Chapter 17
Performance Characteristics for Sparse Matrix-Vector Multiplication on GPUs

Sarah AlAhmadi, Thaha Muhammed, Rashid Mehmood, and Aiiad Albeshri

17.1 Introduction

High-performance computing techniques can effectively enhance the performance of sparse linear equation systems, which have Sparse Matrix-Vector multiplication (SpMV) as the most important scientific computation unit [1]. Numerous important scientific, engineering and smart city applications require computations of sparse matrix-vector multiplication (SpMV) [2–6]. SpMV is a core computing part of many scientific and engineering applications such as finite element methods, signal processing, magneto-hydrodynamics, graphics processing, electrical power systems, data mining, graph analytics, and information retrieval [1, 7–11]. The widespread importance of sparse matrix computation has become research hotspots and brought about significant research endeavors into implementations based on modern-day parallel processors, mainly GPUs [1, 7, 10, 12, 13]. However, there are many challenges in computing SpMV such as the differences in sparsity patterns, that make such computations difficult.

The irregularities of sparse patterns result in a number of matrix representation issues [10]. Thus, there exists diverse sparse matrix storage layouts intended to exploit various sparsity designs and distinctive techniques for getting and manipulating matrix entries especially on GPUs. Direct or indirect improvements in data layout and data access pattern are solutions to obtain high throughput or indirect improvements. Furthermore, the SpMV performance is affected by the parallel

S. AlAhmadi (✉) · T. Muhammed · A. Albeshri
Department of Computer Science, King Abdulaziz University, Jeddah, Saudi Arabia
e-mail: salahmadi0084@stu.kau.edu.sa; m.thaha.h@ieee.org; aaalbeshri@kau.edu.sa

R. Mehmood
High Performance Computing Center, King Abdulaziz University, Jeddah, Saudi Arabia
e-mail: RMehmood@kau.edu.sa

© Springer Nature Switzerland AG 2020

R. Mehmood et al. (eds.), *Smart Infrastructure and Applications*,
EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-13705-2_17

m.thaha.h@ieee.org

computing device platform being used [10]. SpMV algorithm that achieved good performance in one parallel device platform may not be as efficient as on other platforms due to the difference of the architecture and capabilities between the platforms.

This research will explore the SpMV and Jacobi iterative methods on GPUs with the aim to understand the performance bottlenecks and possibly address the limitations of the existing approaches. In Sect. 17.2, an overview of SpMV and Jacobi iterative techniques are provided. Section 17.3 explores the GPU architecture and performance characteristics of applications on GPU along with techniques to optimize the performance of SpMV. In Sect. 17.4, we explore the important storage formats for SpMV computations on GPU architecture. Finally, in Sect. 17.5 we analyze and discuss the performance of the notable storage formats using the identified performance characteristics and criterions.

17.2 SpMV and Iterative Methods

Sparse Matrix-Vector product (SpMV) is the most important process in scientific computing and engineering applications [1, 7, 8, 12, 14–17]. The performance of SpMV can be improved using parallel computing [1, 15, 16]. Sparse matrix is a matrix that have mostly zeros and very few non-zero elements [12, 17]. The processing of such matrices involves removal of the zeros elements to deal with just the non-zero (nnz) elements. The challenges involved in computing SpMV are numerous. Some of the major challenges are irregularity of the matrices, data transfer between host and device, load imbalance among the threads, memory access, and memory management (storage formats) [1, 7–9, 12, 14–17].

Iterative methods consist of a sequence of computations performed iteratively to produce approximate solutions that gradually reaches the accurate solution. They are, furthermore, partitioned into stationary methods (i.e., Jacobi) and non-stationary methods (i.e., conjugate gradient) like [17, 18]. In this work our attention will be on the stationary methods specifically the Jacobi iterative methods. Jacobi is an excellent candidate to be implemented on GPU although it is slower than other iterative methods, since it's inherently parallel.

Linear systems which have formula $Ax = b$ can be solved using Jacobi method as follows in each iteration compute $Ax = b$ as matrix-vector product, then test for convergence, and repeat until convergence. It involves partitioning of the matrix A into three parts: diagonal, upper-triangular, and lower-triangular portions [9, 19]. Thus, in matrix terms, the Jacobi method can be expressed as in Eq. (17.1):

$$x^k = D^{-1}(L + U)\, x^{k-1} + D^{-1} b \tag{17.1}$$

where k denotes the number of iterations, D is the diagonal entries, L is the lower-triangular matrix, and U is the upper-triangular matrix. Figure 17.1 depicts the Jacobi iterative technique and the SpMV involved. Many researchers have attempted

m.thaha.h@ieee.org

```
k=0
while not convergence do
  for i = 1 to n
    x_i = 0
    for j = 1 to n
      if j≠i
        x_i = x_i + a_i,j * x_j^(k)
      end
    end
  x_i^(k+1) = (b_i - x_i)/a_i,i
  end
  k = k + 1
end
```

Fig. 17.1 Algorithm for Jacobi iterative method depicting the SpMV operations involved

to improve the performance of iterative methods for sparse linear equation systems and SpMV computations [2, 17, 18, 20–27].

17.3 GPU: An Overview

In this section, we provide a brief overview of the general GPU architecture. We further discuss the GPU characteristics that affect the computational performance of the GPU and discuss various optimizations that enhance the performance.

17.3.1 Architecture

In the recent decade, GPUs are considered as a general-purpose processing unit instead of a mere graphics processing unit [7, 12, 28, 29]. GPU has attracted HPC researchers and has become popular in scientific computing due to its high computation capabilities, massive performance, effective usage of memory bandwidth, and the ability to accelerate existing large systems which have been implemented on other processors like CPU [7–9, 14, 30, 31].

Thus, GPUs become an important platform to implement sparse matrix computation to accelerate the performance of SpMV multiplication by processing them parallelly [8, 14, 15, 32]. Hence, many researchers have developed and optimized the existing algorithms to get best utilization out of these devices. Their speed can reach to Teraflops for single-precision calculations and half of this value for double-precision processing [9, 12]. Understanding GPU architecture is important for efficient utilization of the resources. However, the architectures are different for different GPU generations such as Kepler, Fermi, GeForce, PASCAL, and VOLTA. Different GPU families have been designed for different purposes such as

m.thaha.h@ieee.org

GeForce for graphics computation [33] and Tesla P100 for datacenters acceleration [34]. They differ largely on the parallel computing capabilities they have, thus the throughput performance delivered vary. Pascal, for example, is currently the most powerful architecture design for GPU. It turns a normal computer into a supercomputer and provides remarkable performance [33]. Tesla P100 belongs to the PASCAL family and it delivers a double-precision floating point about 5.3 TFLOPS, while Tesla V100 which belongs to the VOLTA architecture reaches to 7 TFLOPS [35, 36]. For all Nvidia versions among the last two decades along with its purposes, see [37]. A discussion on the Tesla P100 architecture can be seen in [34] and for Kepler architecture refer [33]. In addition to the architecture of the device, the selection of the best storage format for a given input matrix is a key issue [9, 12, 15].

Working with GPUs involves working with a heterogenous platform consisting hierarchies of computational units and memories. Figure 17.2 shows the different types of memories on such platforms and Fig. 17.3 shows the hierarchy of memory and computations on GPU. In general, GPU consists of an array of Streaming Multiprocessors (SM) that contains processing cores, and many types of memories such as registers and cache. The programming model for the GPU is single instruction multiple data (SIMD) applied into groups of 32 threads called warps. In subsequent sections, we further discuss about warps and its effects on the performance of SpMV.

Compute Unified Device Architecture (CUDA) is an API dedicated to GPU programming [38]. It depends on the C language and presents new possibilities for accelerating GPU kernels. Further, CUDA is developed to simplify and improve GPU programming and accelerating high-performance parallel computations [1, 7,

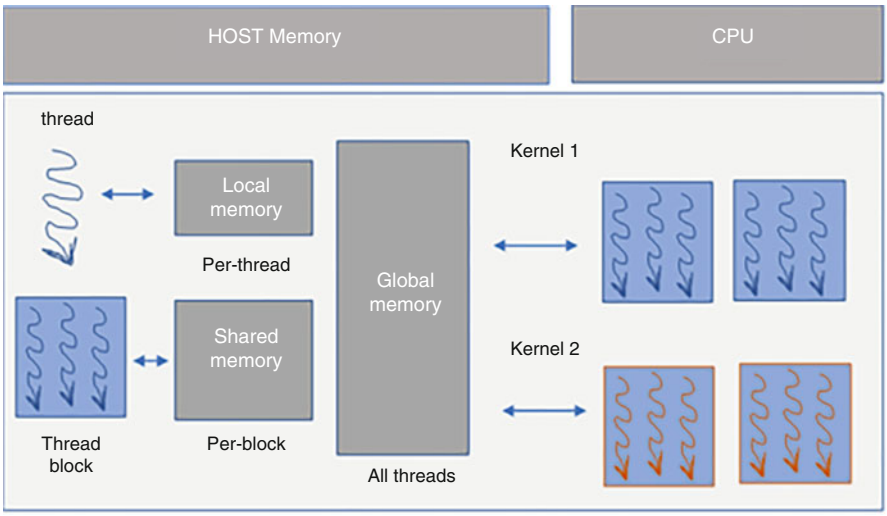

Fig. 17.2 Types of memory on heterogeneous platform

m.thaha.h@ieee.org

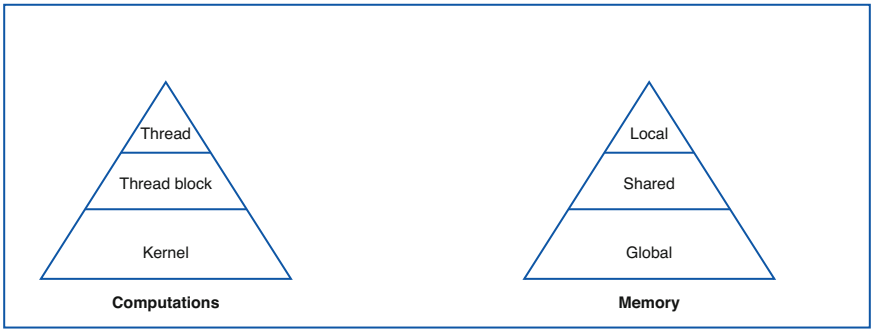

Fig. 17.3 Memory and computation hierarchies on GPU

38, 39]. CUDA views GPU as a grid of blocks where each block has a set of threads. The grid of blocks can be organized either as one-, two-, or three-dimensional computing units. A 2D grid example can be seen in [33].

17.3.2 Performance Characteristics: Discussion

The features that affect the performance of computations on GPUs can be broadly classified into three: (1) Execution configuration, (2) Memory throughput, and (3) Instruction throughput. In the following subsections, we discuss these performance characteristics in detail.

Execution Configuration These are the parameters that need to be configured at execution to improve the performance of GPU computations. The major execution configurations are:

1. *Active warps (Resource usage):* Registers and shared memory are critical components and need attention because it affects the number of active warps which in turn affects the GPU utilization. When the number of active warps is maximum, the GPU utilization is at maximum. The number of active warps can be maximized by good management of compute resources, registers, and shared memory. Reducing the number of registers utilized by a kernel results in higher warps being processed simultaneously. And when a thread block consumes more shared memory, fewer thread blocks are processed simultaneously by an SM. If the amount of shared memory used by each thread block is reduced, then more thread blocks can be processed simultaneously.
2. *Occupancy:* Occupancy is having enough warps to keep the device completely occupied. It is the ratio between active warps and maximum number of warps. We can compute occupancy by dividing the maximum number of threads per SM by 32. We can check it using CUDA occupancy calculator or the nvprof profiler. "*To enhance the occupancy, the thread block configuration needs to be*

m.thaha.h@ieee.org

resized or the resource usage needs to be readjusted to permit more active warps simultaneously and improve utilization of compute resources."

3. *Memory operations*: Load and store operations on the data should be measured to find the efficiency of the operation.

Memory Throughput Efficient utilization of the theoretical memory bandwidth of the GPUs improves the computational performance. The bandwidth utilization can be improved by considering the following factors:

1. *Data Transfer between Host and Device*: For the best performance of kernels, data transfers should be minimized between the host and device whenever possible and should be optimized by various techniques [40]. Moving more code from host to device is an efficient way to optimize the transfer process. In addition, every data transfer has an associated overhead, hence grouping many small transfers into single transfer reduces the overhead associated with each transfer and produces an overall better performance.
2. *Memory Access*: Memory access is an important factor that affects the overall performance of GPU applications. GPU has many types of memory as shown in Fig. 17.3. Scattered addresses in global memory need to be avoided; coalesced and aligned access can overcome throughput reduction. Thus, to increase global memory throughput, it is important to make the memory access transactions both aligned and coalesced. Coalesced access occurs within a warp scope when all the 32 threads in a warp use one memory transaction; more precisely, they access a contiguous chunk of memory [35]. Aligned access is to make sure the first address of a device memory transaction is a multiple of the transaction size, which usually has either 32, 64, or 128 bytes depending on the target device characteristics [35, 40]. Figure 17.4 shows the different memory access patterns.

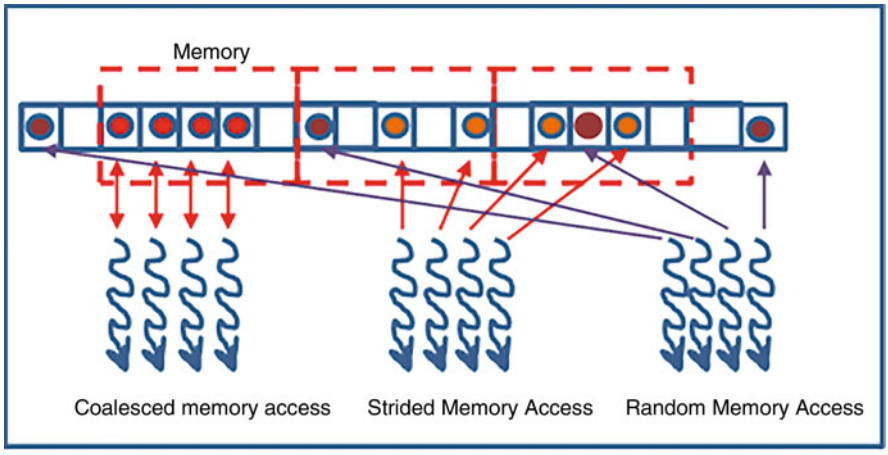

Fig. 17.4 Memory Access Patterns

m.thaha.h@ieee.org

The best access pattern is the coalesced access while the worse is the random access. As a rule, the more memory transactions required by a warp, the lower the memory throughput which leads to lower performance.

Instruction Throughput This describes the instruction optimizations that lead to the best performance. It can be summarized into three main perspectives as the following:

- *Arithmetic instructions*: Avoiding instructions that cost many operations per clock cycle such as mod operator. Avoid non-required conversions between datatypes.
- *Control Flow Instructions and Warp Divergence*: Control flow instructions (if, do, for, switch, while) can significantly impact the performance as it may cause warp divergence problem which degrades the instruction throughput [40]. Warp divergence occurs when the threads inside a warp have different execution paths. This conflicts with the fact that the GPU is a single instruction multiple data (SIMD) processor which would require all threads in a warp to execute one instruction. More precisely, threads on GPUs are organized as warps and each warp executes one instruction at a time for all threads inside that warp, each with its private data (SIMD). When control flow constructs are assigned to a warp, different branches might occur (e.g., some threads execute an if block when the condition is correct while others execute else block when the condition is wrong), which results in executing multiple instructions per warp. As a result, this process will be serially executed which results in idle threads in a warp as only one instruction will be executed at a time while next instruction will be loaded after the current instruction is finished [35, 41]. In other words, the instructions will be executed sequentially, thus the total number of instructions per warp increases. So, for best performance we should avoid different execution paths within a warp.
- *Synchronization:* It impacts the performance because of two reasons [35]. The first is the cost of the number of operations it requires (differs according to compute capability of the target device) while the second is forcing the multiprocessor to be idle while it is not required.

17.3.3 Performance Optimization Strategies

Performance optimization strategies according to [38] can be classified into three main categories, parallelism optimization, optimization of memory throughput, and optimization of instruction throughput. Each of these dimensions can be quantified using several metrics which can be measured using tools such as NVIDIA visual profiler, nvprof as command-line profiler tool, or by comparing the achieved throughput of a kernel to the corresponding peak theoretical throughput of the device to show how much improvement has been achieved by the kernel. The dimensions with their metrics have been explained in Table 17.1. For example, we can observe the memory operations in Nsight profiler using memory statistics menu.

m.thaha.h@ieee.org

Table 17.1 Performance dimensions of GPU Kernels

Performance dimension	Performance angles	Helping metrics
Memory optimization	• Aligned access	– gld_efficiency
	• Coalesced access	– gst_efficiency
	• Data transfer between host and device	– gld_transactions
		– gst_transactions
		– gld_throughput
		– gst_throughput
Instruction throughput	• Instruction throughput	– branch_efficiency
	• Warp divergence	– inst_per_warp
		– warp_execution_efficiency
Execution configuration optimization	• Occupancy	– achieved_occupancy
	• Tune grid blocks size	

Also, memory transaction metrics (**e.g., gld_transactions** and **gst_transactions**) can act as indirect indicators to measure the coalesced accesses. Higher memory transactions indicate a high probability of uncoalesced access. List of CUDA performance metrics collected by the nvprof can be found on [40].

17.3.4 Performance Optimization: Discussion

Looking for best kernel performance requires tuning multiple performance factors. It can be likened as a puzzle board that needs to compose many pieces to get a complete picture. We should look at different angles using multiple metrics to build better combination of performance aspects and get the best performance. However, there are possibilities of conflicts between these performance aspects that may degrade the overall performance even if we achieve high scores for individual factors. For example, getting more occupancy does not ensure the best performance, we can find that in some cases, low occupancy also has provided higher performance because there are other factors affecting the overall performance, for example, memory operations. In the same manner, getting high memory throughput does not equate to the best performance due to low efficiency these operations might have. As a matter of fact, memory efficiency is very important aspect to consider. It can be improved by changing the thread/block configuration. In general, there are some good tips for better configuration, keeping the block size always a multiple of the warp size (i.e., 32) and launching more blocks by setting the second dimension (block.y) to 1 to reduce the block size and obtain more blocks to launch. This will enhance the inter-block parallelism [35].

Therefore, enhancing the performance of GPU kernels can be done on multiple levels. One level might be focusing on exposing more parallelism by managing the

m.thaha.h@ieee.org

used resources such as registers and shared memory, or by controlling the occupancy (higher level of active warps at any given time) or any other aspects on this level as explained on previous section. Memory access management is another level to looking for. Optimization memory access to ensure coalesced and aligned access can immensely enhance the performance. The last level that we can work on is enhancement of instruction throughput by avoiding warp divergence or avoiding costly arithmetic operations such as mod operator. All these are examples and each level represents a separate field of optimizations. We can optimize on one level or more to find a good balance to get the better performance. Thus, looking for best GPU performance is a complicated process and requires checking the kernel from many angles and it does not depend on just a single metric.

17.4 SpMV Storage Formats and Computation Techniques

Data structures are a core aspect when dealing specially with SpMV and GPUs [8]. They have a strong impact on the performance of the algorithms that are used to solve SpMV [7–9, 15]. It represents the storage pattern of the input matrices in the memory and is responsible in providing the best data access [9, 14]. Numerous efforts have been made to improve storage formats specifically for SpMV on GPU and other architectures for iterative linear solvers on GPUs since sparse matrices show up in many applications which involve diverse computational patterns [16]. Accordingly, various storage formats have been proposed to facilitate the productivity and recovery of important information from the input matrix. The most prominent formats are the CSR, Coordinate format (COO), DIA, and ELL [39, 42]. In addition, some adaptation have been made to these basic formats such as CSR5 [43] and CSRNS [7], along with other hybrid Schemes [1, 8–10, 31, 42, 44]. CSR scheme is preferred always among comparable formats and it has been chosen because it is widely adopted, general-purpose storage format, and gives minimum memory accesses [7–9, 15, 16]. Furthermore, all prior storage formats are considered explicit while there are other storage schemes such as MTBDDs [45] which are considered implicit formats [17].

In this research, we have included our performance analysis on CSR, COO, ELL, DIA, HYB, and CSR5 schemes. The descriptions of these formats along with their analysis are explained in the next section.

17.5 Performance Analysis of Notable Sparse Storage and Computation Techniques

In this section, we explore several research efforts for SpMV optimization on GPU over the last years. We first describe how each scheme store the data and then we have defined the issues and limitations of each scheme. We have classified the

m.thaha.h@ieee.org

techniques according to the basic SpMV formats they are derived from. The basics storage formats are COO, CSR, ELL, DIA, and HYB.

17.5.1 Sparse Storage and SpMV Kernels: Qualitative Analysis

In the following subsection, we discuss some of the notable sparse formats and associated SpMV techniques [39, 44]:

- **The Coordinate (COO)** format is the most basic data structure to store a sparse matrix. It is made of three arrays: Row, Col, and Data to store the row indices, the columns indices, and the values of non-zero components, respectively.
- **The Compressed Sparse Row (CSR)** format is the most well-known format for sparse matrix storage. It comprises three arrays: RowPtr, Col, and Data to store row pointers to the offset of each row, indices of non-zero components, and values of non-zero components, respectively.
- **The ELLPACK (ELL)** structure stores a sparse matrix in two arrays: Data and Col. The array Data stores the values of non-zero components while Col array stores the columns indices of each non-zero component.
- **The Hybrid ELL/COO (HYB)** structure stores the greater part of non-zero components in ELL format and the rest of the non-zero components in COO. All non-zero components at the columns on the left of a threshold value are stocked in the ELL and the rest non-zero components are represented as COO format.
- **CSR5** proposed by [43] is an optimization of CSR format and it combines segmented sum technique for better load balance and compressed row data for better load/store operation efficiency. It is insensitive to sparsity structure of the input matrix. The matrix is partitioned into groups of 2D tails. These tails require extra information indicating their start index and columns indices, named as tail_ptr and tail_descriptor arrays, respectively. In addition, it has the CSR arrays val, col., and ptr. Thus, we have totally row_ptr, col_idx, val, tile_ptr, and tile_desc, where tile_desc further includes four arrays. tile_ptr works as row_ptr on CSR and it stores the row index of the first entry in each tile. Tail_desc has four different data structures, namely bit_flag, y_offset, seg_offset, and empty_offset arrays. These four arrays denote the start of each row inside the tiles, the address of the partial sum for each column, accelerating the segmented sum, and help the partial sums to find correct locations in y if the tile includes any empty rows. The illustrations of these schemes are shown in (Figs. 17.5, 17.6, 17.7, 17.8, 17.9 and 17.10).

Fig. 17.5 Original matrix

$$A = \begin{bmatrix} 1 & 0 & 3 & 0 \\ 0 & 4 & 1 & 0 \\ 2 & 0 & 7 & 5 \\ 0 & 6 & 0 & 8 \end{bmatrix}$$

m.thaha.h@ieee.org

$$Var\_Arl = [1\ 3\ 4\ 1\ 2\ 7\ 5\ 6\ 8]$$
$$Col\_Arr = [0\ 2\ 1\ 2\ 0\ 2\ 3\ 1\ 3]$$
$$Ptr\_Arr = [0\ 2\ 4\ 7\ 9]$$

Fig. 17.6 CSR scheme

$$Val\_Arr = \begin{bmatrix} 1 & 3 & * \\ 4 & 1 & * \\ 2 & 7 & 5 \\ 6 & 8 & * \end{bmatrix} \qquad Col\_Arr = \begin{bmatrix} 0 & 2 & * \\ 1 & 2 & * \\ 0 & 2 & 3 \\ 1 & 3 & * \end{bmatrix}$$

Fig. 17.7 ELL scheme

$$Val\_Arr = \begin{bmatrix} * & 1 & * & 3 \\ * & 4 & 1 & * \\ 2 & 7 & 5 & * \\ 6 & 8 & * & * \end{bmatrix} \qquad offset\_Arr = [-2\ \ 0\ \ 1\ \ 2]$$

Fig. 17.8 DIA scheme

$$ELL \begin{cases} Val\_Arr = \begin{bmatrix} 1 & 3 \\ 4 & 1 \\ 2 & 7 \\ 6 & 8 \end{bmatrix} & Col\_Arr = \begin{bmatrix} 0 & 2 \\ 1 & 2 \\ 0 & 2 \\ 1 & 3 \end{bmatrix} \end{cases}$$

$$COO \begin{cases} Val\_Arr = [5] \\ Col\_Arr = [3] \\ Row\_Arr = [2] \end{cases}$$

Fig. 17.9 HYB scheme: ELL and COO

$$A = \begin{bmatrix} 3 & 0 & 2 & 1 & 0 & 5 \\ 0 & 4 & 6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 5 & 3 & 0 & 0 & 7 \\ 0 & 1 & 0 & 8 & 0 & 2 \\ 9 & 5 & 7 & 6 & 7 & 0 \end{bmatrix}$$

$$val = \begin{bmatrix} 3 & 5 & 2 \\ 2 & 4 & 5 \\ 1 & 6 & 3 \end{bmatrix} \quad col\_index = \begin{bmatrix} 0 & 5 & 0 \\ 2 & 1 & 1 \\ 3 & 2 & 2 \end{bmatrix} \quad Tail\_0$$

$$val = \begin{bmatrix} 7 & 2 & 7 \\ 1 & 9 & 6 \\ 8 & 5 & 7 \end{bmatrix} \quad col\_index = \begin{bmatrix} 5 & 5 & 2 \\ 1 & 0 & 3 \\ 3 & 1 & 4 \end{bmatrix} \quad Tail\_1$$

$$m \times n = 6 \times 6 \quad nnz = 18 \quad \omega = 3 \quad \sigma = 3 \quad no\ of\ tails = (18/2) = 9$$

Fig. 17.10 CSR5 scheme

The main issues that should be taken into consideration regarding these basic formats are memory footprint in COO, coalesced access and thread mapping in CSR, and zero padding in ELL. Further, we shall illustrate the limitations of the selected techniques and analyze it with the performance evaluation criterion for SpMV and compare it with the performance characteristics of GPU. This comparison will show us the limitations of the existing techniques and how they are restricted to a few perspectives from a pool of GPU's performance considerations.

If we look at the performance criteria in each research, we can observe that the performance aspects covered on each technique is incomplete. Most of them focus on the speed of the technique while a few study the memory issues of their algorithms and seldom take into consideration the utilization rate of the GPU such as the occupancy rate of the device and the benefits from the massive parallelism provided by the target GPU. Table 17.2 illustrates the detailed performance data

m.thaha.h@ieee.org

for all the schemes discussed in this article and discusses the performance aspect considered in each research.

CSR and CSR Optimizations The main drawback of the scaler CSR (one thread per row) is the uncoalesced access of the data and indices arrays [1]. To rectify this issue, a vector CSR version is proposed (a warp per row) [1]. In addition, CSR is widely used for various types of sparse matrices, this flexibility introduces thread divergence problem especially for those sparse matrices with a variable number of non-zeros per row [1, 16]. This likely will cause many threads within a warp to be idle while waiting for the thread with the longest data to process. These drawbacks have been overcome by CSR vector version, but the performance of this version is strongly sensitive to the row size of the target matrix such that it is inefficient when rows have few non-zeros.

ELL and ELL Optimizations ELL accomplishes high performance on regular matrix structures (i.e., with an equivalent number of non-zeros on each row) [39, 46]. However, on irregular matrices unavoidably it leads to memory footprint inefficiency and misuse of computation (i.e., short rows make their thread inactive for most of the time) results in load imbalance. The granularity of ELL SpMV on GPU is one thread per row. Nevertheless, it implicates potential space wastage with the way that all rows are zero-padded to length $N_{\max}$. Subsequently, this configuration is most productive when the variance of non-zeros among rows is small [10].

AdELL+ SpMV kernel proposed by [46] is an improvement of ELL format and it is also kind of hybrid format that combines ELL and CSR. It outperforms the comparable kernels in terms of speed of execution measured on GFLOPS for both regular and irregular matrices. They have discussed memory bandwidth but without comparison with others, so we cannot decide about amount of improvements done on this point. They also have measured memory footprints compared with CSR structure and it has achieved less footprints than CSR.

HYB Single storage configuration only provides the best performance only in limited situations which gave birth to the idea of hybrid formats. HYB format, for example, is the first hybrid format consisting of COO and ELL formats to overcome the sensitivity to the sparsity structures in both ELL and COO. It successfully achieved good performance and is considered as one of the best formats especially on the unstructured matrices. However, it has higher costs including high level of data organization, has complicated program logic, and costs time in terms of memory transfer [12].

SHEC [10] is another segmented hybrid format that consists of ELL and CSR (vector version). They combine the advantage of ELL granularity (i.e., one thread per row) and CSR granularity (i.e., one warp per row). SHEC is intended for further improvements on the throughput of SpMV and specially to lessen the memory footprint on GPUs.

Another hybrid scheme has been proposed in [13] which combines DIA with the ELLPACK structure. This combination isolates the diagonal elements of the

m.thaha.h@ieee.org

sparse matrix using the DIA scheme while the residual elements are stored in ELLPACK format. This is immensely beneficial for iterative methods, specifically Jacobi iteration because it uses the diagonal values in its calculation, so the isolation on the proposed format will give faster access to the diagonal elements. However, the performance is limited compared to ELLPACK and it is highly efficient for those matrices having a relatively dense diagonal band [13].

CSR5 [43] have been introduced as a storage format that is based on segmented CSR. The authors considered computation intensity factor to measure their performance compared with others. It significantly improves the load imbalance problem that CSR suffer from. CSR5 is a complex storage format and requires several arrays. These arrays involve more memory access operations (many load operations) and large memory space to load this information which may affects the total memory efficiency. Poor resource management lead to less GPU utilization since it effects number of blocks and warps working concurrently which subsequently affects rate of the device occupancy. Furthermore, the memory bandwidth measurements are not provided and the technique is not space efficient due to the large number of arrays used. In addition to its complexity, CSR5 has significant overheads due to the preprocessing process such as the matrix transpose operations and transformation from CSR **val** and **col** arrays into CSR5 arrays

17.5.2 Performance Comparison

Table 17.2 compares the six considered SpMV kernels which are CSR (scalar vector), COO, DIA, ELL, HYB, and CSR5. The comparison is in terms of the used GPU device, peak theoretical values of performance and memory bandwidth, and the achieved performance and memory throughput. In addition, we provide the name of the matrix benchmark suites used in the experiments. Some have used wide variety of real application matrices derived from finite element method-based modeling, linear programming, circuit simulation, and connectivity graphs from partial web crawls. It should be noted that the notable CSR version is CSR scalar which has granularity of one thread per row. The comparison includes several experiments from different studies of the selected structures. In this article, we have considered experiments of double-precision computations and unstructured matrices excluding single-precision and structured matrices except for DIA and ELL schemes. We have considered the structured matrices for DIA and ELL because they are dedicated for such matrices. GFLOPS refers to performance throughput. The calculation for single-precision and double-precision flops are different. The formula to calculate the peak value of double precisions is given in Eq. (17.2).

$$2 \times \left[\text{multiply add}\right] \times \left[\#\text{of multiprocessors}/8\right] \times \left[\text{processor clock}/1000\right] \quad (17.2)$$

m.thaha.h@ieee.org

Table 17.2 Comparison of SpMV Kernels

	GFLOPS			Memory bandwidth	
Tech. name	Device	Peak GFLOPS	Obtained GFLOPS (MAX)	Peak memory bandwidth	Effective bandwidth (MAX)
(COO 1990) [37]	GTX 280 [35]	77.76	4	141.7	58
	GTX 285 [38]	88.56	5	159.0	–
(CSR Scalar 1990) [37]	GTX 280 [35]	77.76	4	141.7	55
	GTX 285 [38]	88.56	4.2	159.0	–
	GTX 980 [39]	144.1	18	141.7	–
(ELL 1985) [37]	GTX 280 [35]	77.76	13.5	141.7	140
	GTX 285 [38]	88.56	15	159.0	–
(DIA 1989) [37]	GTX 280 [35]	77.76	16.7	141.7	141
	GTX 285 [38]	88.56	18.2	159.0	–
(HYB 2008) [35]	GTX 280 [35]	77.76	14	141.7	141
	GTX 285 [38]	88.56	15.7	159.0	–
	GTX 980 [39]	144.1	15		–
(CSR5 2015) [39]	GTX 980 [39]	144.1	27	224	–

The peak GFLOPS discussed in this article is either calculated using Eq. (17.2) or from the device specifications given on the website, or is mostly reported in various researches. The obtained performance throughput measures the number of floating point operations per second, and it is calculated by dividing the required arithmetic operations by the average execution time [42]. Peak memory bandwidth is clearly defined on the device specifications, otherwise it can be calculated using Eq. (17.3).

$$(\text{Memory clock} \times \text{Bus Width}/8) \times \text{GDDR type multiplier} \tag{17.3}$$

GDDR multiplier values vary according to memory type. For GDDR3, GDDR5, and GDDR5X, it is 2, 4, and 8, respectively. Division by 8 is to change from bit to byte. Effective bandwidth is defined as the total number of bytes written/read by all threads divided by the average execution time [42].

In [39], they have implemented the basics formats for structured and unstructured matrices with single and double-precision computations. In addition, they have considered experiments with and without cache. They have considered the GPU performance measured in GFLOPS as well as memory bandwidth measured in GB/s as performance aspects. They have evaluated the performance results using single and double-precision floating points and measured the performance enhancement. However, they do not consider peak performance and peak memory bandwidth to measure the achievable performance compared with device capabilities. If we compare the achieved results with the peak values, we observe they have lower performance compared to the device capabilities as stated in Table 17.2.

In [39, 42], they have the same experiments on different devices with slight enhancements as compared to [42]. SpMV is a memory-bounded computation and

m.thaha.h@ieee.org

hence they did not achieve the peak performance of the used devices [10, 39, 46–48]. More precisely, if we study the performance characteristics that have been discussed in Sect. 17.3 for the selected kernels we can observe many limitations. Coalesced memory access, for example, is a difficult issue on sparse matrix computations because different storage schemes require many pointers that point to the address of the first element of the blocks, slices, and individual rows. However, the need for these addresses mean the need for more arrays (at least one beside the data array) which would result in loading more arrays into the device global memory. This would increase the memory transactions which may degrade the performance if the accessing pattern is not coalesced. Furthermore, instead of a single array, all the arrays should have a coalesced access to ensure better performance. COO, DIA, ELL, and HYB formats are fully coalesced [1]. On the other hand, CSR does not provide a coalesced access either for the data array nor to the other arrays. CSR5 supports memory coalesced access by accessing the data and column arrays in column-major order instead of row-major order as seen in the classic CSR.

Warp divergence is another performance characteristic and likely to occur on CSR. It results in load imbalance between threads; however, it is significantly less in CSR5 by dividing the elements into fixed-size tails. Moreover, other performance aspects such as instruction throughput, occupancy, block-thread heuristics, number of resources used, and other performance aspects are not considered. For the best performance and device utilization we should include a combination of performance characteristics to evaluate the performance which most researches lack of. Furthermore, the properties of GPU architecture included in the experiments have significant impact on the achieved performance as we have seen in our comparison. However, even with different GPU devices, the achieved SPMV performance is low as compared to the high throughput each device can provide.

17.6 Conclusion

In this chapter, we discussed the performance of SpMV on GPU architectures. We provided an architectural overview of GPU devices and defined the performance dimensions of GPU computations. We explored the performance of a few major existing sparse matrix storage formats. We concluded that there is lack of performance aspects considered during the evaluation of the existing SpMV algorithms, specifically to measure the memory throughput achieved by the SpMV computations. Since SpMV computations are memory-bound, the achieved performance should be compared to the peak theoretical bandwidth of the GPUs. We conclude that to achieve better performance analysis a combination of performance aspects/criterion should be noted. In addition, the performance of SpMV on different GPU device architecture varies. Hence, a comprehensive and standard set of performance characteristics need to be used by the researchers while comparing and analyzing SpMV on GPUs.

m.thaha.h@ieee.org

References

1. Yang, W., Li, K., Li, K.: A hybrid computing method of SpMV on CPU–GPU heterogeneous computing systems. J. Parallel Distrib. Comput. **104**, 49–60 (2017)
2. Mehmood, R., Lu, J.A.: Computational Markovian analysis of large systems. J. Manuf. Technol. Manag. **22**, 804–817 (2011)
3. Mehmood, R., Meriton, R., Graham, G., Hennelly, P., Kumar, M.: Exploring the influence of big data on city transport operations: a Markovian approach. Int. J. Oper. Prod. Manag. **37**, 75–104 (2017)
4. Mehmood, R., Alturki, R., Zeadally, S.: Multimedia applications over metropolitan area networks (MANs). J. Netw. Comput. Appl. **34**, 1518–1529 (2011)
5. Mehmood, R., Graham, G.: Big data logistics: a health-care transport capacity sharing model. Proc. Comput. Sci. **64**, 1107–1114 (2015)
6. Altowaijri, S., Mehmood, R., Williams, J.: A quantitative model of grid systems performance in healthcare organisations. ISMS 2010—UKSim/AMSS 1st International Conference on Intelligent Systems. Model. Simul. 431–436 (2010)
7. Huan, G., Qian, Z.: A new method of sparse matrix-vector multiplication on GPU. In: International Conference on Computer Science and Network Technology, pp. 954–958 (2012)
8. Hassani, R., Fazely, A., Choudhury, R.-U.-A., Luksch, P.: Analysis of sparse matrix-vector multiplication using iterative method in CUDA. In: 2013 IEEE Eighth International Conference on Networking, Architecture and Storage, pp. 262–266 (2013)
9. Cheik Ahamed, A.-K., Magoulès, F.: Efficient implementation of Jacobi iterative method for large sparse linear systems on graphic processing units. J. Supercomput. **73**, 3411–3432 (2017)
10. Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.R.: HPCTOOLKIT: tools for performance analysis of optimized parallel programs. Concurr. Comput. Pract. Exp. **22**, 685–701 (2010). http://hpctoolkit.org
11. Brahme, D., Mishra, B.R., Barve, A.: Parallel sparse matrix vector multiplication using greedy extraction of boxes. In: 2010 International Conference on High Performance Computing, pp. 1–10 (2010)
12. Ahamed, A.-K.C., Magoules, F.: Fast sparse matrix-vector multiplication on graphics processing unit for finite element analysis. In: 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems, pp. 1307–1314 (2012)
13. Guo, P., Wang, L., Chen, P.: A performance modeling and optimization analysis tool for sparse matrix-vector multiplication on GPUs. IEEE Trans. Parallel Distrib. Syst. **25**, 1112–1123 (2014)
14. Guo, P., Wang, L.: Auto-tuning CUDA parameters for sparse matrix-vector multiplication on GPUs. In: Proceedings—2010 International Conference on Computational and Information Sciences, ICCIS 2010, pp. 1154–1157 (2010)
15. Merrill, D., Garland, M.: Merge-based parallel sparse matrix-vector multiplication. In: International Conference for High Performance Computing, Networking, Storage and Analysis, SC 16, pp. 678–689 (2016)
16. Hou, K., Feng, W.C., Che, S.: Auto-tuning strategies for parallelizing sparse matrix-vector (SpMV) multiplication on multi- and many-core processors. In: Proceedings—2017 IEEE 31st International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2017, pp. 713–722 (2017)
17. Mehmood, R., Crowcroft, J.: Parallel Iterative Solution Method for Large Sparse Linear Equation Systems. UCAM-CL-TR-650. University of Cambridge, Computer Laboratory (2005)
18. Mehmood, R.: Disk-Based Techniques for Efficient Solution of Large Markov Chains. Computer Science, University of Birmingham (2004)
19. Barrett, R., Berry, M., Chan, T.F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., van der Vorst, H.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. Society for Industrial and Applied Mathematics, Philadelphia (1994)

m.thaha.h@ieee.org

20. Eleliemy, A., Fayez, M., Mehmood, R., Katib, I., Aljohani, N.: Loadbalancing on parallel heterogeneous architectures: spin-image algorithm on CPU and MIC. In: 9th EUROSIM Congress on Modelling and Simulation. EUROSIM (2016)
21. Kwiatkowska, M., Mehmood, R.: Out-of-Core solution of large linear Systems of Equations Arising from stochastic modelling. In: Process Algebra and Probabilistic Methods: Performance Modeling and Verification, pp. 135–151. Springer, Berlin (2002)
22. Kwiatkowska, M., Mehmood, R., Norman, G., Parker, D.: A symbolic out-of-core solution method for Markov models. Electr. Notes Theor. Comput. Sci. **68**, 589–604 (2002)
23. Mehmood, R.: A Survey of Out-of-Core Analysis Techniques in Stochastic Modelling. University of Birmingham, UK (2003)
24. Mehmood, R.: Serial disk-based analysis of large stochastic models. In: Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.-P., Siegle, M. (eds.) Validation of Stochastic Systems: A Guide to Current Research, pp. 230–255. Springer, Berlin (2004)
25. Mehmood, R., Crowcroft, J., Elmirghani, J.M.H.: A parallel implicit method for the steady-state solution of CTMCs. In: 14th IEEE International Symposium on Modeling, Analysis, and Simulation, pp. 293–302 (2006)
26. Mehmood, R., Parker, D., Kwiatkowska, M.: An Efficient BDD-Based Implementation of Gauss-Seidel for CTMC Analysis. University of Birmingham, UK (2003)
27. Mehmood, R., Parker, D., Kwiatkowska, M.: An Efficient Symbolic Out-of-Core Solution Method for Markov Models., University of Birmingham, UK (2003)
28. Magoulès, F., Ahamed, A.-K.C.: Alinea: an advanced linear algebra library for massively parallel computations on graphics processing units. Int. J. High Perform. Comput. Appl. **29**, 284–310 (2015)
29. Muhammed, T., Mehmood, R., Albeshri, A., Katib, I.: UbeHealth: a personalized ubiquitous cloud and edge-enabled networked healthcare system for smart cities. IEEE Access. **6**, 32258–32285 (2018)
30. Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C.: GPU computing. Proc. IEEE. 879–899 (2008)
31. Fevgas, A., Daloukas, K., Tsompanopoulou, P., Bozanis, P.: Efficient solution of large sparse linear systems in modern hardware. In: 2015 6th International Conference on Information, Intelligence, Systems and Applications (IISA), pp. 1–6 (2015)
32. Kirk, D.B., Hwu, W.M.W.: Programming Massively Parallel Processors: A Hands-on Approach (2013)
33. Cheng, J., Grossman, M., McKercher, T.: Professional CUDA C Programming. Wiley, New York (2014)
34. NVIDIA: Pascal GPU Architecture | NVIDIA. https://www.nvidia.com/en-us/data-center/pascal-gpu-architecture
35. NVIDIA: NVIDIA Tesla P100 Whitepaper (2016)
36. NVIDIA: NVIDIA Tesla V100. https://www.nvidia.com/en-us/data-center/tesla-v100/?ncid=van-tesla-v100
37. NVIDIA: History of NVIDIA – From Graphics Cards to Mobile Processors. http://www.nvidia.co.uk/object/corporate-timeline-uk.html
38. Nvidia: Nvidia CUDA C Programming Guide Version 4.2 (2012)
39. Bell, N., Garland, M.: Efficient Sparse Matrix-Vector Multiplication on CUDA (2008)
40. Nvidia: Profiler User's Guide
41. Saad, Y.: SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations Version 2 (1994)
42. Bell, N., Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis—SC '09, p. 1 (2009)
43. Liu, W., Vinter, B.: CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. Arxiv, Ithaca, NY (2015)
44. Guo, P., Lee, C.W.: A performance prediction and analysis integrated framework for SpMV on GPUs. In: Procedia Computer Science, pp. 178–189. The Author(s), (2016)

m.thaha.h@ieee.org

45. Fujita, M., McGeer, P.C., Yang, J.C.-Y.: Multi-terminal binary decision diagrams: an efficient data structure for matrix representation. Formal Meth. Syst. Design. **10**, 149–169 (1997)
46. Maggioni, M., Berger-Wolf, T.: Optimization techniques for sparse matrix-vector multiplication on GPUs. J. Parallel Distrib. Comput. **93-94**, 66–86 (2016)
47. Filippone, S., Cardellini, V., Barbieri, D., Fanfarillo, A.: Sparse matrix-vector multiplication on GPGPUs. ACM Trans. Math. Softw. **43**, 1–49 (2017)
48. Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., Demmel, J.: Optimization of sparse matrix-vector multiplication on emerging multicore platforms—long version. Parallel Comput. **35**, 178–194 (2009)

m.thaha.h@ieee.org