CPU and GPU* All data read/written on the device must be copied to/from the device (over the PCIe bus). This is very expensive and keep data resident on device. This may involve porting more routines to device, even if they are not computationally expensive.
2. *In-kernel iterative algorithm implementation* KER transfers control from the device back to CPU by interacting through the PCIe. KER is sometimes a simple solution for inter-block synchronization, which is needed for a reduction or a collective writes to GM. In GPUs, data cached by a kernel cannot be used in a subsequent kernel launch because the TBs are dynamically assigned to SMs and data locality is lost at kernel exit. This leads to additional overhead for saving the data back into GM and reloading it again after kernel re-entry. A thread reference to some data requires a load instruction that reads GM and store into ShM, loads data into a register prior to use, and stores instruction to write back the register data into ShM. L1 cache is accessed while executing the load instruction. However, once data are stored into ShM, L1 cache is no more accessed for the data. Instead of the KER, we propose an in-kernel approach using a custom inter-block synchronization, called reordered synchronization (ROS), to avoid the overhead associated with multiple KERs and the loss of data locality. ROS enables staying in kernel without increasing the synchronization overheads as compared to KER. Also, we may combine reductions and/or collective writes to further reduce the number of needed ROS, if the algorithm allow.
3. *Use of shared memory in the case of sufficient data reuse* Data load or store on GM or texture memory causes few hundreds cycles (400-cycle for K20) long latency operation, while ALU operations, branch resolution, or accesses to ShM are short-latency events taking from 8 to 20 cycles. GPU puts the burden on application programmers to explicitly manage the usage of shared and global memory. To use ShM, we first must load data from GM to ShM, which is a long latency operation. Next, every data invocation requires moving the data from ShM to the specific SP lane register, i.e., close to the execution unit. If there is not enough reuse of data that consumes more time and energy than simply loading data directly from GM into register file (RF). Here, compiler stores short-lived working data in faster storages such as the RF, which is close to the execution units and recycle the registers if data is not referenced any more. Actually, data must be stored in RF just before its reference to avoid power consumption due to long latency register context switching. Effectively leveraging ShM requires increasing kernel and thread grain size by combining operations, whenever possible. Short kernels and threads must have realistic memory usage using direct data load from GM into RF. Therefore,

we must load data from GM into ShM if there is sufficient data reuse, otherwise, data are loaded directly into RF.

4. *Maximizing memory bandwidth* The memory bandwidth for the graphics memory on the GPU is high compared to the CPU, but there are many data-hungry cores, so memory bandwidth is still a performance bottleneck. The maximum bandwidth is achieved when data is loaded for multiple threads (warp) in a single transaction, e.g., memory coalescing. This will happen when data access patterns meet certain conditions: 16 consecutive threads (i.e., a half-warp) must access data from within the same memory segment. This condition is met when consecutive threads read consecutive memory addresses within a warp. If the condition is not met, memory accesses are serialized, which significantly degrades performance.
5. *Auto-tuning to optimize GPU occupancy* When programming for the GPU architecture, the programmer maps each loop iteration to a thread. Obviously, there must be at least as many total threads as cores; otherwise, cores will be left idle. For best performance, the number of threads must be much greater than the number of cores. Accesses to GM have several hundred cycles of latency when a thread stalls waiting for data, another thread can switch in this time to hide latency. Hence, one GPU feature is the very fast thread switching and the support for many concurrent threads. Different GPU architectures and vendors have different specifications on the occupancy parameters like SMs per GPU, grid size, thread block size, thread grain size, and some compiler flags. The resources must be shared between threads because high use of on-chip memory and registers will limit the number of concurrent threads. Typically, it is best to use many thousands of threads in total. The optimal number of threads per block should be found using auto-tuning which consists of running a parametric kernel for a few relevant combination of architectural parameters and select the one that sustain the best performance.

Efficiently implementing in-kernel iterative algorithm requires redesigning the inter-block synchronization mechanism to avoid exit of the kernel and the loss of data locality. For this we shortly review the major synchronization schemes [3,29,35]. The lock-based synchronization uses atomic operations on global variables defined in the global memory. When all threads of a block finish their work, the first thread of each block atomically decrements a global variable and continues checking as long as nonzero. The drawback is the hot spot in polling of GM by terminating block. In lock free, each terminating block b sets its entry in a global input array Ain(b) to post its termination. Next, the thread checks the completion of other blocks using other block locations of Ain. Note that access to Ain(b) needs not be atomic because Ain(b) can be set only by block b. The barrier is passed when a block finds that all entry of Ain are set.

Relaxed synchronization [20] allows two iterations to overlap in time. A completing thread block stores its range of results and may start the next iteration by using others partial results. Completing thread block post their completing results using a global array, where one entry is allocated to each block. The block terminates the current iteration when it has processed all partial results in the current operation. A two entry circular buffer is used to store the block partial results because at most two iterations can overlap their execution at any time.

In the following, we describe a scheme called reordered synchronization (ROS) which consists of scheduling the operations so that a TB completing its part of work sharing with synchronization (WS-S) starts an independent work sharing (WS) operation to avoid polling GM while waiting for the completion of other TBs.

6.4 Work sharing with synchronization

To optimize the implementation of linear algebra solvers we propose a work sharing with synchronization (WS-S) construct which is implemented on the top of our ROS, which provides the needed inter-block synchronization (IBS). In the following we describe the WS-S and present an example of using it to optimize the implementation of the BiCG-Stab solver.

A WS-S construct has two sections WS-1 and WS-2. WS-1 consists of a compound of statements that are data parallel which terminates by at least one reduction (R). WS-2 is a compound of computations that must be done anyway and being data independent from R. Note that WS-2 is processed in the context of a WS-S in parallel with the current IBS period. WS-2 avoids a completing TB from polling GM for the completion of other TBs and postpone the polling until the latest time R result is imperatively needed by some other data dependent computation. WS-1 allows merging multiple data parallel operations and must terminates by one or more inner products. Hence, multiple reductions can be combined in one single WS-S structure to save IBS overheads. One important utility of WS-1 is combining the multiple vector operations by passing data from one operation to the next through the shared memory (ShM) of each SM, i.e., most vector producer-consumer relationships are satisfied within ShM. For example, the WS-S (line 15) has one scalar computation, three data parallel vector operations, and two inner products requiring one single global reductions using WS-S.

On completion of the parallel section, the kth TB carries out a combined tree-reduction (intra-block) to find the block partial result for each reduction, report to GM all the partial results and set up one single completion (Ain[k]). If all the blocks have already terminated, the inter-block partial results are reduced, saved, and a tagged check indication (Aout[i] for each TBi) is set in GM, otherwise WS-2 is started. Since TBs process the reordered computations in WS-2 instead of polling the completion of the reductions, it is most likely that on the completion of WS-2 all the TBs have completed their WS-1 sections and the results from the global reductions are ready in GM. On completion of WS-2, the kth TB checks the completion of the combined reduction (Aout[k] $= 1$) which enables exiting the current WS-S construct, i.e., the global reduction results are ready for use in subsequent computation. Otherwise, TB must wait until Aout[k] is set before exiting.

In some cases, the solver algorithm does not allow finding an independent computation that can be reordered in the WS-2 section. It is highly preferable to let TB waits without polling on GM while WS-1 are still in progress by the TBs. The delay depends on the degree of load unbalancing of the TBs which is functions of the thread load since the last global synchronization. The proper delay is found using auto-tuning which runs the algorithm for a small set of possible delays and retain the delay value that gives the best results for a given problem size.

There are many vectors whose computation consists of data parallel operations over a fixed vector range for each given TB. These vectors are called *range vectors* (RVs). For example, the statement sh_p = sh_r + β × sh_v denotes the TB operation over a range of the vectors. where p, r, and v are three RVs that are maintained in ShM and need not be saved on GM, except in the case of kernel exit, because their range is fixed for each given TB. Hence, TBs maintain RVs in ShM. Example of RVs are r, $\hat{r}$, p, s, and x.

Only vector and scalar results that are shared among all the TBs are reported to GM such as p and s, e.g., shared vectors (SVs). Usually, SVs intervenes as operand vectors in the SpMV function. SVs must be written to GM before the start of the next operation. SV vectors are cooperatively computed, but needed by all TBs in each solver iteration. Hence, the need for synchronization to make sure that all TBs have completed their update of SVs before starting the next operation. Some vectors can be both RV and SV.

Algorithm 2 Optimized BiCG-Stab Solver using Compact Custom Matrix Storage and Synchronization

```
1: x = some initial guess
2: r = b − SpMV(Bdia, x)
3: r̂ = r or a guess (r̂ × r ≠ 0)
4: ρ = ω = α = 1
5: v = p = 0
6: WS-S{ρ = reduce(cm_r0, sh_r)};
7:        {sh_v = sh_p − ω × sh_v};
8: Loop: WS-S{β = ρ × α/ρ̂ × ω;
9:          sh_p = sh_r + β × sh_v;
10:           p = sh_p};
11:           {tune_delay};
12:         WS-S{sh_v = SpMV(Bdia, sh_p&p);
13:           v = reduce(cm_r0, sh_v)};
14:           {tune_delay};
15:         WS-S{α = ρ/v;
16:           sh_s = sh_r − α × sh_v;
17:           s = sh_s;
18:           sh_t = SpMV(Bdia, sh_s&s);
19:           u = reduce(sh_t, sh_t);
20:           v = reduce(sh_s, sh_t)};
21:           {tune_delay};
22:         WS-S{ω = u/v;
23:           sh_x+ = α × sh_p + ω × sh_s;
24:           sh_r = sh_s − ω × sh_t;
25:           error = reduce(sh_r, sh_r);
26:           ρ̂ = ρ;
27:           ρ = reduce(cm_r0, sh_r)};
28:           {sh_v = sh_p − ω × sh_v};
29:         if  error > ϵ  then  Loop
```

The advantages of the above WS-S construct over traditional KER approaches are the following:

1. *Reordered synchronization* There are two ROS advantages. First, the synchronization time due to different WS-1 completion overlap with WS-2 compound that must be computed anyway or wait for some adjusted time delay. Second, the use of vectorized notifications allow eliminating the need for atomic access in the case of global mutex, which significantly reduces the inter-block synchronization overheads. TBs may check in parallel different vector inputs in any order. Thus, ROS saves the polling bandwidth needed for GS.
2. *Combining global synchronization* Normally a global synchronization (GS) is needed for each inner product (reduction) and after saving each global data. The later is needed for making sure that all TBs completed reporting SVs to GM. The use of WS-S constructs seems to be convenient in combining GSs, e.g., implementing a minimum number of GSs within each iteration. The BiCG-Stab algorithm has five inner products (lines 13, 19, 20, 25, and 27) and two SV saving on GM (lines 10 and 17) which requires seven inter-block synchronization. These are normally implemented using either seven kernel exit and re-entries for each iteration or using lock-based or lock-free synchronization. However, the use of our proposed WS-S construct allowed implementing the BiCG-Stab main loop using only four WS-S structures as shown on Fig. 2 (lines 8, 12, 15, 22).
3. *Fused algebra operators* Linear algebra libraries have optimized code for their arithmetic operators, but have the drawback of implementing each operator as a separate kernel. For this the operator output data is stored back in global memory before kernel exit. This forces the next operator to reload its operands back into ShM. Our BiCG-Stab implementation use fuse WS successive operations by chaining them one after the other and storing the outcome in ShM to improve data locality, reduce bandwidth utilization and code size. Fusing WS operations is a clear advantage over the use of library calls which leave operator output in global memory.
4. *Auto-tuning* The use of WS-S requires the use of auto-tuning to optimize the computing occupancy as well as adjusting the algorithm data locality to ShM capacity. For example, ShM needs to be large enough to store the SVs for all the TBs that can be running or pending on each SM. In the case of insufficient capacity, some SVs must be stored back into GM and loaded into ShM before use, e.g., swapping between GM and ShM. As ShM is different from one GPU to another, there is need run a few parametric code scenarios and select the one that sustain the best performance.

6.5 A tool for customizing the solution for structured grid

In the previous subsections, we have presented a general methodology to customize the design of the solver algorithm for reservoir simulation (FRS) for arbitrary grid size, number of components, and boundary conditions. To improve the portability of the above work we have generalized the above methodology for code customization by developing a structured grid development tool (SGDT). SGDT takes advantage of the regularity in the data structure and the algorithm properties in developing optimized GPU code for the SpMV and linear algebra solver. Currently, we experienced two

iterative solvers, which are the BiCG-Stab and QMR. In the following, we summarize the features of SGDT:

1. *Generalized Hepta-matrix* Deriving the generalized hepta-matrix GH based on the knowledge of the structured grid (J,H,I) all together with a set of cell components, stencil relationships, initial and boundary conditions, and FRS model for updating the data blocks following each solver iteration. This enables the automatic generation for a family of GH matrices, which depends on reservoir grid dimension and size, components, wells, stencil, boundary, etc. This module takes advantage of the regularity of GH at the block level described above.
2. *Optimizing storage and computation for SpMV* Decomposing GH it into submatrices with similar data pattern, separately optimizing the storage of each submatrix, memory access and indexing operations, which are the prerequisite for the design of an optimized SpMV CUDA Kernel. This module applies the optimization methodology described in previous subsections over instances that differ in the number of grid dimension and number of cells, number of components, stencil structures, location and number of wells, etc. The outcome is a parametric SpMV CUDA code which will be optimized using an auto-tuning approach. Auto-tuning searches for a combination of GPU architectural parameters (grid, thread blocks, thread granule size, and other compiler flags) for final optimization of the SpMV CUDA code.
3. *Optimizing the solver* Given an iterative solver expressed using vector, matrix, and scalar operations. Building a dependence graph, identify global synchronization points, implement optimized operators like SpMV, inner product, vector addition and scaling, and conditional for convergence detection and solution refinement. Alternatively, the user may choose to implement the vector and matrix operators by using library calls in addition to the proposed inter-block synchronization, which is customized for each global operation that must be done by all the threads. The addressed computing features are common in most iterative procedures for solving large systems of linear equations. This module uses the optimization methodology previously described for deriving GH Bdia storage, SpMV, and parallel BiCG-Stab solver. Similarly, auto-tuning is carried out over a parametric solver code as a final optimization step.

6.6 Application to quasi-minimal residual solver

Some of the well-known Krylov subspace methods are the quasi-minimal residual (QMR) [33] and the transpose-free QMR (TFQMR). It is known that BiCG can be erratic in practice where the residual norm may increase for several iterations. QMR is designed to smooth out this problem and makes progress toward the solution or at worst stall when BiCG temporarily diverges. However, QMR takes many more iterations to converge as compared to BiCG-Stab.

QMR algorithm [37] involves operators such as matrix–vector multiplication (SpMV), vector–vector multiplication (VM), and scaling (SV) or copying vectors. The main loop includes two SpMV, five VMs, and nine AXPY with seven SV operations. Implementing QMR on GPU reveals the need for nine inner products, five vector

saving on GM and two convergence conditions. The use of proposed WS-S construct allowed implementing QMR main loop using only nine WS-S structures as one single kernel.

In the next section we evaluate performance of optimized code generated by the proposed tool and compare its performance to similar solvers implemented using some of the most optimized numerical libraries like cuBLAS and cuSparse. We show that customized sparse solver solutions outperform those designed using standard sparse storage formats both at the SpMV and solver levels.

7 Performance evaluation

In this section, we present the performance evaluation of the proposed optimized kernels. Using our proposed tool, we generate test solver matrices (generalized Hepta-matrix GH) with main sparse Hepta-diagonal pattern and a distribution of tens of wells that arise in simulating multi-phase oil reservoir fields. The system is discretized using finite volume and simulated using a structured grid with multiple components. We evaluate performance of the synthesized solver kernel that refers to the customized GH storage scheme, SpMV on GH, and the BiCG-Stab solver algorithm on the Kepler family of GPUs. We compare the performance of the above kernels with different standard sparse matrix storage formats (CSR, BSR, and HYB) available in the cuSparse optimized library. DIA storage scheme is not available in the above library, we manually implemented it because it is relevant for this study.

The experiments are run on Tesla K20Xm hosted by an Intel Core i7 CPU. Table 1 shows the details of the main features of the GPU where the implementations are run.

7.1 Scalability of SpMV using custom storage Bdia for GH

We present the performance of our optimized SpMV Kernel that consists of a matrix–vector multiply ($\mathrm{GH} \cdot X = V$), where GH is an $N \times N$ generalized hepta-matrix that is stored using our proposed Bdia storage and V is an $N \times 1$ dense vector. GH matrices correspond to a grid of $J \times H \times I$ cells, each cell has k components, and there are Nw randomly distributed wells. Here we study the proposed SpMV implementation, where the sparse matrix is stored using the proposed Bdia scheme with JHIk rows and $7k$ columns. The SpMV scalability and performance is assessed versus the number of NZs ($NZ = (7JHI - 2HJ - 2HI - 2JI)K^2$) in Bdia matrix of size $S = 7JHIk^2$. Evaluation is carried out for $k = 2$ and $\mathrm{Nw} = 50$. Figure 7 shows the execution time (ET) and Flops achieved by SpMV. Figure 7a, b show the execution time when the number of arithmetic operations (Na) is within the range $u = [2.8 \times 10^3, 2.5 \times 10^6]$ and in $v = [2.8 \times 10^6, 1.14 \times 10^8]$, respectively.

For relatively small matrix size (first quarter of u range) SpMV time is nearly constant because the number of arithmetic operations is not large enough to amortize the overhead of loading and indexing Bdia data structure. In this case, the ratio of data transfer time from lower level memory over the arithmetic operation time is relatively large. One could improve performance by increasing thread grain size to customize for small matrices.

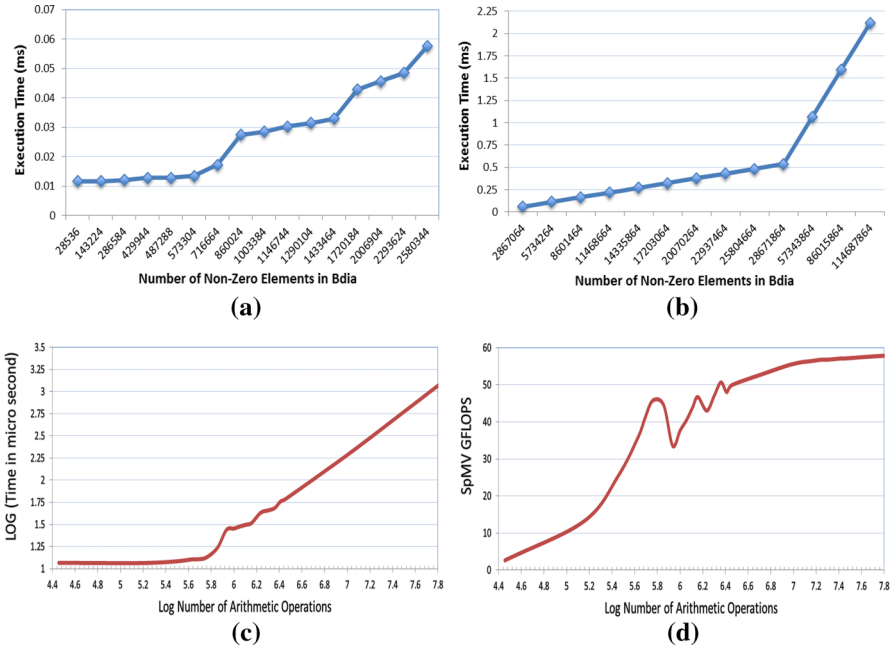

Fig. 7 SpMV execution times and FLOPS. **a** SpMV time for GH size of 10^3 to 10^5, **b** SpMV time for GH size of 10^5 to 4×10^6, **c** SpMV Time versus the number of arithmetic operations, **d** SpMV FLOPS versus the number of arithmetic operations

Figure 7c shows the plot of the SpMV time versus the number of arithmetic operations carried out by Bdia. SpMV timescales linearly versus Na $\leq 7 \times 10^5$ up to the largest possible value of Na which corresponds to the largest NZ (57×10^6) that the GPU memory can support. Since Na is proportional to NZ in Bdia, Fig. 7c confirms the linear scalability of SpMV time versus the grid size and the number of components k. Hence, SpMV timescales linearly with the grid size for large grids starting from a cubic grid $(23.4)^3$ which leads to Na $= 7 \times 10^5$.

Figure 7d shows the plot of the SpMV flops performance versus the number of arithmetic operations carried out by Bdia. SpMV shows its best performance for large Bdia matrices with up to about 60 GFLOPS. This confirms the ability of proposed automatically generated SpMV kernels to sustain high flops performance. SpMV time scalability and flops performance are two important indicators of the effectiveness of the proposed storage scheme and SpMV data structure indexing in minimizing the solver matrix storage as well as sustaining high performance.

7.2 Comparing Bdia to different library implementation using CSR, BSR, HYB, and DIA storages

In this subsection, we compare the performance of proposed SpMV to that implemented by using some standard sparse matrix storage schemes, which are available in numerical libraries. We implemented SpMV for generalized hepta GH using cuSparse

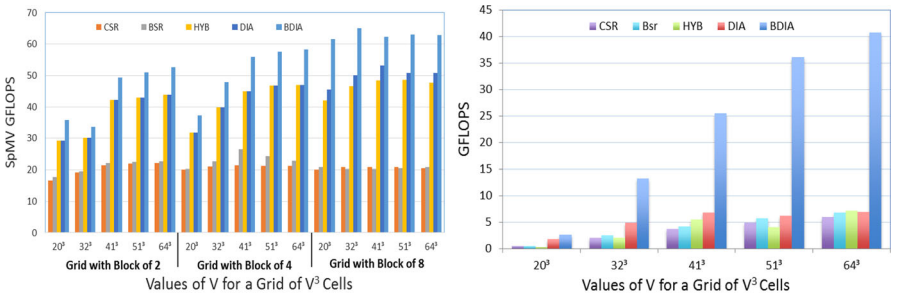

Fig. 8 SpMV performance for GH. **a** With a number of cells ranging from 8×10^3 to 2.6×10^5 and varying the block size as 2×2, 4×4 and 8×8, **b** for 2×2 skewed half-blocks

and cuBLAS, which are two of the most optimized numerical libraries for GPU for INAAs. For this, SpMV is implemented by storing GH using CSR, BSR and HYB library storage schemes. Note that BSR is adapted to the block size used, i.e., we used BSRk when using $k \times k$ blocks. We have written a separate kernel for coding the DIA storage format because it is not available in the above libraries. To compare with the library based implementation, SpMV algebra operators are implemented by calling the corresponding functions in the above libraries.

In our FRS-structured grid, there are two different types of hepta matrices in which successive rows differ by a block bias (GH-b) or alternating between a block and a half-block bias (GH-h) from one row to the next. This effect is due to the nature of the components and the possibility of state changes. GH-b has a regular block pattern, while GH-h has an alternating block bias from one row to the next.

We considered cubic structured grid of size V^3 for V in (20, 32, 41, 51, and 64), where the number of cells ranges from 8×10^3 to 2.6×10^5. The cubic representation is used to ease comparison to others. Using our SGDT tool we generated GH solver matrices for all of the above grids and varied the number of components as 2×2, 4×4, or 8×8 block sub-matrix. Hence, the corresponding Bdia matrix has from 2.24×10^5 to 1.17×10^8 NZs, i.e., from 3.2×10^4 to 1.7×10^7 NZs for the hepta-matrix. This Bdia size is the largest storage that can be handled in the working GPU.

Figure 8a shows the Flops performance for running SpMV for the above solver GH-b matrices using our proposed Bdia storage and the above storage schemes. CSR and BSR show close performance level, which is ranked last compared to the others. It is clear that the compressed row format is inadequate to exploit the regularity of GH-B along the diagonals. HYB and DIA show noticeable improvement over CSR and BSR due to their hybrid structure that take advantage of the diagonal structure in GH. Bdia is performing the best as it is faster by 136–225% than CSR and BSR and 14–30% than HYB and DIA, respectively. We also notice the lack of scalability SpMV using CSR, BSR, HYB, and DIA storages as their performance shows no improvement versus an increase in grid size. On the other hand, SpMV with Bdia scales up its performance by up to 50% for larger grid and block sizes for the above experiments. This indicates the effectiveness of Bdia storage in exploiting the regularity along the diagonal as well as the blocked data structure and its variable block size.

We also generated similar test cases for GH-h with its alternating half-block bias structure to study the effects of increasing the complexity in GH data structure. Figure 8b shows the SpMV FLOPS for which Bdia storage is used for GH-h for the studied structured grid sizes with $k = 2$.

The increasing irregularity in GH-h increases the indexing overhead for all the studied storage schemes especially for small grid sizes. Figure 8b shows that using CSR, BSR, HYB, and DIA for storing GH-h leads to a significant drop in SpMV flops performance (drop up to quarter) compared to the case of SpMV with GH-b. The use of these storages slow down SpMV by at least 68% as compared to the case of GH-b. SpMV with GH-h with Bdia is slower by up to 57% for small grids but its scalability scales reduces the performance drop to about 32% for larger grids, i.e., for $V \geq 32$. Bdia storage scheme has a noticeable performance gain over the studied storages because its design is adapted to three important structural factors, which are the data layout of the hepta-matrix data structure, the blocked pattern, and the stencil regularity in processing the large majority of the cells.

To compute SpMV, a row of data of $14k$ NZs is loaded from GM and used to compute a result using $28k$ operations, where k is always below 16. Although the storage is optimized, SpMV inherently has important overhead component due to its runtime indexing and low data reuse. The time to load data from GM into ShM is ten fold the time for arithmetic operations on the same data [19]. In SpMV, data are loaded once and involved in two operations only.

7.3 Scalability of BiCG-Stab using SpMV and Bdia storage

Our proposed optimization tool SGDT takes as input the BiCG-Stab algorithm (Algorithm 2) for solving $\mathrm{GH} \times X = b$ and produce CUDA kernel by (1) replacing GH with Bdia storage, (2) implementing our SpMV(Bdia), and (3) restructuring the operations to implement our enhanced global synchronization.

In this subsection, we present the evaluation for our automatically generated solver algorithm BiCG-Stab. Evaluation is based on using the previous structured grid of size V^3 for V in (20, 32, 41, 51, and 64). Using our SGDT tool we implemented BiCG-Stab algorithm using the proposed SpMV and the generated GH matrices for each of the above grid configurations. The GH block structure was varied as 2×2, 4×4, 8×8 for the case of 2, 4, and 8 components per cell. The above represent the implementation with the largest storage that can be handled.

Figure 9a shows the FLOPS performance for running our optimized BiCG-Stab algorithm using our proposed Bdia(GH) storage for hepta-diagonal matrices. We compare performance to the implementation of BiCG-Stab using CSR, BSR, HYB, and DIA storages, where the algebra operators are implemented using CuSPARSE library calls, which is considered among the most optimized numerical library for sparse matrix computing. The execution time of BiCG-Stab is dominated by SpMV (82.02%) in addition to some small fractions for the inner product (4.22%) and Axpy (8.67%), which consists of vector addition and scaling. Similar to the SpMV performance, our BiCG-Stab implementation is up to 23% faster than that using HYB or DIA storages and up to 160% faster than that using CSR or BSR storages with CuSPARSE library

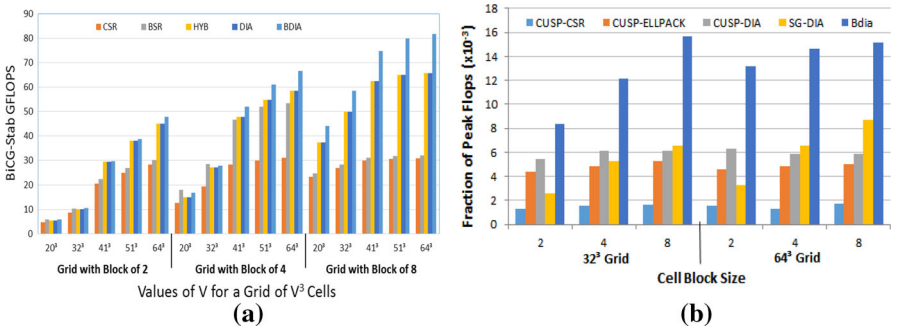

Fig. 9 BiCG-Stab performance for GH. **a** Using Bdia storage for grid size of 10^3–10^5 cells and blocks of 2×2, 4×4, 8×8, **b** comparing obtained performance to others

calls. Performance is most effective for large structured grid sizes. Like in the case of the SpMV, the proposed BiCG-Stab implementation has scalable performance as its flops performance linearly increases with the number of cells. Although most of the BiCG-Stab time (82%) is spent on SpMV, it achieves higher flops performance than SpMV because most of its other operations are on data parallel vector operations (range vectors) in addition to other combined operations requiring inter-block synchronization like the inner product and collective write to GM.

7.4 Performance of optimized QMR solver

QMR has been implemented using GH and BDIA storage, optimized SpMV and the work sharing construct with its optimized synchronization. Evaluation is performed using the previously defined cubic structured grid sizes to enable comparison with BiCG-Stab. Evaluation shows that the number of operations for QMR and the execution time are nearly the double of those obtained for BiCG-Stab with the same problem sizes. Note that SpMV operation dominates the execution time for both solvers. The main trends also show scalable execution time performance versus an increase in the number of grid cells and cell block size. This shows the portability of proposed optimizations to a class of linear algebra solvers that have similar structures and convergence conditions and mainly differ by the required number of matrix and vector operations and the number of global synchronization for each iteration.

8 Related work

To implement an effective forward reservoir simulation (FRS) on GPUs, we proposed an optimized implementation of the BiCG-Stab algorithm for solving $\mathrm{GH} \times X = b$, where GH is a class of sparse Hepta-diagonal matrices including cell–cell, cell–well, and well–cell sub-matrices. The solver performance strongly depends on the solver matrix storage scheme, thus we proposed an optimized storage (Bdia) for the class of GH matrices. Bdia provides different storage optimizations for its three basic data

structures. Furthermore, Bdia has been used in optimizing the SpMV implementation both in memory storage and execution time.

BiCG-Stab is implemented as one single CUDA kernel for which the threads internally iterate until convergence of the solver. The benefits of this approach can be summarized by: (1) avoiding costly kernel exit-re-entry, (2) preserving data locality for range vectors, (3) saving memory bandwidth and (4) enhancing reuse of cached data in ShM. This has a clear advantage over implementations that use a separate kernel for each (1) library call, (2) inner product, or (3) collective write to GM. To avoid library calls we developed optimized linear operators like SpMV, inner product, vector add, and scaling. Inner products and collective writes to GM are combined and implemented using a custom reordered synchronization, which avoids polling GM by scheduling some work that must be done anyway or inserting tuned delays if no independent work can be found.

The PETSc library tools are used for automatically generating optimized SpMV kernel which is auto-tuned using OrCUDA toolkit [22]. An optimized storage scheme is proposed for the sparse matrix data structure that is a structured grid diagonal storage called (SG-DIA). SpMV flop performance is evaluated using the SG-DIA storage scheme and compared to other sparse matrix storage formats available in the PETSc library which relies on CUSP library for basic vector operations. For a structured grid where the number of cell components is above 6, proposed SpMV using SG-DIA is 1.5 faster than those implemented using standard CSR, ELLPACK, and DIA which are available in PETSc/CUSP library. OrCUDA toolkit auto-tuning doubles the speedup of generated kernel compared to hand-tuned code and CUSP library. To compare with our results, the GPUs used have different peak Flops as the K20X (our) and K10 with 3.95 and 4.58 TFLOPS, respectively. Figure 9b shows the fraction of peak flop performance for SpMV using the proposed Bdia storage and optimization, SG-DIA storage [22], CUSP-CSR, CUSP-ELLPACK, and CUSP-DIA for two grid sizes and for 2×2, 4×4, and 8×8 component blocks. This corresponds to the largest problem size that our GPU can support.

Proposed Bdia storage and SpMV kernel optimization approach gives faster results than SG-DIA and the CUSP storages. Both SpMV(Bdia) and SpMV(SG-DIA) use very optimized storage. In addition, kernel auto-tuning determines the most suitable grid size, thread block size, and compiler flags. In general, SpMV does not achieve a significant fraction of peak performance due to runtime overheads in indexing the data structure and fetching data that is referenced only once. Since most linear algebra solvers spent a large fraction of their runtime in SpMV operations, our approach explicitly optimize the SPMV kernel for the class of generalized hepta GH sparse matrices including cell–cell, well–cell, and cell–well sub-matrices. Furthermore, we optimized the storage and the SpMV kernel for each of the three sub-matrix data structures by taking advantage of the general properties of each sub-matrix in the context of reservoir simulation.

Numerical libraries have computational kernels, which are important for solving large sparse linear systems iteratively. Libraries generally have optimized codes for a limited set of linear algebra operators. However, applications that implement an iterative linear solver using the library operators often fail to leverage the full potential of the accelerator because consecutive cuBLAS operators do not keep reusable data

locality. Accelerating the BiCG-Stab solver for GPUs [6] is based on reformulating the strategy and creating application-specific kernels as opposed to an implementation that invoke cuBLAS library operators. The optimizations used are: (1) reducing the number of kernel launches by merging operators, reducing GPU–host communication, and enhancing the operator routines.

The number of kernels launches is reduced to eight for the BiCG-Stab compared to 16 cuBLAS function invocations. However, these eight kernel invocations cause additional overhead and loss of data locality as many computed vectors must be saved back into GM before kernel exit, and later loaded back into ShM after kernel re-entry. Our in-kernel approach uses a custom inter-block synchronization to stay in-kernel to avoid multiple kernel invocations and loss of data locality, which is particularly useful for inner products and collective writes to GM.

Our storage scheme confirms that the diagonal-based storage format is profitable for structured matrices especially when NZ values are limited to a small number of matrix diagonals [8,9]. It also takes advantage of the blocked structure of the matrix and its dependence over the number of cell components as concluded in [12]. Since each row has a fixed number of NZs, mapping a row with fixed number of blocks to threads leads to balanced thread load [15]. Furthermore, the scope of the data needed for SpMV is transferred from GM to ShM using coalesced memory access, where the sum of products are accumulated using cached data. In our case, we have used customized storage and SpMV organization for the hepta-matrix and the two well matrices so that to minimize the storage and to take full advantage of the data locality. One characteristic of iterative algorithm, like the BiCG-Stab, is that at each iteration the thread block updates a number of intermediate vectors but with fixed range for each thread block. In-Kernel implementation allows reuse of these range vectors that are kept in shared memory from starting the iterative solver to its convergence. Hence, there is a lot of time saving compared to writing back these range vectors on GM, before each kernel exit, and reloading them back into ShM on kernel re-entry in the case of multi-kernel invocation. Our proposed implementation is customized for the class of iterative linear algebra algorithms by taking advantage of combining the reductions and global writes to GM to reduce the number of global synchronization. Furthermore, we proposed a customized inter-block synchronization without atomic (array notifications) and preserving data locality along with the iteration space.

For general sparse matrix data layout, the AMB [25] achieves favorable speedup over the cuSparse library by using blocking optimization, thread load balancing, and adjusting the thread communication to computation ratio. The optimizations needed for efficient implementation of the iterative solver using SpMV are not addressed. A kernel exit-re-entry is assumed in the SpMV implementation which does not preserve the vector data locality and incurs excessive synchronization overhead when the SpMV results are used within an iterative solver implementation. Using the AdELL+ [23] for irregular sparse matrix data layout, the SpMV optimization is done at the warp level such as controlling the warp granularity and blocking. Similar to the proposed approach, a parametric coding is used to enable the use of auto-tuning to find the most suitable architectural parameters. However, in case of matrices with increasing regularity some extra storage data structure is required to do the computations in SpMV kernel with matching vector elements. It is clear that SpMV is not a stand alone

objective because it is repeatedly invoked (more than 80% of the time) in any iterative solver. We proposed a comprehensive SpMV-Solver optimization that is aware of the thread block data locality and the need for global synchronization to guarantee exact algorithm behavior. Specifically, we proposed a customized and modular construct to (1) minimize the overheads involved in repeatedly loading and saving vector data at the thread block level and (2) implement inter-block synchronization with the least possible overhead. Spare computations that must done anyway are processed during the waiting time at the barrier to avoid the traditional polling that may increase access delays of active blocks. We believe this approach is most suitable for efficiently implementing linear algebra solvers on GPUs which are at the heart of science simulation.

The OpenACC model [16,21] depends on launching a different kernel for each parallel region which is the only way to synchronize across all thread blocks. OpenACC is very useful for early implementation of iterative linear algebra solvers (ILAS). Having little control over code transformations makes the above optimizations unpractical to implement optimized ILAS.

9 Conclusion

In today's scientific computing, GPUs are becoming a core computing resource to improve the accuracy in scientific and engineering simulation with significant acceleration. In Petroleum Forward Reservoir Simulation (FRS), an application of a stencil to a structured grid, leads to a system of linear equations where the solver matrix has a generalized hepta-diagonal (GH) pattern with some regularity and structural uniqueness. In this paper, we presented a customized storage scheme that takes advantage of GH sparsity pattern and stencil regularity in optimizing both the matrix storage and SpMV computation. An application of the proposed storage format, we have also presented an optimized implementation of BiCG-Stab iterative linear solver using an In-Kernel (IK) structure to preserve vector data locality over the iterations. Unlike the solver implementation using numerical library operators, IK structure of BiCG-Stab is intended to avoid multiple kernel (MK) invocations. This approach helped in preserving vector data locality by avoiding reloading of vector data to memory on each kernel exit and re-entry. For inter-block synchronization, we used a lock-free mechanism in which thread blocks carry out some independent computations instead of wasting bandwidth in repeatedly polling the memory. We presented the performance evaluation of both SpMV and BiCG-Stab implementations using the proposed sparse matrix storage scheme for a class of GH matrices that have derived from a range of FRS-structured grids. Results show the significant performance improvements in SpMV and BiCG-Stab implementations as compared to other proposed implementations found in literature using standard sparse storages and numerical library routines with multiple kernel invocations.

Acknowledgements This project was funded by the National Plan for Science, Technology, and Innovation (MAARIFAH) King Abdulaziz City for Science and Technology- through the Science & Technology Unit at King Fahd University of Petroleum and Minerals (KFUPM) the Kingdom of Saudi Arabia, award number (12-INF3008-04). Thanks for King Fahd University of Petroleum & Minerals (KFUPM) for computing support.

References

1. Abu-Sufah W, Karim A (2012) An effective approach for implementing sparse matrix–vector multiplication on graphics processing units. In: IEEE 14th International Conference on High Performance Computing and Communication, pp 453–460
2. Ahamed C, Kassim A, Frdric M (2012) Iterative methods for sparse linear systems on graphics processing unit. In: IEEE 14th International Conference on High Performance Computing and Communication, pp 836–842
3. Alejandro D, Polanco R (2009) Collective communication and barrier synchronization on NVIDIA CUDA GPU. MS thesis, University of Kentucky
4. Aliaga J, Perez J, Quintana-Orti E, Anzt H (2013) Reformulated conjugate gradient for the energy-aware solution of linear systems on GPUs. In: 42nd International Conference on Parallel Processing (ICPP), pp 320–329
5. Anzt H, Tomov S, Dongarra J (2014) Implementing a sparse matrix vector product for the sell-c/sell-c-σ formats on NVIDIA GPUs. Technical report on UT-EECS-14-727, University of Tennessee
6. Anzt H, Tomov S, Luszczek P, Sawyer W, Dongarra J (2015) Acceleration of GPU-based krylov solvers via data transfer reduction. Int J High Perform Comput Appl 29(3):366–383
7. Balay S, Abhyankar S, Adams M, Brown J, Brune P, Buschelman K, Eijkhout V, Gropp W, Kaushik D, Knepley M et al (2014) PETSc users manual revision 3.5. Technical report on Argonne National Laboratory (ANL)
8. Bell N, Garland M (2008) Efficient sparse matrix–vector multiplication on CUDA. Technical report on NVR-2008-004, NVIDIA Corporation
9. Bell N, Garland M (2009) Implementing sparse matrix–vector multiplication on throughput-oriented processors. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09. ACM, pp 18:1–18:11
10. Bell N, Garland M (2012) CUSP: generic parallel algorithms for sparse matrix and graph computations. https://code.google.com/archive/p/cusp-library/
11. Bordawekar R, Baskaran MM (2008) Optimizing sparse matrix–vector multiplication on GPUs. Technical report on RC24704, IMB Research
12. Buatois L, Caumon G, Lvy B (2009) Concurrent number cruncher: a GPU implementation of a general sparse linear solver. Int J Parallel Emerg Distrib Syst 24(3):205–223
13. Davis TA, Hu Y (2011) The University of Florida sparse matrix collection. ACM Trans Math Softw 38(1):1:1–1:25
14. Goharian N, Jain A, Sun Q (2003) Comparative analysis of sparse matrix algorithms for information retrieval. Computer 2:0–4
15. Greathouse JL, Daga M (2014) Efficient sparse matrix–vector multiplication on GPUs using the CSR storage format. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '14. IEEE Press, Piscataway, pp 769–780
16. Grillo L, de Sande F, Reyes R (2014) Performance evaluation of OpenACC compilers. In: 22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp 656–663
17. Heroux MA, Bartlett RA, Howle VE, Hoekstra RJ, Hu JJ, Kolda TG, Lehoucq RB, Long KR, Pawlowski RP, Phipps ET et al (2005) An overview of the Trilinos project. ACM Trans Math Softw 31(3):397–423
18. Hoberock J, Bell N (2010) Thrust: a parallel template library. https://thrust.github.io/
19. Huan G, Qian Z (2012) A new method of sparse matrix–vector multiplication on GPU. In: 2nd International Conference on Computer Science and Network Technology (ICCSNT), pp 954–958
20. Khan AH, Al-Mouhamed M, Firdaus LA (2015) Evaluation of global synchronization for iterative algebra algorithms on many-core. In: 16th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD), pp 1–16
21. Lee S, Vetter JS (2012) Early evaluation of directive-based GPU programming models for productive exascale computing. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12. IEEE Computer Society Press, Los Alamitos, pp 23:1–23:11
22. Lowell D, Godwin J, Holewinski J, Karthik D, Choudary C, Mametjanov A, Norris B, Sabin G, Sadayappan P, Sarich J (2013) Stencil-aware GPU optimization of iterative solvers. SIAM J Sci Comput 35(5):S209–S228

23. Maggioni M, Berger-Wolf T (2016) Optimization techniques for sparse matrixvector multiplication on GPUs. J Parallel Distrib Comput 9394:66–86
24. Matam KK, Kothapalli K (2011) Accelerating sparse matrix vector multiplication in iterative methods using GPU. In: IEEE International Conference on Parallel Processing (ICPP), pp 612–621
25. Nagasaka Y, Nukada A, Matsuoka S (2016) Adaptive multi-level blocking optimization for sparse matrix vector multiplication on GPU. Procedia Comput Sci 80:131–142 (**International Conference on Computational Science 2016**)
26. Neelima B, Reddy GRM, Raghavendra PS (2014) Predicting an optimal sparse matrix format for SpMV computation on GPU. In: IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW), pp 1427–1436
27. NVIDIA: CUDA basic linear algebra subroutines (cublas) library, CUDA sparse matrix (cusparse) library. https://developer.nvidia.com/gpu-accelerated-libraries
28. NVIDIA (2013) Tuning CUDA applications for kepler. http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html. Accessed 10 June 2013
29. NVIDIA (2008) NVIDIA CUDA programming guide 2.0. NVIDIA
30. NVIDIA (2011) CUDA toolkit. https://developer.nvidia.com/cuda-toolkit
31. NVIDIA (2014) cuSparse library. https://developer.nvidia.com/cusparse
32. Owens JD, Luebke D, Govindaraju N, Harris M, Krüger J, Lefohn AE, Purcell TJ (2007) A survey of general-purpose computation on graphics hardware. In: Computer graphics forum, vol 26. Wiley Online Library, pp 80–113
33. Saad Y (2003) Iterative methods for sparse linear systems, 2nd edn. Society for Industrial and Applied Mathematics, Philadelphia
34. Tomov S, Dongarra J, Baboulin M (2010) Towards dense linear algebra for hybrid GPU accelerated manycore systems. Parallel Comput 36(5):232–240
35. Xiao S, c. Feng W (2010) Inter-block GPU communication via fast barrier synchronization. In: IEEE International Symposium on Parallel Distributed Processing (IPDPS), pp 1–12
36. Yan SEA (2014) yaSpMV: Yet another SpMV framework on GPUs. ACM SIGPLAN Notices 49(8):107–118
37. Yang LT (2000) Data distribution and communication schemes for IQMR method on massively distributed memory computers. In: Proceedings of the International Workshop on Parallel Processing, ICPPW, Toronto, Canada, August 21–24, pp 299–306
38. Zaza A (2015) A CUDA based parallel multi-phase oil reservoir simulator. PhD thesis, Computer Engineering Department, King Fahd University of Petroleum & Minerals (KFUPM), Saudi Arabia