



## Article

# SURAA: A Novel Method and Tool for Loadbalanced and Coalesced SpMV Computations on GPUs

Thaha Muhammed <sup>1</sup>, Rashid Mehmood <sup>2,\*</sup>, Aiiad Albeshri <sup>1</sup> and Iyad Katib <sup>1</sup>

<sup>1</sup> Department of Computer Science, King Abdulaziz University, Jeddah 21589, Saudi Arabia; m.thaha.h@ieee.org (T.M.); aalalbeshri@kau.edu.sa (A.A.); iakatib@kau.edu.sa (I.K.)

<sup>2</sup> High Performance Computing Center, King Abdulaziz University, Jeddah 21589, Saudi Arabia

\* Correspondence: RMehmood@kau.edu.sa

Received: 22 January 2019; Accepted: 27 February 2019; Published: 6 March 2019



**Abstract:** Sparse matrix-vector (SpMV) multiplication is a vital building block for numerous scientific and engineering applications. This paper proposes SURAA (translates to *speed* in arabic), a novel method for SpMV computations on graphics processing units (GPUs). The novelty lies in the way we group matrix rows into different segments, and adaptively schedule various segments to different types of kernels. The sparse matrix data structure is created by sorting the rows of the matrix on the basis of the nonzero elements per row (*npr*) and forming segments of equal size (containing approximately an equal number of nonzero elements per row) using the Freedman–Diaconis rule. The segments are assembled into three groups based on the mean *npr* of the segments. For each group, we use multiple kernels to execute the group segments on different streams. Hence, the number of threads to execute each segment is adaptively chosen. Dynamic Parallelism available in Nvidia GPUs is utilized to execute the group containing segments with the largest mean *npr*, providing improved load balancing and coalesced memory access, and hence more efficient SpMV computations on GPUs. Therefore, SURAA minimizes the adverse effects of the *npr* variance by uniformly distributing the load using equal sized segments. We implement the SURAA method as a tool and compare its performance with the de facto best commercial (cuSPARSE) and open source (CUSP, MAGMA) tools using widely used benchmarks comprising 26 high *npr variance* matrices from 13 diverse domains. SURAA outperforms the other tools by delivering 13.99x speedup on average. We believe that our approach provides a fundamental shift in addressing SpMV related challenges on GPUs including coalesced memory access, thread divergence, and load balancing, and is set to open new avenues for further improving SpMV performance in the future.

**Keywords:** sparse matrix-vector multiplication (SpMV); high performance computing (HPC); graphics processing units; general-purpose computing on graphics processing units (GPGPUs); iterative methods; data analysis; sparse matrix storage; load balancing; coalesced memory access; thread divergence; Freedman–Diaconis rule

## 1. Introduction

Sparse Linear algebra is vital to scientific computations and various fields of engineering and thus has been included among the seven dwarfs [1] by the Berkeley researchers. Among the sparse numerical techniques, iterative solutions of sparse linear equation systems can be considered as of prime importance due to its application in various important areas such as solving finite differences of partial differential equations (PDEs) [2–4], high accuracy surface modelling [5], finding steady-state and transient solutions of Markov chains [6–8], probabilistic model checking [9–11], solving the time-fractional Schrödinger equation [12], web ranking [13–15], inventory control and manufacturing systems [16], queuing systems [17–23], fault modelling, weather forecasting, stochastic automata

networks [24,25], communication systems and networks [26–31], reliability analysis [32], wireless and sensor networks [33–37], computational biology [38], healthcare [27,39,40], transportation [41,42], natural language processing [43], operations research [23], reliability analysis [44,45], big data [46,47], and smart cities [48,49].

Sparse Matrix-Vector (SpMV) multiplication is the key operation of iterative methods for solving linear equation systems [50]. Iterative solvers such as Jacobi and Gauss–Seidel mainly consist of multiple iterations of Sparse Matrix-Vector computations, whereas a single iteration in Krylov solvers includes several SpMV operations. Sparse Matrix-Vector (SpMV) Multiplication is a Level-2 Basic Linear Algebra Subprograms (BLAS) operation between a sparse matrix and a dense vector and can be represented mathematically as  $y = A \times x + b$  [51]. Other sparse computations such as eigenvalue problems ( $Ax = \lambda x$ ) also require a large number of SpMV operations.

Implementation of parallel iterative solvers on different kinds of graphics processing units (GPUs) has numerous issues. Specialized storage structures are used to improve the performance of SpMV. These structures have design issues for translating it to GPUs. The major issues include coalesced memory access to both the sparse matrix  $A$  and the vector  $y$ , load balance among array threads and warps, thread divergence among a warp of threads, performance variance based on the structure of the sparse matrices, and the amount of memory access required for computations. These issues are due to the irregular sparsity structure of the matrix, which results in poor performance. Researchers have suggested several storage formats to improve SpMV on GPUs (see Section 3). Leading sparse libraries such as CUSP [52] and cuSPARSE [53] provide a number of sparse storage techniques including Coordinate (COO), Compressed Sparse Rows (CSR), ELLPACK (ELL), Hybrid (HYB), and Diagonal (DIA), and associated BLAS operations. However, the performance depends upon the sparsity structure and, due to the diversity in the sparsity pattern, they do not perform efficiently for all matrices. The burden of using an appropriate format for a given matrix usually lies with the user. Nevertheless, SpMV computations deliver low throughput due to the reasons mentioned above.

In this article, we propose SURAA (an Arabic word meaning speed), a novel method for SpMV computations on GPUs that addresses the major challenges of SpMV computations on GPUs, i.e., load balanced execution, coalesced memory access, and redundant computations. The novelty lies in the way we group matrix rows into different segments, and the way we dynamically schedule various segments to different types of kernels. The sparse matrix data structure is created by sorting the rows of the matrix on the basis of the nonzero elements per row ( $npr$ ) and forming segments of equal size (containing approximately an equal number of nonzero elements per row) using the Freedman–Diaconis rule [54,55]. To the best of our knowledge, no other work exists that used the Freedman–Diaconis rule for SpMV computations the way we used, or used adaptive kernels the way we have done in this paper. The segments are assembled into three groups based on the mean  $npr$  of the segments. For each group, we use multiple kernels to execute the group segments on different streams. Hence, the number of threads to execute each segment is adaptively chosen. Dynamic parallelism available in Nvidia GPUs (Major Version 3.5, NVIDIA Corporation, Santa Clara, CA, USA) is utilized to execute the group containing segments with the largest mean  $npr$ , providing improved load balancing and coalesced memory access, and hence more efficient SpMV computations on GPUs. Note that the row segments in SURAA are dynamically scheduled by the dynamic kernel and are executed at the same time using the hardware streams on the GPU. The use of multiple streams provides higher performance by hiding the kernel launch overhead. This could be considered as reducing the kernel launch overhead, with the exception that the number of dynamic kernels that can be executed on GPU in parallel depends on the number of available Streaming Processors because the number is fixed for a given GPU. The number of segments that are added to the dynamically executed group is tunable. In our experiments, we find that depending upon the structure of the matrices, tuning the size of dynamically executed group enhances the performance of the scheme; see Section 4.4.

We implement the SURAA method as a tool and compare its performance with the de facto best commercial (cuSPARSE) and open source (CUSP) tools using widely used benchmarks comprising 26 matrices from 13 diverse domains. A total of seven other SpMV storage and computation techniques have been compared with SURAA. We have used three widely used performance metrics to evaluate SURAA and other techniques; throughput in Giga Floating Point Operations per second (GFLOP/s), speedup, and effective memory bandwidth (GB/s). Each scheme is also evaluated in terms of their performance with increasing *npr variance* (variance in the number of non-zeros per row) and *nnz* (the total number of non-zeros in the matrix).

On average, SURAA outperforms all other seven SpMV computation methods by achieving an average speedup of  $40\times$  for SELL-P,  $29.89\times$  against WPK1,  $12.98\times$  for CSR,  $11.33\times$  for BSR,  $1.50\times$  for HYB,  $1.15\times$  for WPK2, and  $1.1\times$  for CSR5. SURAA provides the best throughput for 20 matrices out of the 26 matrices in the benchmark dataset. The average collective speedup of SURAA against all the other schemes is  $13.99\times$ . The highest speedups achieved by SURAA against other schemes for any single matrix are  $562\times$ ,  $531.56\times$ ,  $147\times$ ,  $26.70\times$ ,  $3.06\times$ ,  $1.97\times$ , and  $1.94\times$  against SELL-P, WPK1, CSR, BSR, HYB, WPK2, and CSR5, respectively. We repeat for clarity that these are SURAA's peak performance values against specific storage schemes for any single matrix. SURAA provides the highest *mean* throughput (GFLOP/s) of 15.75, followed by 14.28 for CSR5. SURAA also leads the table with the highest *max* throughput value of 36.70 GFLOP/s, followed by 26.8 for SELL-P. SURAA also provides the highest effective memory bandwidth of 226.25 GB/s compared to the second best 160.75 GB/s for SELL-P. SURAA also demonstrates a much lower setup overhead, equal to five SpMV execution time, compared to 69 and 95.5 SpMV executions for SELL-P and CSR5.

An important aspect of our analysis reveals (see Section 5.4, Page 24) that SURAA comparatively delivers higher performance with the increase in the *npr variance* and *nnz*. SURAA achieved the highest throughput (GFLOP/s) among all the schemes for the highest *npr variance* matrices.

We note that the speedup of SURAA against CSR5 and WPK2 was small, however, SURAA provided the highest throughput for 20 out of 26 matrices. More importantly, none of the schemes except SURAA delivered consistently high performance on all the performance benchmarks. SURAA code can be found at the GitHub online repository (see: <http://github.com/stormvirux/suraa>). The rest of the paper is organized as follows. In Section 2, we discuss the background materials related to this paper. In Section 3, we discuss the state of the art. In Section 4, we describe our proposed technique. The experimental results from the implementation and analysis is given in Section 5. Section 6 concludes and summarizes future work.

## 2. Background

### 2.1. Sparse Storage Schemes

Several sparse storage schemes have been proposed by researchers to improve the performance of sparse matrix computations. Many of them have been designed for matrices with particular sparsity structure or from a given application domain—for example, see [56]—which proposes a compact representation of sparse matrices arising from Markov Chains. However, there are a few general storage formats that are frequently used to store sparse matrices. Most of the sparse matrices are stored using one of these formats. We shall have a brief discussion regarding these formats in the following subsections. Figure 1 shows an example matrix  $A$  that we shall use to illustrate various sparse storage schemes. Implementation of these storage mechanisms and associated SpMVs can be found in cuSPARSE [53] and CUSP [52] libraries.

$$\begin{array}{c}
 \xleftrightarrow{n} \\
 \left[ \begin{array}{cccccc}
 1 & 0 & 2 & 0 & 0 & 3 \\
 0 & 0 & 0 & 0 & 4 & 0 \\
 5 & 6 & 0 & 0 & 0 & 7 \\
 0 & 0 & 0 & 8 & 0 & 9 \\
 0 & 10 & 0 & 11 & 0 & 0
 \end{array} \right] \\
 \updownarrow m
 \end{array}$$

**Figure 1.** The dense representation of matrix  $A$ .

### 2.1.1. Coordinate Storage (COO)

The coordinate (COO) [57] format is a simple sparse storage scheme. Three arrays, *row*, *col*, and *val* are used to store the row indices, column indices, and values of the nonzero elements in the matrix. Figure 2 illustrates the COO representation of the example matrix  $A$ . Each GPU thread will compute the product of a non-zero with the associated element in vector  $x$ . Further reduction is required to sum the partial products to achieve the final results. The reduction procedure creates an overhead that slows down the SpMV.

val	1	2	3	4	5	6	7	8	9	10	11
row	0	0	0	1	2	2	2	3	3	4	4
col	0	2	5	4	0	1	5	3	5	1	3

**Figure 2.** The coordinate storage (COO) representation of matrix  $A$ .

### 2.1.2. Compressed Sparse Row (CSR)

The compressed sparse row (CSR) format like the COO format stores the column indices and nonzero values in arrays named *col* and *val* explicitly. However, a third array of row pointers, *ptr*, is used to store the row pointers. For an  $m \times n$  matrix, *ptr* has length  $m + 1$  and stores the offset into the  $i$ th row in *ptr*[ $i$ ]. The last entry in *ptr*, which would otherwise correspond to the  $(m + 1)$ -st row, stores nnz, the number of nonzeros in the matrix. Figure 3 illustrates the CSR representation of an example matrix. SpMV for CSR based matrix can be performed using two techniques, CSR scalar and CSR vector [58]. In CSR scalar, one thread is assigned per row for SpMV operation. Each thread will compute the products and sum up the products for each row. However, due to workload imbalance and non-coalesced memory access, CSR scalar has poor performance. CSR vector is an improvement to CSR scalar, in which a warp (32 threads) is assigned to a row to perform SpMV. However, variance in the number of non-zero elements per row results in idle threads, which causes load imbalance, which leads to poor performance.

val	1	2	3	4	5	6	7	8	9	10	11
col	0	2	5	4	0	1	5	3	5	1	3
ptr	0	3	4	7	9	11					

**Figure 3.** The compressed sparse row (CSR) representation of matrix  $A$ .

### 2.1.3. Hybrid ELL/COO (HYB)

The hybrid format was proposed by Bell and Garland to eliminate the issues with ELL [58]. The major disadvantage of ELL was the performance degradation on varying number of non-zeros per row of the matrix. The HYB storage scheme divides the matrix into two parts, a dense part and a sparse

part. In the dense part, we store the typical number of non-zeros per row in the ELL storage format and the rest of entries belonging to rows with higher non-zeros in the COO format. The size of the ELL matrix needs to be determined normally from the input matrix. We can add a  $k$ th column to the ELL part if at least one third part of the matrix rows contains  $k$  non-zeroes. The remaining non-zeros in the rows are stored using the COO format. Another way to compute  $k$  is to use the frequency of the non-zeros per row as seen in CUSP implementation. The remaining non-zeros in the rows are stored using the COO format. An example of HYB storage for a sparse matrix  $A$  is shown in Figure 4. Hence, this storage technique requires two 2D arrays of size  $m \times k$  for storing the ELLAPCK part, where  $k$  is the number of selected columns, and three separate arrays to store the COO part. If the actual matrix is of size  $m \times n$ , then the value array of the COO part would have a dimension of  $(n - k)$ .

$$A = \begin{bmatrix} 1 & 0 \\ 3 & 4 \\ 6 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 12 & \end{bmatrix} \quad \begin{array}{l} \text{val} = [12 \quad 8] \\ \text{col} = [5 \quad 4 \quad 15] \\ \text{row} = [1 \quad 13 \quad 6] \end{array}$$

Figure 4. Hybrid ELL/COO (HYB) representation of matrix  $A$ .

## 2.2. Dynamic Parallelism

In this section, we give a general introduction to dynamic parallelism which was introduced in CUDA 5. It introduced nested parallelism with the help of nested kernel calls from the GPUs. In the earlier GPU parallel computing model, kernels were invoked in sequence from the host side. In order to achieve higher performance, each of these kernels had to expose ample parallelism.

Dynamic parallelism has been known to provide excellent performance for irregular applications such as graph-based applications and simulations [59,60]. Many graph-based applications produce dynamic sparse matrices/graphs that change during the execution. Graph algorithms such as PageRank, Hyperlink-Induced Topic Search (HITS), Random walk with restart is dominated by SpMV and has been shown to perform well with Dynamic Parallelism. The use of Dynamic parallelism simplifies the code and results in code near to the high-level description, which simplifies the development of algorithms.

Dynamic parallelism provides the ability to launch kernels from a kernel on the device. Each thread of the main kernel can launch a new kernel. A coarse-grained kernel can launch finer-grained kernels to do work where needed, which reduces the execution and data transfer as launch configuration decisions are made on runtime from the device. Algorithms requiring constructs that are difficult to program using a single level of parallelism, such as recursion, irregular loop structure, etc. [61], can be programmed in a finer way using Dynamic Parallelism.

Dynamic parallelism is only supported by devices of compute capability 3.5 and higher. It has an overhead for invocation similar to general CUDA kernels. An overhead is caused by runtime's execution tracking and management software and may result in decreased performance. The overhead is incurred when the applications link against device run-time library [61]. It could be possible to hide kernel launch overhead by a developer, but, in our work, we do not try to explicitly overlap or hide the kernel launch overhead. In this paper, it is not our intention to discuss, compare and/or quantify dynamic parallelism overhead with the counterpart approaches. Some of the works on the comparison and discussion of dynamic parallelism overhead are [59,62–64].

## 3. Literature Survey

There are many studies that have attempted to improve the performance of Sparse Matrix vector multiplications on GPUs. Many of these have been developed to improve the performance of a specific sparsity structure. In this section, we shall discuss some of the approaches for SpMV computations on GPUs.



Jagged Array Diagonals (JAD) format [65,66] can be considered as an optimization of ELL format on GPUs. Initial preprocessing is required to store a sparse matrix in JAD format. JAD provides better performance in SpMV kernel than ELL, but the occupancy in JAD is lesser than ELL. Abu-Sufah and Abdel Karim [67] introduced an improvement to the Transpose Jagged Diagonal Storage (TJDS) by the use of blocking technique, which is called Blocked Transpose Jagged Diagonal Storage (BTJDS). This technique does not achieve a good memory bandwidth utilization as compared to HYB and ELL.

Choi et al. [68] implemented the blocked version of CSR algorithm (BCSR) on GPU. Furthermore, they also propose a second sparse storage structure, which is the blocked version of ELL called the blocked ELLPACK (BELLPACK). ELL-R was introduced in [69] to improve the performance of the ELL storage format on GPUs. It has an additional data structure that helps in reducing the performance degradation due to thread divergence caused by non-zero entries in the ELL storage format. Kreutzer et al. [70] propose a sparse matrix storage scheme to reduce the space overhead of extra padded zeroes in the ELL format and improve the performance of SpMV on GPUs. The major disadvantage of this technique is that the multiplication by permutation destroys the dense blocks and diagonal elements which prevents cache reuse and degrades the performance.

ELLR-T is based on GPU kernel modification of ELL-R [69,71]. In ELL-R format, each row is handled by a thread during SpMV, whereas in ELLR-T, T number of threads operate on the row at the same time, similar to vector CSR (In vector CSR 32 threads operate on a row). Here, T can be 1, 2, 4, 8, 16, or 32. However, in this technique, the authors do not propose a way for selecting best T based on the input sparse matrix, rather they leave it to the user to select by experimentation. Dziekonski et al. [72] proposed a sparse matrix storage scheme called Sliced ELLR-T. The storage scheme was designed specifically to solve complex-valued sparse linear equations in computational electromagnetics. The proposed technique is based on Sliced ELL [73] and ELLR-T storage schemes. The amount of space required by this scheme is less than ELL-R and ELLR-T but more than CSR. The bigger the matrix, the better the performance of the SpMV kernel using this scheme.

Padded Sliced ELLPACK (SELL-P) [74] is derived from SELLPACK [73] and SELL-C- $\sigma$  [75], wherein the rows with a similar number of nonzero elements are gathered into slices, thus minimizing the storage overhead as compared to ELLPACK scheme. By assigning multiple threads per row, as in ELLR-T [69], they modify SELL-C- $\sigma$  by padding rows with zeroes such that the row length of each slice becomes a multiple of the number of threads assigned to each row. The implementation of SELL-P is available in the MAGMA library. The implementation of the sorting in the MAGMA library is performed on the CPU, which increases the overhead.

All the schemes discussed until now are focused on the storage of the sparse matrices and they mostly use just one kernel or fixed number of threads as opposed to SURAA.

Maggioni and Berger-Wolf [76] proposed a storage scheme based on ELL called Adaptive ELL that balances the non-zero rows to GPU warps. In this technique, the computations on the nonzero elements are distributed on the warps (hardware level blocks) adaptively. Using a best-fit heuristic greedy policy, they try to fill the warps with the rows. They solve an optimization problem of maximizing the efficiency of each warp and minimizing the load unbalance among the warps. This is a finer level (lower level) optimization unlike our technique, which uses multiple parallel kernels and variable threads to perform scheduling on sorted segments and dynamic parallelism. Adaptive ELL also requires four extra data structures. Of these four extra data structures, three data structures will have the size of total working warps in the GPU and one will have the size of the number of rows in the matrix. Older GPUs, such as Tesla K20 and Tesla K40, have lower memory and hence this technique will lead to difficulty in processing larger matrices. However, many of the newer GPUs such as Pascal P100 and Volta V100 have 12 to 16 GBs of DRAM and can also access memory on other GPUs over high bandwidth buses.

Ashari et al. [77] propose a sparse storage scheme based on CSR called Adaptive Compressed Sparse Row (ACSR). In this scheme, the rows are grouped into bins, such that a bin consists of rows with an equal number of non-zeros per row. The rows of the sparse matrix are moved into bins

depending on the number of non-zeros per row. The binning is done based on powers of 2. Since the binning is not done on consecutive rows, it is termed as inter-binning. Each bin is either executed by a bin specific kernel or a row specific dynamic kernel. However, as their work is based on CUDA 5.0, in their technique, dynamic parent kernel does not process more than 2048 rows. Our work exploits virtual pooling and execute rows greater than 2048. SURAA is also different from their work in that it achieves coalesced memory access due to reordering the similar loads (rows with similar  $npr$ ) together in the memory rather than logically performing inter-binning. Moreover, we differ in the way we assemble matrix rows into different segments, and segments into groups, as we use the Freedman–Diaconis rule [54,55] to achieve a relatively balanced segment size. Consequently, we achieve higher levels of load balancing due to the way we devise row segments and allocate them to dynamic kernels. This decreases the idle number of threads and leads to better memory coalescing. The row segments in SURAA are dynamically scheduled by the dynamic kernel and are executed at the same time using the hardware streams on the GPU. The use of multiple streams provides higher performance by hiding the kernel launch overhead. However, the number of dynamic kernels that can be executed on GPU in parallel depends on the number of available Streaming Processors because these are fixed for a given GPU.

ELL-WARP is a variant of ELLPACK-R and padded JAD (JDS) developed for finite element unstructured grids by Wong et al. [78]. They propose two kernels, ELL WARP(K1) also known as WPK1 and ELL WARP v2 (K2) which is also known as WPK2. Both these kernels initially sort the rows by length. WPK1 then arranges rows into groups of warp size and pads accordingly and then it reorders the data within each warp in a column-major coalesced pattern. However, in WPK2, the rows are subdivided recursively beginning when the number of elements a thread should execute exceeds the prescribed threshold. The number of elements a thread should execute are continued to be subdivided until the number reaches below the threshold. This is done to efficiently process the larger rows as compared to WPK1.

Lu and Vinter [79] propose a storage format for sparse matrix-vector multiplication called CSR5. The 5 in the name indicates the number of data structures used to store the sparse matrices. The designed data storage scheme is insensitive to the sparsity of the data structure provides similar performances for sparse as well as dense matrices. This technique uses a combination of row block method [77] and segmented sum method [80]. The design objective is improved load balancing and reduction in overhead due to memory access and synchronization. The matrix transpose operations and the complexity causes overhead. Moreover, higher meta-data requirement increases the required memory for the representation. However, CSR5 does provide improved performance over the earlier SpMV techniques.

Hou et al. [81] proposed an auto-tuning framework for AMD APU platforms to find appropriate binning scheme and select appropriate kernel for each bin. The process of grouping rows with similar number of nonzeros together is referred to as binning by the authors. The proposed binning method is a coarse grained method with different granularities. The auto tuning technique selects the best granularity for a given matrix using machine learning techniques (C5.0). In their binning strategy they combine multiple neighboring rows to be a single virtual row. The granularity or the number of neighboring rows selected is determined by the auto-tuner. The autotuner hence needs to be trained with appropriate data for the proper feature selection and prediction which is non-trivial. Moreover, for irregular matrices with highly varying non-zero per row, a fixed granularity will require each bin requiring a different kernel. Executing large number of kernels at the same time (which is subject to the device capability) requires multiple streams which is hardware dependent and hence might degrade the performance if the number of bins increases. Unlike this work, we do not use machine learning; instead, we segment the matrix on the basis of Freedman–Diaconis rule [54,55]. The other major difference is the use of dynamic parallelism in our work and the way we adaptively schedule different kernels with varying number of threads, which addresses memory coalescing, thread divergence, and load balancing.

Flegar and Anzt [82] propose a load-balanced GPU kernel based on COO for computing SpMV product. The proposed kernel use a "warp vote function" along with a warp level segmented scan to avoid atomic collisions. It also uses an "oversubscribing" parameter to determine the number of threads allocated per each physical core. The three arrays in the COO scheme is sorted with respect to the row index to decrease the number of atomic operations such that elements with same row index require only one atomic operation.

The major issues that need to be addressed are improved coalesced memory access to the matrix, better load balanced execution of SpMV, higher memory bandwidth utilization, preprocessing requirements, utilization and improvement in cache usage, and reduction of thread divergence. A comparative analysis of the related approaches for SpMV computations with respect to these issues have been provided in Table 1. The schemes are listed in chronological order of their publications. We observe that no single work has comprehensively addressed all the issues.

From the discussions, it is clear that due to the variance in the number of nonzeros per row, and differences in the sparsity structure, a single kernel with fixed number of threads would not be able to guarantee the best performance for various matrix sizes, application domains, and sparsity structures. Hence, in this work, we propose the SURAA method that balances the load efficiently and achieves coalesced memory access to attain the best overall performance. This is due to the fact that sparse matrices have a range of diverse sparsity structures, and dynamic kernels (together with Freedman–Diaconis (FD) rule based segmenting) allow us to dynamically schedule the matrix rows based on the matrix properties enabling improved memory and computation layout, and in turn providing improved load balancing, memory coalescing, and thread divergence.

**Table 1.** Comparative analysis of various discussed features in discussed sparse matrix-vector (SpMV) multiplication techniques.

Work	Coalesced Access	Load Balanced	Mem. Bandwidth	Pre-Processing	Cache Locality	Thread Divergence
JAD <sup>1</sup> [65,66]	Yes	No	No	Yes	No	No
BCSR <sup>2</sup> [68]	No	No	Yes	No	No	No
ELL-R <sup>3</sup> [69]	Yes	No	Yes	No	No	No
TJDS <sup>4</sup> [67]	No	No	Yes	Yes	No	No
pJDS <sup>5</sup> [70]	Yes	Yes	Yes	Yes	No	No
ELLR-T <sup>6</sup> [69,71]	Yes	Yes	No	No	No	Yes
Sliced ELL <sup>7</sup> [73]	No	Yes	Yes	Yes	No	No
Sliced ELLR-T <sup>8</sup> [72]	No	Yes	Yes	Yes	No	Yes
SELL-C- $\sigma$ <sup>9</sup> [75]	No	Yes	Yes	Yes	No	No
SELL-P <sup>10</sup> [74]	No	Yes	Yes	Yes	No	No
Adaptive ELL <sup>11</sup> [76]	No	Yes	No	Yes	Yes	No
Adaptive CSR <sup>12</sup> [77]	Yes	No	No	No	Yes	Yes
Wong et al. [78]	Yes	Yes	No	Yes	No	No
CSR5 <sup>12</sup> [79]	Yes	Yes	Yes	Yes	No	No

<sup>1</sup> Jagged Array Diagonals. <sup>2</sup> Blocked Compressed Sparse Row. <sup>3</sup> Jagged Diagonals. <sup>4</sup> Transposed Jagged Array Diagonals. <sup>5</sup> Padded Jagged Diagonals. <sup>6</sup> Ellpack Rows Thread. <sup>7</sup> Sliced Ellpack. <sup>8</sup> Sliced Ellpack Rows-Thread. <sup>9</sup> Sliced Ellpack-C. <sup>10</sup> Sliced Ellpack-P. <sup>11</sup> Ellpack. <sup>12</sup> Compressed Sparse Row.

#### 4. SURAA: The Proposed Method and Tool

Load-balanced execution and coalesced memory access are the two crucial factors that we consider to improve the performance of SpMV. A load-balanced execution balances the workload of the threads and eradicates idle threads. When a group of consecutive threads accesses consecutive memory locations, we call it coalesced memory access. Coalesced memory access tremendously improves the throughput of data access from global memory and hence improves the SpMV performance.



The two major CSR based algorithms are the CSR scalar and the vector CSR (VCSR). The issue with scalar CSR is that it does not correctly balance workload as one thread is assigned per row. This results in imbalanced workload for the threads in a warp. The vector CSR rectifies this issue by assigning threads equal to the number of warps (32 threads) equal to closest power of the mean number of the non-zero per row on the GPU. Matrices that have a uniform number of non-zeros per row and are a multiple of a warp accelerates the SpMV. However, there are many matrices with less number of nnz per row, especially the matrices with the number of non-zeros per row less than sixteen. These matrices are not efficiently executed by VCSR as many GPU threads remain idle while the other threads in a warp are under execution. Therefore, warps that are assigned to rows with a substantial number of nonzeros per row lag behind other warps with a fewer number of the non-zero per row.

Hence, it is essential to consider the sparsity structure and coalesced memory access to derive benefit from the merits and avoid the flaws of both the scalar and vector CSR techniques. To this end, we develop a load balanced dynamic technique based on the CSR storage format called SURAA. SURAA uses sorting, segmentation, asynchronous kernels, streams, and dynamic parallelism. Sorting is used to bring all the rows with similar non-zero elements together and segmentation is used to divide the work into manageable blocks to distribute the workload evenly. Dynamic kernel enables us to launch multiple child kernels from within the parent kernel which can efficiently execute the sorted segments.

The terminologies and the data structures used are illustrated in Table 2. The format is very similar to the CSR format, except that we introduce an array named *perm* to store the sorted indices. *perm* is of size *m*. A temporary vector, *nprv*, of size *m* is generated to speed up the pre-processing (sorting), which stores the number of non zero elements per row. In the rest of the paper, we shall use the terminologies defined in the Table 2 for our discussions.

**Table 2.** The terminology and data structures used for matrix representation.

Structure	Details
<i>x</i>	The vector for multiplication
<i>m</i>	The number of rows in the matrix
<i>n</i>	The number of columns in the matrix
<i>y</i>	The result vector
<i>nnz</i>	The total number of non zeros
<i>npr</i>	The number of non zeros per row
<i>nprv</i>	Vector for storing the non zeros per row
<i>val</i>	Vector to store the non zeros of the matrix
<i>row_ptr</i>	Vector to store the row pointers to the matrix
<i>col_idx</i>	Vector to store the column indices to the non zeros
<i>perm</i>	Vector for storing the indices of the sorted matrix

The SURAA tool is divided into two phases—(a) the setup phase and (b) the execution phase—comprising a total of six algorithms (details to follow). Algorithm 1 shows our main driver program which invokes both the setup phase and the execution phase. We shall discuss the driver algorithm and the two phases in the next subsections.

**Algorithm 1** SURAA\_Master\_Program

---

**Input:**  $A_{csr}(val, row\_ptr, col\_idx)$ ,  $vector : x$ ,  $vector : perm$ ,  $vector : nprv$   
**Output:**  $vector : y$

```

1: function MAIN
2:    $G_1, G_2, G_3 \leftarrow \text{Setup\_Phase}(A_{csr}, x)$ 
3:   for Each segment  $s_i \in G_3$  do
4:      $scalarSpMV \ll grid_{s_i}, stream_i \gg (s_i)$ 
5:   end for ▷ Asynchronous returns for the kernel
6:   for Each segment  $s_j \in G_2$  do
7:      $Y \leftarrow (nnz(s_j) \gg 1) + 31$ 
8:      $Y \leftarrow Y \gg 5$ 
9:      $Y \leftarrow Y \ll 5$  ▷ Thread per blocks as a multiple of 32
10:     $vectorSpMV \ll grid_{s_j}, stream_j \gg (s_j, Y)$ 
11:  end for ▷ Asynchronous returns for the kernel
12:   $DynamicSpMV \ll grid_{G_1} \gg (G_1)$ 
13: end function

```

---

**4.1. Setup Phase**

Algorithm 2 depicts the setup phase where the matrix is sorted, segmented, and grouped into various groups. The matrix is sorted row by row in the descending order of the number of non-zero elements per row. The rows with the same number of number of non-zeros are further sorted based on the column index of the first non-zero element, which further balances the load of vector  $x$ . The sorting can be done on the GPUs or the CPUs. In our work, we use a GPU based parallel radix sort to perform the sort. The function  $radixSort\_Key(\dots)$  is used to perform the sorting of the  $nprv$  vector. It sorts the  $nprv$  vector and stores the permutations of the original  $nprv$  vector in the  $perm$  vector and the associated  $npr$  values in the  $nprSort$  vector. The input matrix is then rearranged using the  $perm$  vector.

**Algorithm 2** SURAA\_Setup\_Phase

---

**Input:**  $A_{csr}(val, row\_ptr, col\_idx)$ ,  $vector : x$ ,  $vector : perm$ ,  $vector : nprv$   
**Output:**  $set : G_1$ ,  $set : G_2$ ,  $set : G_3$

```

1: function SETUP_PHASE
2: Initialisation :  $perm \in seq[0, m]$ 
3:    $nnzpr \leftarrow$  non-zero elements per row
4:    $nprSort, perm \leftarrow radixSort\_Key(nprv, perm)$  ▷ Sort Descending
5:   Rearrange  $A_{csr}$  using  $perm$ 
6:    $q12 \leftarrow nprSort[m * 1/4];$ 
7:    $q34 \leftarrow nprSort[m * 3/4];$ 
8:    $Seg\_size \leftarrow (2 \times (q34 - q12)) / \sqrt[3]{m}$ 
9:   for Each segment  $s$  do
10:    if  $32 < nprv[s_i] < max\_child\_threads$  then
11:       $G_1 \leftarrow G_1 \cup s_i$ 
12:    else if  $nprv[s_i] \geq 32$  then
13:       $G_2 \leftarrow G_2 \cup s_i$ 
14:    else
15:       $G_3 \leftarrow G_3 \cup s_i$ 
16:    end if
17:  end for
18: end function

```

---

Secondly, the sorted matrix is divided into various segments. The division into segments is based on the Freedman–Diaconis rule [54,55], which is given below:

$$Seg\_size \leftarrow 2 \times (IQR(nprv) / \sqrt[3]{m}), \quad (1)$$

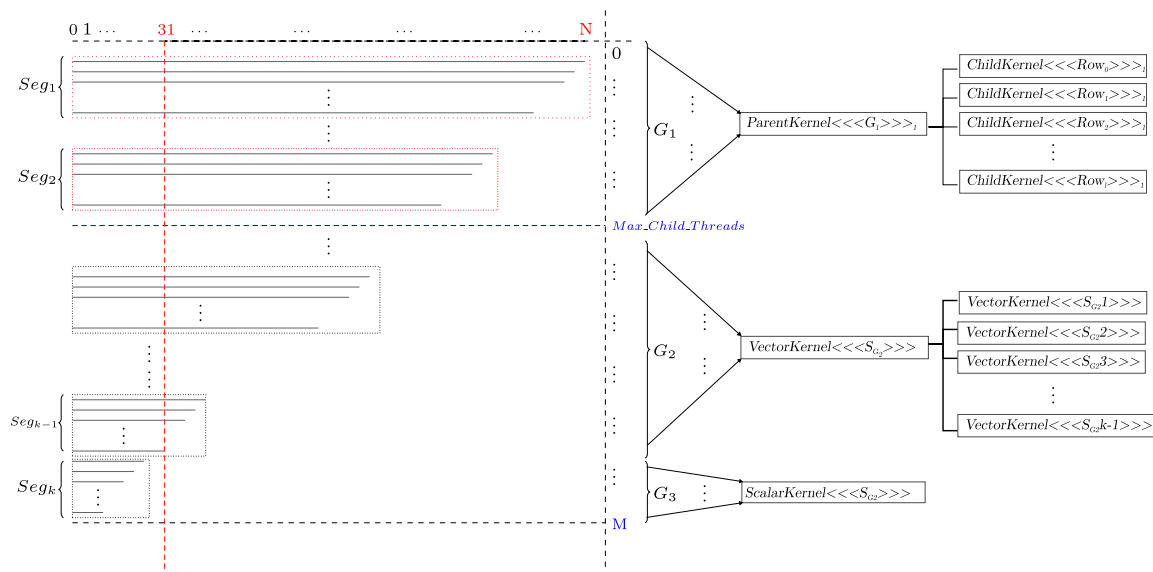
where  $Seg\_size$  is the size of each segment,  $IQR$  is the Interquartile range,  $nprv$  has been defined already, and  $m$  is the number of rows in the matrix. Moreover,  $IQR(nprv) = q_{34} - q_{12}$ , where  $q_{34}$  and

$q_{12}$  are the third and first quartiles of  $nprv$ , respectively. In our algorithm, we calculate the quartiles of the sorted  $npr$  array ( $nprSort$ ). Since the matrix is pre-sorted, we can get the optimal size of the segments as well as the number of segments using the Freedman–Diaconis rule for each matrix with  $\mathcal{O}(1)$  complexity.

In the third step, the segments are allocated to various groups depending upon the average number of non-zero elements per row in each segment. This is also depicted in the left part of Figure 5. The number 0 to  $N$  indicates the  $npr$  for each row and the numbers 0 to  $M$  indicate the number of rows in the matrix. The red dashed line in the figure indicates the size of the warp (0 to 31) for GPUs, which is used as a threshold in our method. Since the matrix is sorted in the descending order of  $npr$ , we add each segment  $s_i$  from the beginning of the matrix to Group 1 ( $G_1$ ) until the mean number of non-zero per row for a segment becomes less than 32 or the total number of rows in  $G_1$  exceeds  $max\_child\_threads$  (The number of child grids in the queue ready for execution). Since the segment size might not be a multiple of group size, if  $n \times segment\_size > max\_child\_threads$ , we will only include  $(n - 1) \times segment\_size > max\_child\_threads$ . The rest of the segments which has mean  $npr$  greater than or equal to 32 is allocated to  $G_2$ . The remaining segments that have a mean number of non-zero per row less than 32 are added to  $G_3$ .

The last segment might not have the same number of rows as the other segments and hence is executed separately. All the segments in  $G_3$  are executed asynchronously on different streams using a scalar CSR kernel, whereas, all the segments in  $G_2$  are executed by a modified vector CSR kernel asynchronously. Finally, we employ dynamic parallelism based kernel to run the segments in  $G_1$ , which we shall discuss in greater details in the following subsections.

After setting up the matrix, we proceed to the execution phase where we schedule and allocate various segments to different kernels.



**Figure 5.** SURAA: Segmentation, grouping, and allocation of kernels among the three groups for the proposed scheme.

## 4.2. Execution Phase

### 4.2.1. Dynamic Parallel Kernel

Figure 5 further illustrates the allocation of various segments to different kernels. The segments with rows containing larger  $npr$  are executed as a part of the dynamic kernel, whereas the other segments are executed based on the average length of the rows in the segment. The segments that form the part of the dynamic kernel have been shown in red. In the figure, the segments from  $Seg_1$  to  $Seg_i$  have  $npr$  less than or equal to  $max\_child\_threads$  (The blue dashed line shows the  $max\_child\_threads$

parameter) and hence will be grouped together in  $G_1$ , which shall be executed by the dynamic kernel. The segment  $Seg_k$  has mean  $npr$  less than 32 and hence is executed by the scalar CSR based kernel. The rest of the segments, between  $Seg_2$  and  $Seg_k$  are grouped together into  $G_2$  and executed by the vector kernel.

Algorithm 3 is the parent algorithm for the dynamic kernel that will execute on the segments of  $G_1$ . This parent kernel assigns child kernels, depicted in Algorithm 4, to each row. Each thread of the parent kernel will work on a row by invoking child kernels. Multiple warps (depending upon the row size) of the child algorithm will execute the row assigned to them. As Multiple warps are involved for a row in the child kernel, we require a reduction within each warp as well as among the multiple warps. We use the CUDA function `_shfl_down_sync` to compute the partial sums of the threads within a warp. The function `_shfl_down_sync` is faster than the `_shfl_down` function and is available in the CUDA version 9.0 and later. Each warp stores their respective partial sums in the shared memory (`shared_mem`) and a final reduction of the shared memory is performed to get the sum.

The size of the grids for the child kernels is dependent upon the number of non-zeros in each row. In our example shown in Figure 5, the parent kernel will then invoke child kernels for all the segments from  $Seg_1$  to  $Seg_2$ , i.e., for each row in  $G_1$ , a child kernel is assigned to process the row. `ChildKernel <<< Row 0 >>>` executes the first row in the first segment and `ChildKernel <<< Row 1 >>>` executes on the second row of the first segment and so on throughout  $G_1$ . A segment specific vector kernel is used to execute each segment at the same time using various cuda streams for each segment in  $G_2$ . In addition, the last two segments,  $Seg_{k-1}$  and  $Seg_k$ , are added to  $G_3$  and executed using a scalar CSR based algorithm.

---

#### Algorithm 3 SURAA\_Dynamic\_Kernel\_ $G_1$

---

**Input:**  $A_{csr}(val, row\_ptr, col\_idx)$ ,  $vector : x$ ,  $vector : y$ ,  $vector : perm$ ,  $Thread\_per\_row : Y$

**Output:**  $vector : y$

```

1: function DYNAMICSPMV
2:    $tid \leftarrow global\_thread\_Id$ 
3:   if  $tid < rows(G_1)$  then
4:      $row\_id \leftarrow G_1\_rows$ 
5:      $row\_cur \leftarrow row\_ptr[row\_id]$ 
6:      $row\_next \leftarrow row\_ptr[row\_id + 1]$ 
7:      $nnz \leftarrow row\_next - row\_cur$ 
8:      $threads\_per\_block \leftarrow (nnz >> 1) + 31$ 
9:      $threads\_per\_block \leftarrow threads\_per\_block >> 5$ 
10:     $threads\_per\_block \leftarrow threads\_per\_block << 5$ 
11:     $stream \leftarrow newCudaStream$ 
12:    ChildKernel<<<1,threads_per_block,stream>>>
      (  values[row_cur..row_next),
        col_idx[row_cur..row_next),
        nnz, x, y, perm)
13:   end if
14: end function

```

---

In the Child kernel, initially we calculate the thread Id ( $tid$ ), lane Id ( $lid$ ), and warp Id ( $wid$ ) to access the data structures. Each child thread computes the partial sum and stores in the `sum` vector. Segmented parallel reduction using the shuffle instruction (`_shfl_down_sync`) is used to accumulate the partial results. The partial sum is then stored in the shared memory (`shared_mem`) for processing of all the relevant threads. We perform one more reduction to accumulate the final sum and store it in the result vector ( $y$ ).

**Algorithm 4** SURAA\_Child\_Kernel**Input:**  $A_{csr}(val, row\_ptr, col\_idx)$ ,  $vector : x$ ,  $vector : y$ ,  $vector : perm$ **Output:**  $vector : y[n]$ 

```

1: function CHILDKERNEL
2: Initialisation :  $sum \leftarrow 0$ ,  $shared\_mem[0..31] \leftarrow 0$ 
3:    $tid \leftarrow thread\_Id$ 
4:    $tid \leftarrow thread\_Id$ 
5:    $lid \leftarrow tid \% 32$ 
6:    $threads\_total \leftarrow blockDim$ 
7:    $wid \leftarrow tid / 32$ 
8:   for  $i = tid; i < nnz; i += threads\_total$  do
9:      $sum += values[i] \times x[col\_idx[i]]$ 
10:  end for
11:  for  $i = 16; i > 0; i = i >> 2$  do
12:     $sum += \_\_shfl\_down\_sync(0xffff, sum, i)$ 
13:  end for
14:  if  $lid == 0$  then
15:     $shared\_mem[wid] \leftarrow sum$ 
16:  end if
17:   $sum \leftarrow \sum_{i=0}^{32} shared\_mem[i]$ 
18:  if  $tid == 0$  then
19:     $y[0] \leftarrow sum$ 
20:     $row\_id \leftarrow perm[tid]$ 
21:  end if
22: end function

```

## 4.2.2. Scalar Kernel

Algorithm 5 illustrates the segment based scalar SpMV execution. SpMV for the segments in  $G_3$  is done by this algorithm. In this algorithm, one thread is assigned per each row. For each row, the threads will compute the products of the non-zero elements with the corresponding vector  $x$ . Each thread will finally sum the products of each row and write to the corresponding row in the result vector  $y$ . The kernel is quite handy in handling the product of the smaller rows as compared to other kernels. For each segment, this kernel is launched with a total of  $((segment\_size + 256 - 1) / 256)$  blocks. The total number of threads per block can be varied as required to achieve optimal performance.

**Algorithm 5** SURAA\_Scalar\_Kernel**Input:**  $A_{csr} : (val, row\_ptr, col\_idx)$ ,  $vector : x$ ,  $vector : y$ ,  $vector : perm$ **Output:**  $vector : y$ 

```

1: function SCALARSPMV
2:    $tid \leftarrow global\_thread\_Id$ 
3:    $row\_id \leftarrow perm[tid]$ 
4:   for  $i = row\_ptr[rid]$  to  $row\_ptr[rid + 1] - 1$  do
5:      $sum \leftarrow sum + val[i] \times x[col\_idx[i]]$ 
6:   end for
7:    $y[row\_id] \leftarrow sum$ 
8: end function

```

## 4.2.3. Vector Kernel

Algorithm 6 illustrates the segment based vector SpMV algorithm. In this algorithm, a total of  $Y$  threads operate on each row. We assign  $Y$  as 32, 64, or 128 depending on  $npr$ .  $Y$  would be the smallest multiple of 32 greater than  $npr$ . The calculation of  $Y$  can be seen in lines 7, 8, and 9 in Algorithm 1. As in the Child kernel (see the explanation of Algorithm 4), we calculate the thread Id ( $tid$ ), lane Id ( $lid$ ), and warp Id ( $wid$ ) to access the data structures. A total of  $Y$  threads are assigned to each row. These  $Y$  threads will then use segmented parallel reduction using the shuffle instruction ( $\_\_shfl\_down\_sync$ ) to accumulate the partial results. The partial sum is then stored in the shared memory ( $shared\_mem$ )



for processing of all the relevant threads. We perform one more reduction to accumulate the final sum and store it in the result vector ( $y$ ). This is very similar to the CSR Vector scheme. The kernels for each segment is launched with  $((segment\_size * Y) + 256 - 1) / 256$  blocks.

---

**Algorithm 6** SURAA\_Vector\_Kernel
 

---

**Input:**  $A_{csr} : (val, row\_ptr, col\_idx)$ ,  $vector : x$ ,  $vector : y$ ,  $vector : perm$ ,  $Thread\_per\_row : Y$

**Output:**  $vector : y$

```

1: function DCSR VECTOR
2:    $tid \leftarrow thread\_Id$ 
3:    $lid \leftarrow tid \% Y$ 
4:    $wid \leftarrow tid / Y$ 
5:    $row\_id \leftarrow perm[wid]$ 
6:    $row\_cur \leftarrow row\_ptr[row\_id]$ 
7:    $row\_next \leftarrow row\_ptr[row\_id + 1]$ 
8:   for  $i = lid; i < row\_next - row\_cur; i += Y$  do
9:      $sum += values[i] \times x[col\_idx[i]]$ 
10:  end for
11:  for  $i = Y; i > 0; i = i >> 2$  do
12:     $sum += \_\_shfl\_down\_sync(0xffff, sum, i)$ 
13:  end for
14:  if  $tid \& ((Y >> 2) - 1) == 0$  then
15:     $shared\_mem[tid >> 5] \leftarrow sum$ 
16:  end if
17:  if  $lane\_id == 0$  then
18:     $sum += shared\_mem[(tid >> 5) + 1]$ 
19:     $y[row\_id] \leftarrow sum$ 
20:  end if
21: end function

```

---

#### 4.3. A Note on $max\_child\_threads$

The parameter  $max\_child\_threads$  in our code is a tunable parameter. It is the number of child grids that are pending, running, suspended, or waiting in the queue ready for execution. Exceeding the  $max\_child\_threads$  in earlier CUDA versions between 5.0 and 5.9 results in discarding the kernel. Such kernels were not executed.  $max\_child\_threads$  signifies the size of the buffer that tracks the running kernels as well as the kernels to be executed. The maximum child kernels can be modified in such scenarios using `cudaDeviceSetLimit()` which takes in two parameters, `cudaLimitDevRuntimePendingLaunchCount` and  $x$ , where  $x$  is the new number of child kernels. Changing the Pending Launch Count changes the size of the buffer. However, this leads to degradation in performance for older CUDA devices. The default space of the buffer is set as 2048 by default.

However, from CUDA 6.0 onwards, in addition to the fixed sized pool of size `cudaLimitDevRuntimePendingLaunchCount`, a variable sized virtualized pool has been added to the buffer. The fixed sized pool is used until it is filled. After that, the virtualized pool is used. The cost associated with the virtualized pool is slightly higher, but in the overall scenario for SpMV we find that use of virtualized pool contributes to better performance until a threshold. Wrong pool size affects the performance of SpMV. In our experiments, we tried 1024, 2048, 4096, and 8192 as pool sizes. We will discuss the effects of pool sizes on our benchmark suite in Section 5.

#### 4.4. Section Summary and Clarifications

We would like to clarify here that in using multiple dynamic kernels we do not imply that a single kernel with a fixed number of threads will not be able to guarantee a better or the best performance. The SURRA Master Program (Algorithm 1) first launches the Scalar (Algorithm 5) and Vector kernels (Algorithm 6) asynchronously. Subsequently, it invokes the dynamic parallelism kernel (Algorithm 3). Dynamic parallelism enables executing the child kernels (which are managing different

rows: Algorithm 4) with different thread configurations, determined dynamically, and this is not possible if we were to use a static kernel.

We reiterate that the number of threads for the dynamic kernels are calculated during the execution, dynamically, using the *Max\_Child\_Threads* parameter and the properties of the matrix being multiplied to a vector. If we were to reuse the Algorithm 2 for the setup phase and then spawn three separate asynchronous, non-dynamic, kernels, we will not be able to dynamically adjust the configuration of each thread that is spawned by the kernels. The dynamic kernel allows us to spawn a different number of child kernels, up to *Max\_Child\_Threads* (as discussed in Section 4.4), each having its own thread configuration parameters and this is not possible with a non-dynamic kernel. Consequently, it will not allow us to achieve higher performance as we are able to in the current case (for the results, see Section 5).

In summary, the six algorithms, their descriptions, and the other discussions in this section show that we have utilized streams, warps, warp divergence, and dynamic kernels to efficiently use the GPU resources and launched the maximum possible number of concurrently executing kernels.

## 5. Results and Analysis

In this section, we evaluate the performance of SURAA against seven well-known and widely used SpMV storage and computation techniques. It comprises seven subsections. The details of the experimental testbed including the storage schemes used are given in Section 5.1. The details of the dataset comprising 26 matrices for comparatively benchmarking SURAA and other schemes are provided in Section 5.2. Section 5.3 analyzes in detail the comparative performance of SURAA against the seven other schemes using three performance metrics, namely throughput, speedup and effective memory bandwidth. Section 5.4 provides a comparative analysis of SURAA with other schemes with respect to *npr* variance of all the 26 matrices in our dataset. Section 5.6 compares the setup cost or overhead of SURAA with other SpMV computation techniques. Section 5.5 discusses the parametric configuration for SURAA's performance in terms of the number of maximum child threads (*max\_child\_threads*) that can be invoked by the dynamic kernel. Finally, Section 5.7 relates SURAA with our broader work on developing sparse iterative solvers.

### 5.1. Experimental Testbed

The platform used for the experiments is a part of the 230 TFLOPS Aziz supercomputer (ranked number 360 in June 2015 among the Top500 machines). It consists of 496 nodes and 11,904 cores in total. Each node contains two Intel Xeon E5-2695v2 processors (Intel Corporation, Santa Clara, CA, USA), each with 12 Cores (2.4 GHz). In addition, 380 of these nodes have 96 GB Random Access Memory, while each of the rest 112 nodes contain 256 GB. This sums up to a total of 66 TB memory. It also has two Nvidia K20X GPU (Kepler) accelerator nodes and two Intel Phi 5110P MIC accelerator nodes. It is controlled by 64-bit Red Hat Enterprise Linux v6.5 (Red Hat, Riyadh, Saudi Arabia). We used the K20 nodes to execute our code. We used CUDA toolkit 9.0 with C++ for the compilation of our code. The code was compiled using *nvcc -O3 -arch = sm\_35*. We compare our scheme with the following seven SpMV storage and computation techniques:

1. The best CSR-based SpMV [58] from cuSPARSE v9.0 and CUSP v0.5.1 [52].
2. The best HYB [58] from the above two libraries.
3. BSR format from cuSPARSE. The block size for a given matrix was obtained from the experiments based on the performance. The best performing block size was selected for each matrix.
4. The CSR5 storage technique (see [79] for details of the scheme).
5. The SELL-P [74] sparse storage scheme, which is a padded version of SELL-C- $\sigma$  scheme [75]. The implementation of the SELL-P scheme that we have used in this paper is from the MAGMA library [83,84]. The best block sizes for the scheme were selected from our experiments based on the best performance.
6. The WPK1 and WPK2 storage techniques by Wong et al. [78].








## 5.2. Benchmark Suite

Table 3 illustrates the selected sparse matrices in our benchmark suite. We have selected 26 matrices for the benchmarking. Fifteen of these matrices are selected based on the recommendations by [85], which are the most frequently used matrices by the researchers for SpMV and iterative solvers. The other included matrices have been widely used (see e.g., [67,79]). All the selected matrices are from the University of Florida Sparse Matrix Collection [86]. The chosen matrices are from diverse applications that include computational fluid dynamics, bioengineering, thermal, quantum chemistry, circuit simulation, power and web search. Figure 6 indicates the normalized mean and standard deviation of the non zeros per row for the matrices in our Benchmark suite. The red dots in the figure indicates the normalized mean and the bars indicate the normalized standard deviation. We can observe the irregularity in the number of non-zeros per row for these selected matrices. The high standard deviation for most of the matrices indicates the challenge in designing a uniform storage format and SpMV algorithm.

**Table 3.** Benchmark dataset: 26 matrices.

Struct	Matrix	Rows	Columns	<i>nnz</i>	Density (%)	Symmetry	Problem Domain
	<i>12month1</i>	12 471	872 622	22 624 727	$2.08 \times 10^{-1}$	unsymmetric	undir. bipartite graph
	<i>amazon0312</i>	400 727	400 727	3 200 440	$1.99 \times 10^{-3}$	unsymmetric	directed graph
	<i>bundle_adj</i>	513 351	513 351	20 207 907	$7.66 \times 10^{-3}$	symmetric	computer graphics
	<i>Freescale1</i>	3 428 755	3 428 755	18 920 347	$1.61 \times 10^{-4}$	unsymmetric	circuit simulation
	<i>FullChip</i>	2 987 012	2 987 012	26 621 990	$2.98 \times 10^{-4}$	unsymmetric	circuit simulation
	<i>hollywood-2009</i>	1 139 905	1 139 905	57 515 616	$4.43 \times 10^{-3}$	symmetric	undirected graph
	<i>in-2004</i>	1 382 908	1 382 908	16 917 053	$8.85 \times 10^{-4}$	unsymmetric	directed graph
	<i>mouse_gene</i>	45 101	45 101	14 506 196	$7.13 \times 10^{-1}$	symmetric	undir. weighted graph
	<i>rail4284</i>	4284	1 096 894	11 284 032	$2.40 \times 10^{-1}$	unsymmetric	linear programming
	<i>RM07R</i>	381 689	381 689	37 464 962	$2.57 \times 10^{-2}$	unsymmetric	Computational Fluid Dynamics (CFD)
	<i>spal_004</i>	10 203	321 696	46 168 124	$1.41 \times 10^0$	unsymmetric	linear programming
	<i>thermal2</i>	1 228 045	1 228 045	4 904 179	$3.25 \times 10^{-4}$	symmetric	thermal
	<i>TSOPF_RS_b2383</i>	38 120	38 120	16 171 169	$1.11 \times 10^0$	unsymmetric	power network
	<i>wb-edu</i>	9 845 725	9 845 725	57 156 537	$5.90 \times 10^{-5}$	unsymmetric	directed graph
	<i>Zd_Jac2</i>	22 835	22 835	1 642 833	$3.15 \times 10^{-1}$	unsymmetric	chemical simulation
	<i>llr71c</i>	70 304	70 304	1 528 092	$3.09 \times 10^{-2}$	unsymmetric	chemical simulation
	<i>bundle1</i>	10 581	10 581	770 811	$0.68 \times 10^0$	symmetric	computer graphics
	<i>thermomech_TK</i>	102 158	102 158	711 558	$6.81 \times 10^{-3}$	symmetric	thermal

Table 3. Cont.

Struct	Matrix	Rows	Columns	<i>nnz</i>	Density (%)	Symmetry	Problem Domain
	<i>Andrews</i>	60 000	60 000	760 154	$2.11 \times 10^{-2}$	symmetric	computer graphics
	<i>ASIC_680K</i>	682 862	682 862	2 638 997	$5.65 \times 10^{-4}$	unsymmetric	circuit simulation
	<i>flickr</i>	820 878	820 878	9 837 214	$1.45 \times 10^{-3}$	unsymmetric	directed graph
	<i>torso1</i>	116 158	116 158	8 516 500	$6.31 \times 10^{-2}$	unsymmetric	2D/3D
	<i>SiO2</i>	155 331	155 331	11 283 503	$4.67 \times 10^{-2}$	symmetric	Quantum Chemistry
	<i>Ga3As3H12</i>	61 349	61 349	5 970 947	$1.58 \times 10^{-1}$	symmetric	Quantum Chemistry
	<i>av41092</i>	41 092	41 092	1 683 902	$9.97 \times 10^{-2}$	unsymmetric	2D/3D

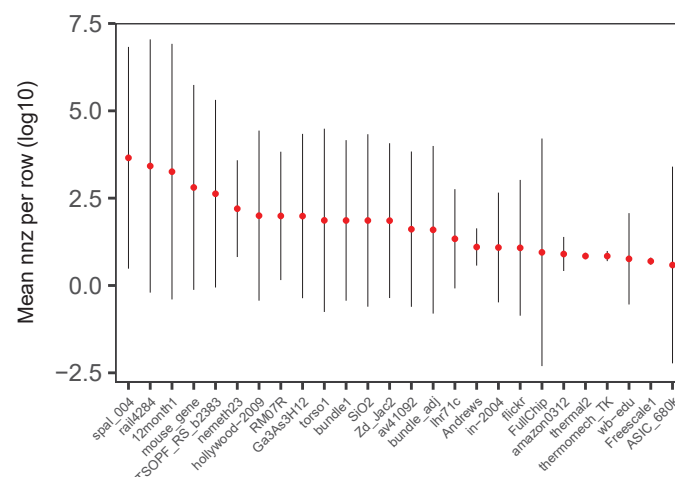


Figure 6. The normalized mean and the standard deviation of the matrices in the used Benchmark suite

### 5.3. SpMV Performance

In this subsection, we evaluate the performance of SURAA in terms of three main performance metrics: (1) Throughput (GFLOP/s), (2) Average Speedup, and (3) Effective Memory Bandwidth (GB/s). We first define these three performance metrics in this section.

Next, the throughput and speedup are evaluated in Section 5.3.1; it provides a detailed comparative analysis of SURAA including individual performance for all matrices in the dataset, average performance over all the matrices, best performance for any single matrix in the dataset, and a detailed discussion of comparison of SURAA with each of the other seven techniques (see the paragraph headings, WPK1 & WPK2, HYB, etc.).

The aggregate throughput and speedup are analyzed in Section 5.3.2. The effective memory bandwidth performance is compared in Section 5.3.3.

The throughput in GFLOP/s is calculated based on the effective flop count in an SpMV execution, as given below in Equation (2):

$$\text{throughput (GFLOP/s)} = \frac{2 \times \text{nnz}}{1.0e9 \times \text{time}}. \quad (2)$$

In the above equation, *nnz* denotes the total number of non-zero elements in the given matrix and *time* denotes the execution time in seconds.

The average speedup is calculated using the standard method depicted in Equation (3)

$$averagespeedup = \frac{\sum_{i=1}^{26} \frac{SURAA_i}{reference\_scheme_i}}{26}. \quad (3)$$

Here, in the above equation,  $SURAA_i$  indicates the throughput in GFLOP/s for each matrix from our benchmark suite of 26 matrices and  $reference\_scheme_i$  indicates the throughput (in GFLOP/s) of the reference scheme against which we make comparison, i.e., CSR, HYB, BSR, CSR5, SELL-P, WPK1, or WPK2.

The effective memory bandwidth in GB/s is calculated as in Equation (4):

$$e\_bandwidth \text{ (GB/s)} = \frac{brw}{1.0e9 \times time}. \quad (4)$$

In the above equation,  $brw = bytes\_read + bytes\_written$ , with  $bytes\_read$  and  $bytes\_written$  representing the number of bytes read and written during a single kernel execution time.

### 5.3.1. Throughput and Speedup

Figure 7 illustrates the double precision performance for the 26 benchmark matrices on an Nvidia K20 GPU. Each matrix was executed 1000 times using each sparse storage technique and their average was used to calculate throughput (GFLOP/s) using Equation (2). These throughput values for all the eight storage schemes are depicted as a bar chart in Figure 7.

#### Average Throughput Performance

We observe in Figure 7 that, overall, our proposed technique performs better than all the compared schemes, CSR, HYB, BSR, CSR5, SELL-P, WPK1, and WPK-2 (the SURAA throughput is represented by dark blue color bar and it is higher than the other bars in the figure). SURAA provides higher throughput than any other technique for 20 out of 26 matrices (we will come back to the remaining six matrices in a moment). On average, SURAA outperform all seven other SpMV computation methods by achieving an average speedup of  $40\times$  against SELL-P,  $29.89\times$  against WPK1,  $12.98\times$  for CSR,  $11.33\times$  for BSR,  $1.50\times$  for HYB,  $1.15\times$  for WPK2, and  $1.1\times$  for CSR5.

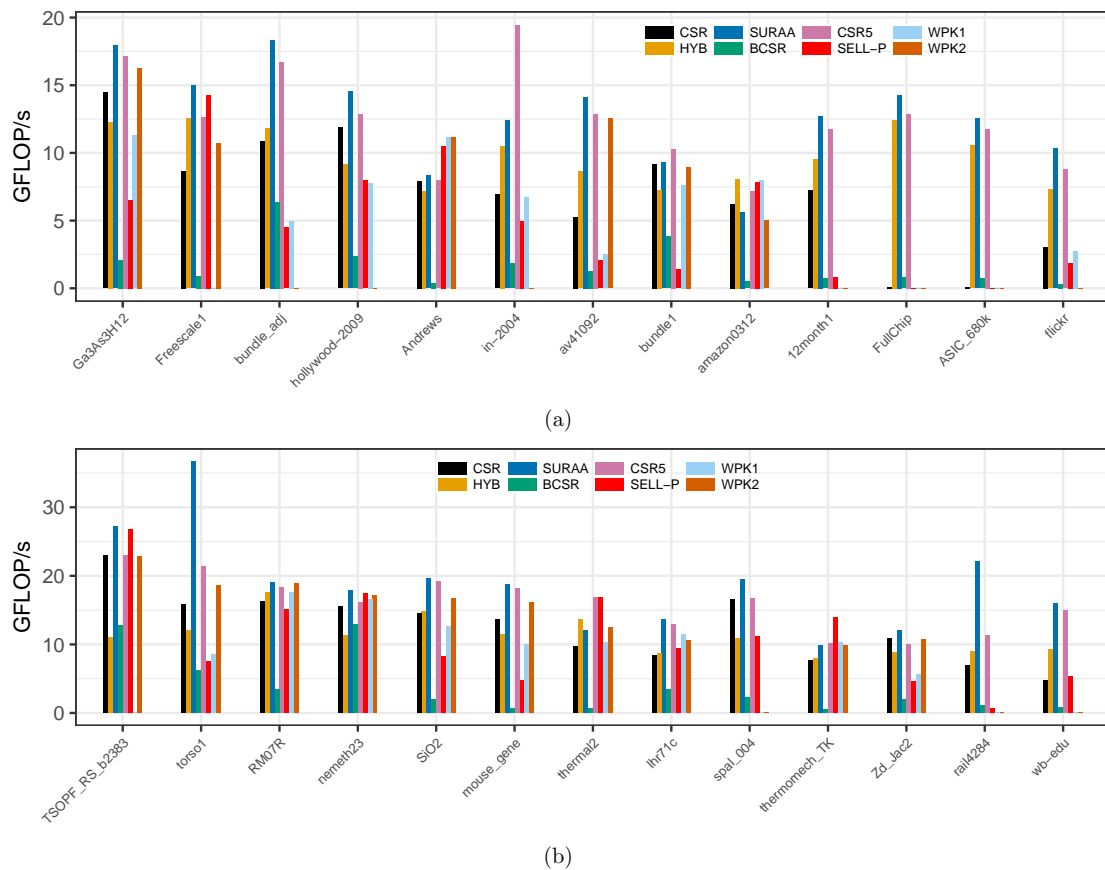
#### Best Performance for Individual Matrices

Note in Figure 7 that the highest speedups achieved by SURAA against other schemes for any single matrix are  $562\times$  (matrix *ASIC\_680ks*),  $531.56\times$  (for the matrix *ASIC\_680ks*),  $147\times$  (*FullChip*),  $26.70\times$  (*flickr*),  $3.06\times$  (matrix *Torso1*),  $1.97\times$  (*torso1*), and  $1.94\times$  (*rail4284*) against SELL-P, WPK1, CSR, BSR, HYB, WPK2, and CSR5, respectively. We repeat for clarity that these are SURAA's peak performance values against specific storage schemes for any single matrix. Even for blocked matrices, such as *ASIC\_680ks* and *TSOPF\_RS\_b2383*, we gain higher performance compared to the BSR technique, which is expected to perform better due to the block structure. Clearly, SURAA outperforms SELL-P, WPK1, CSR, BSR, and HYB by a fairly large margins; however, its performance gain over WPK2 and CSR5 is small.

Note that stating the highest gain against a particular scheme for a given matrix is meant to understand the performances of individual schemes against SURAA. It does not imply that SURAA has equally higher gain over other schemes. For example, the  $562\times$  speedup over SELL-P for the matrix *ASIC\_680ks* only applies to SELL-P, and Figure 7 clearly depicts that SURAA's performance is slightly higher ( $1.06\times$ ) than the next best scheme CSR5. Comparing each scheme with every other scheme will lengthen the paper, may confuse the reader, and some may find such analysis inappropriate.

We now dig deeper into the throughput performance of all the schemes by further elaboration.





**Figure 7.** SURAA: throughput comparison with the state of the art commercial (cuSPARSE) and open source (CUSP, MAGMA) tools: CSR, HYB, BCSR, CSR-5, SELL-P, WPK1, and WPK2 for (a) the 13 matrices from the benchmark set that provided lowest throughput and (b) the 13 matrices that provided the highest throughput for SURAA.

#### BCSR/BSR

SURAA outperforms the BSR scheme, overall, by moderate margins (average speedup:  $11.33\times$ , max:  $26.70\times$ ). Figure 7 depicts that BSR delivers poor performance compared to most schemes including SURAA.

#### HYB

SURAA outperforms the HYB scheme, overall, although by relatively smaller margins (average speedup:  $1.50\times$ , max:  $3.06\times$ ). The highest speedup achieved by SURAA against HYB for a single matrix from the 26 matrices is  $3.06\times$  for the matrix *torso1*; see Figure 7. The HYB scheme outperforms the SURAA method for two matrices, *thermal2* and *amazon0312*. We noted that this is also the case for a few other schemes, and we will discuss them in respective paragraphs. Further investigation reveals that SURAA does not utilize the dynamic parallelism for these two matrices because the largest rows for these matrices have only 11 and 10 non-zeros, respectively, and the average number of non-zeros is 7 and 8, respectively. Hence, SURAA uses the scalar CSR kernel for these two matrices (*thermal2* and *amazon0312*). We also observe (see Table 3 and Figure 6) that both of these matrices have low *npr* variance and standard deviation, and therefore do not benefit from the adaptive kernels in SURAA. We will see in Section 5.4 that SURAA provides higher differential performance for matrices with higher *npr* variance (a behavior which is highly desirable). The performance aspects of SURAA for low *npr* matrices will be investigated in the future to enable SURAA gain higher performance, also, for low *npr* variance matrices.

## CSR

SURAA outperforms the CSR scheme, overall, by moderate margins (average speedup:  $12.98\times$ , max:  $147\times$ ). Note in Figure 7 that CSR delivers moderate performance compared to SURAA and other schemes. However, its overall performance degradation ( $12.98\times$ ) comes from its poor performance for two matrices, *ASIC\_680k* and *FullChip*. SURAA achieves a higher throughput against CSR for all the 26 matrices except one matrix, *amazon0312* ( $0.91\times$ ). The reasons for this low performance of SURAA for this particular matrix have been already discussed alongside the HYB scheme.

## SELL-P

SURAA outperforms the SELL-P scheme, overall, by fairly large margins (average speedup:  $40\times$ , max:  $562\times$ ). The high speedup of SURAA against SELL-P is particularly attributed to the poor performance of SELL-P for two matrices, *ASIC\_680k* (speedup,  $562\times$ ) and *FullChip* (speedup,  $378\times$ ); see Figure 7. SELL-P mostly has produced very low throughput for high *npr variance* matrices such as *ASIC\_680k*, *FullChip*, *rail4284*, and others (see Figures 6 and 7). This behavior will be investigated further in Section 5.4. SURAA achieves a higher throughput against SELL-P for all the 26 matrices except four matrices, *thermal2* ( $0.71\times$ ), *amazon0312* ( $0.72\times$ ), *Andrews* ( $0.79\times$ ), and *Thermomech\_TK* ( $0.70\times$ ). The first two of these matrices have been mentioned already while discussing the HYB performance. The reason for this low performance, as before, is that SURAA did not invoke dynamic parallel kernel for these four matrices because these matrices have low average nonzero per row (see Table 3 and Figure 6). We have mentioned this earlier that SURAA does not invoke dynamic kernel for matrices which have average *npr* value less than 32. A possible modification in the future to improve the SURAA performance for such matrices is either to optimize our CSR scalar kernel or replace just the SURAA CSR scalar kernel with the cuSPARSE CSR kernel, which is very likely to make SURAA faster than all the techniques for all the matrices. Another possibility is to tune the parameter that invokes the dynamic kernel to an optimum value or to embed further intelligence into the kernel invocation process.

## CSR5

SURAA outperforms the CSR5 scheme, overall, although by relatively smaller margins (average speedup:  $1.1\times$ , max:  $1.94\times$ ). The highest speedup achieved for SURAA against CSR5 for any single matrix among the 26 matrices of our dataset is  $1.94\times$  (for matrix *rail4284*). SURAA achieves a higher throughput against CSR5 for all the 26 matrices except for four matrices, *thermal2* ( $0.71\times$ ), *amazon0312* ( $0.78\times$ ), *in-2004* ( $0.63\times$ ), and *bundle1* ( $0.90\times$ ). The reasons for low performance of SURAA for the first two matrices have already been discussed while discussing the HYB performance (i.e.,  $npr \leq 32$ ). The average *npr* for the third matrix, *in-2004*, is 12.23, and it has a medium *npr* variance; see Table 3 and Figure 6. Therefore, the reasons for the low performance of SURAA for this matrix are the same, i.e.,  $npr \leq 32$  implies no invocation of dynamic kernel in SURAA. The potential solutions to address these limitations of SURAA have also been mentioned earlier. The *npr* variance of the fourth matrix, *bundle1*, is high, with an average *npr* of 72.84. This matrix has neither low average *npr* nor low *npr* variance. However, *bundle1* is a small (10,581 rows), square, symmetric matrix and the difference in SURAA and CSR5 performance is quite small, 9.30 versus 10.30 GFLOP/s, respectively. Future work will focus on further investigation of SURAA's performance to improve its behavior in such cases.

## WPK1 and WPK2

SURAA outperforms the WPK1 scheme by large margin (average speedup:  $29.89\times$ , max:  $531.56\times$ ), but its performance over WPK2 is small (average speedup:  $1.15\times$ , max:  $1.97\times$ ). Our experiments suggest that WPK1 does not provide good results for asymmetric matrices (see Figure 7). There are a total of 16 asymmetric matrices in our dataset (see Table 3) and we noted that WPK1 produced relatively low throughput for all these matrices. Two of these 16 matrices, *Freescall* and *Fullchip*,

are exceptions because WPK1 failed to execute SpMV for these two matrices and caused a segmentation error. Particularly, for three matrices, *ASIC\_680ks*, *12month1*, and *rail4284*, WPK1 produced very low throughput results, all less than 0.04 GFLOP/s. On the contrary, WPK2 produced good results. Although it never produced the highest throughput for any of the benchmark matrices, it did produce high throughput comparable to the second high performing scheme CSR5. Overall, it produced the third best average performance after SURAA and CSR5 with small margins. SURAA achieved a higher throughput against WPK1 for all the 26 matrices except for three matrices, *amazon0312* ( $0.70\times$ ), *Andrews* ( $0.74\times$ ), and *Thermomech\_TK* ( $0.96\times$ ). SURAA also achieved a higher throughput against WPK2 for all the 26 matrices except *Andrews* ( $0.75\times$ ) and *thermal2* ( $0.96\times$ ). These four matrices belong to the set of the six matrices where SURAA did not produce the best performance and we have discussed the reasons for SURAA's poor performance for these matrices in the earlier paragraphs. Note that WPK2 failed to execute on ten matrices from the dataset, of which seven are from circuit simulation and directed graph, and the remaining three were rectangular matrices. On inspecting the WPK1 and WPK2 GitHub Web Page [87], we found that the developers have reported code error issues on the page as follows, “trying to debug issues with large unsymmetric matrices—now mainly thrust errors and a few numerical errors to debug on certain matrices”. Note that, in computing speedups of SURAA against other schemes, we did not include the results of the matrices where WPK1 and WPK2 schemes failed to execute.

### 5.3.2. Aggregate Throughput and Speedup

Table 4 provides for all eight schemes the statistical information such as *mean*, standard deviation (*sd*), variance (*var*), maximum (*max*), and minimum (*min*) of the GFLOP/s metric of the SpMV kernel for the 26 benchmark matrices. The values are listed in descending order of the *mean* value. We observe that SURAA provides the highest *mean* GFLOP/s of 15.75. The next best scheme in terms of the *mean* throughput is CSR5, it has the second highest *mean* GFLOP/s, 14.28, followed by 13.65 for WPK2. In terms of the *max* throughput, SURAA again leads the table with the highest value of 36.7077 GFLOP/s, followed by SELL-P with the second best value of 26.8 GFLOP/s, followed by CSR5. Based on the *mean* values, WPK2 comes third after SURAA and CSR5. As noted earlier, WPK1 and WPK2 were unable to execute some of the matrices in the benchmark dataset and their results have not been included in computing *mean* and other qualities in the table.

**Table 4.** Statistical information on the throughput in Giga Floating Point Operations per second (GFLOP/s) metric of the SpMV kernel for the 26 benchmark matrices.

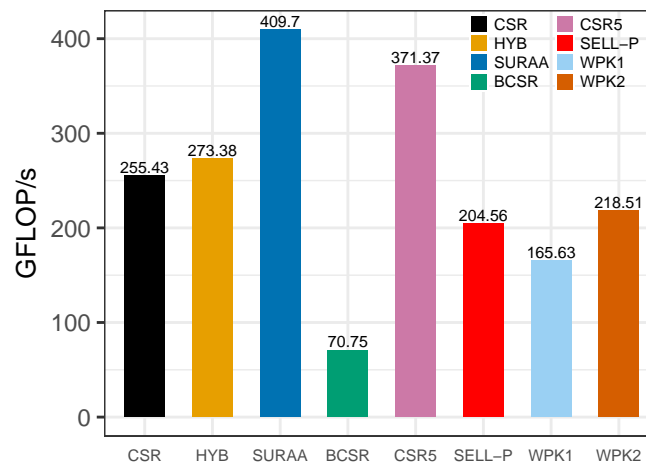
Scheme	Mean	sd	var	max	min
SURAA	15.7577504	6.345914889	40.27063578	36.70778163	5.650293069
CSR5	14.28352308	4.213901396	17.75696497	23.0126	7.17599
WPK2	13.65709	4.444209	19.75099	22.78246	5.016615
HYB	10.51479136	2.494732444	6.223689965	17.53628655	7.179585746
CSR	9.824100642	5.378886198	28.93241673	22.99131028	0.083364399
WPK1	8.717564	4.362194	19.02874	17.58582	0.02361821
SELL-P	7.867543077	6.535661011	42.71486485	26.8	0.0223
BCSR	2.721219	3.399639	11.55754	12.95043	0.313091

In order to calculate the aggregate performance of SURAA, we calculate aggregate speedup as in Equation (5).

$$aggregate\_speedup = \frac{\sum_{i=1}^{26} SURAA_i}{\sum_{i=1}^{26} reference\_scheme_i}. \quad (5)$$

Figure 8 indicates the aggregate GFLOP/s achieved for SURAA and other tools. These GFLOP/s are aggregated over all the matrices for each scheme and do not imply to be the GFLOP/s for a single GPU at one time. The intention of providing the aggregate throughput is to highlight the aggregate performance gain or loss that could be achieved by the use of a single SpMV computation scheme

for matrices from various domains. SURAA has the highest throughput compared to all the other seven techniques, i.e., CSR, HYB, BSR, CSR5, WPK1, WPK2, and SELL-P. For our benchmark suite of 26 matrices, we achieve an aggregate throughput of 409.7 GFLOP/s compared to the next best CSR5 (371.37 GFLOP/s), HYB (273.38 GFLOP/s), CSR (255.42 GFLOP/s), WPK2 (218.51 GFLOP/s), SELL-P (204.56 GFLOP/s), WPK1 (165.163 GFLOP/s), and BSR (70.751 GFLOP/s). The reason for low aggregate throughput for WPK1 and WPK2 is that, as noted earlier, they were unable to execute some of the matrices in the benchmark dataset. It means that SURAA has an aggregate speedup of  $1.1\times$  compared to CSR5,  $1.49\times$  compared to HYB,  $1.60\times$  compared to CSR,  $1.87\times$  compared with WPK2,  $2.0\times$  compared to SELL-P,  $2.47\times$  compared to WPK1, and  $5.79\times$  compared to BSR for the given 26 matrices.



**Figure 8.** SURAA: comparison of aggregate GFLOP/s of all the eight schemes.

### 5.3.3. Effective Memory Bandwidth

Effective memory bandwidth utilization indicates the processor usage efficiency by the kernels [88]. Specifically, effective memory bandwidth denotes the rate at which the bytes are read and written for a given kernel. The general formula to calculate the effective memory bandwidth was given in Equation (4).

We did not find the specific formulas in the literature for calculating the effective memory bandwidth for each of the schemes that we have considered in this paper. Therefore, we looked at each of the schemes, its memory requirements, and devised equations to calculate the number of bytes read and written,  $brw$  (see Equation (4)), based on their storage requirements. Equation (4) was then used to calculate the effective memory bandwidth,  $e\_bandwidth$ , in GB/s. Each scheme was executed 1000 times for each matrix in our dataset to get the average execution time for each of the 26 benchmark matrices. For further details on calculating effective memory bandwidth, see Nvidia documentation at <https://devblogs.nvidia.com/how-implement-performance-metrics-cuda-cc/>.

The formulas for the CSR and HYB schemes were reported in [58]) as given below:

$$e\_brw\_HYB = (12m \times anpr) + 8(m + n) + 16(n - anpr), \quad (6)$$

$$e\_brw\_CSR = (m + 1 + nnz) \times 4 + (nnz + m + n) \times 8. \quad (7)$$

The effective memory bandwidth for the SELL-P scheme was reported in [75], but no formula was given. After consulting [74,75], we devised the formula to calculate  $brw$  for SELL-P as below:

$$e\_brw\_SELLP = (12 \times seg\_size \times \sum_{i=1}^{num\_seg} max\_npr\_seg_i) + 4(2m + n) + \frac{m}{8} + 4, \quad (8)$$

where  $num\_seg$  is the number of segments in the matrix, as determined by the SELL-P scheme,  $seg\_size$  is the size of the segments that is fixed, and  $max\_npr\_seg_i$  is the maximum number of nonzero elements in any one row of segment  $i$ . It was difficult for us to calculate these numbers and therefore we approximated  $e\_brw\_SELLP$  as below:

$$e\_brw\_SELLP = (12m \times anpr) + 4(2m + n) + \frac{m}{8} + 4. \quad (9)$$

The effective memory bandwidth for the CSR5 scheme has not been reported before. After consulting [79] where this scheme was first introduced, we devised a formula to calculate  $brw$  for CSR5 as below:

$$e\_brw\_CSR5 = 4 \times (nnz + m + 1 + (1 + \frac{nnz}{\sigma \times 32}) + (32 \times 2) \times (1 + \sigma)) + 8 \times (nnz + m + n), \quad (10)$$

where  $\sigma$  is a parameter that determines the column width of the blocks in their data structure. It can assume three different values depending on the value of  $anpr$  of a matrix. These values are 4, 32, or  $anpr$ .

The effective memory bandwidth for WPK1 and WPK2 was reported by Wong et al. [78], but it was on a different GPU device (Nvidia GeForce GTX 480) and for different matrices. They did not give formulas to calculate the effective memory bandwidth. We looked at their paper and have derived the following formula to calculate the  $brw$ :

$$e\_brw\_WPKx = (12 \times warp\_size \times \sum_{i=1}^{num\_warps} max\_npr\_warp_i) + 4(5m + 3n) + \frac{m}{8}, \quad (11)$$

where  $num\_warp$  is the number of warps in the matrix, as determined by the WPK1 and WPK2 schemes,  $warp\_size$  is the size of the warps which is fixed, and  $max\_npr\_warp_i$  is the maximum number of nonzero elements in any one row of warp  $i$ . It was difficult for us to calculate these numbers and therefore we approximated  $e\_brw\_WPKx$  as below:

$$e\_brw\_WPKx = (12m \times anpr) + 4(5m + 3n) + \frac{m}{8}. \quad (12)$$

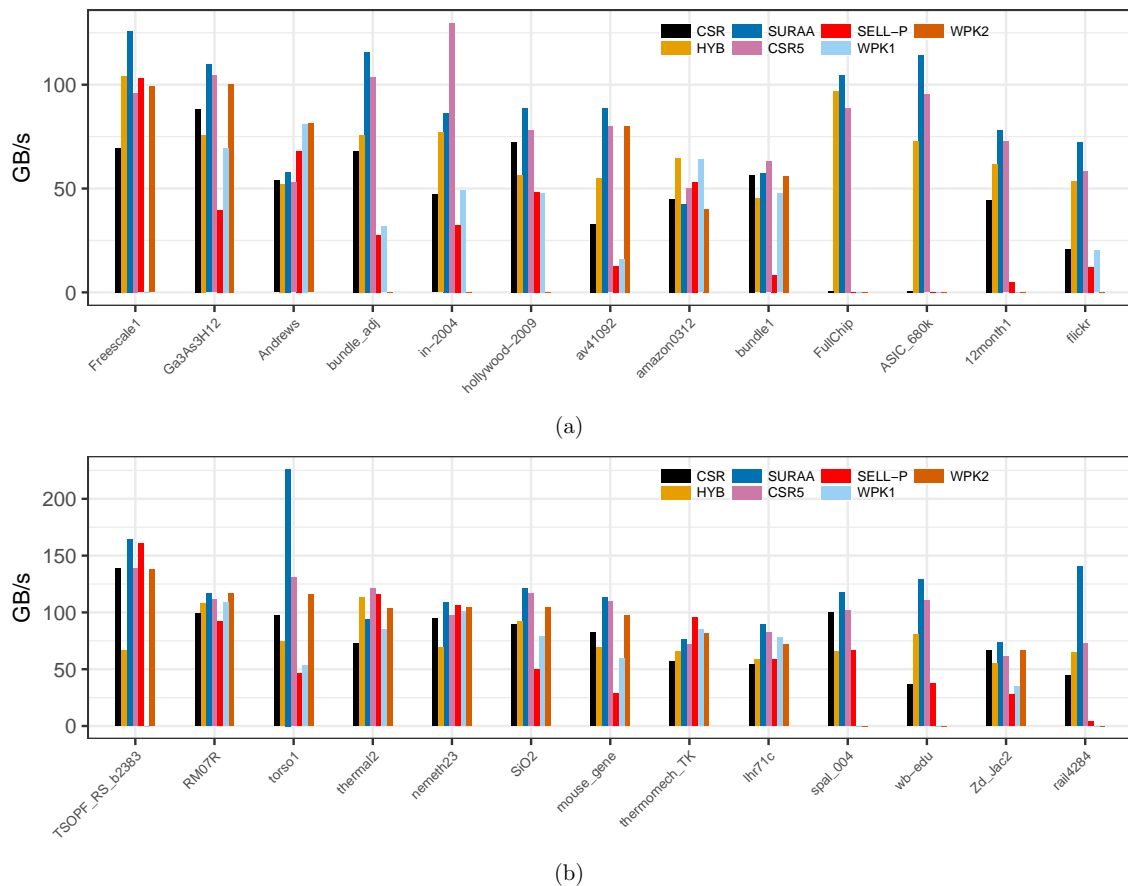
The effective memory bandwidth for the SURAA scheme can be calculated using the  $brw$  formula as below:

$$e\_brw\_SURAA = (2m + 1 + nnz) \times 4 + (nnz + m + n) \times 8. \quad (13)$$

Figure 9 plots the calculated effective memory bandwidth for double precision SpMV for CSR, HYB, SURAA, CSR5, SELL-P, WPK1, and WPK2. The maximum theoretical memory bandwidth of K20M is 208 GBytes/s. On average, SURAA delivers a bandwidth of 50.2% of the theoretical maximum as compared to 44.41%, 43.7%, 34.62%, 30.15%, 28.1%, and 24.02% of the theoretical maximum for CSR5, WPK2, HYB, CSR, WPK1, and SELL-P, respectively.

The highest effective memory bandwidths delivered by SURAA were 226.25 (*torso1*), 164.03 (*TSOPF\_RS\_b2383*), and 140.76 (*rail4284*) GB/s which are 109%, 78.9%, and 67.7% of the peak, respectively. A higher bandwidth, more than the theoretical bandwidth, signifies an efficient use of the cache and shared memory for SURAA when executing the matrix *torso1*. A bandwidth, for SpMV computations, higher than the peak memory bandwidth has also been reported earlier; see, e.g., [58,74]. The highest bandwidths obtained by the other techniques in descending order are: 160.75 GB/s (*ASIC\_680ks*) for SELL-P, 138.87 GB/s (*TSOPF\_RS\_b2383*) for CSR5, 138.48 GB/s (*TSOPF\_RS\_b2383*) for CSR, 137.55 GB/s (*TSOPF\_RS\_b2383*) for WPK2, 112.7 GB/s (*thermal2*) for HYB, and 108.39 GB/s (*RM07R*) for WPK1. Clearly, SURAA provides significantly higher effective memory bandwidth over all the schemes.





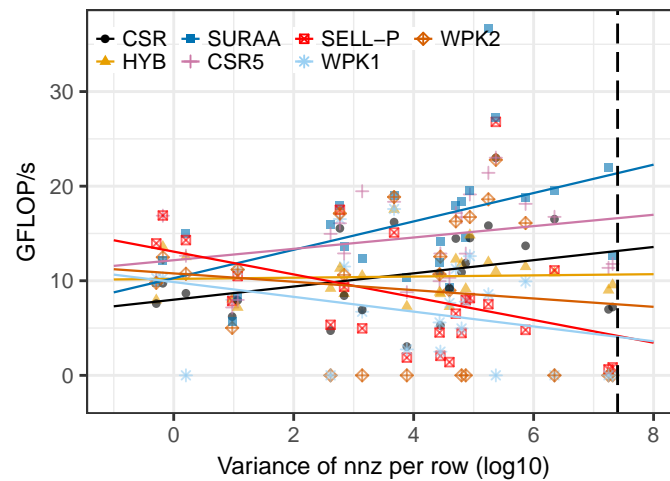
**Figure 9.** SURAA: Effective bandwidth comparison with the state of the art commercial (cuSPARSE) and open source (CUSP, MAGMA) tools: CSR, HYB, CSR-5, SELL-P, WPK1, and WPK2 for (a) the 13 matrices from the benchmark set that provided lowest bandwidths and (b) the 13 matrices that provided the highest bandwidth for SURAA.

#### 5.4. SURAA: Comparative Performance against High $npr$ Variance

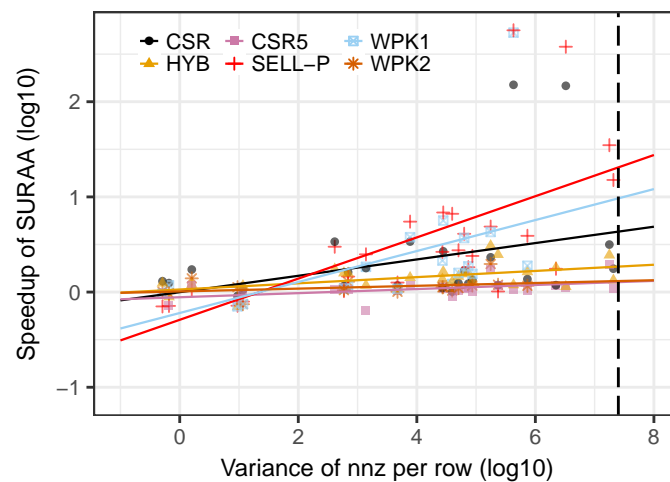
Figure 10 illustrates the relationship between the obtained throughput for the seven schemes, SURAA, CSR5, CSR, HYB, WPK2, SELL-P, and WPK1, with respect to the variance of the non-zero elements of all the 26 matrices in our dataset. The symbols indicate the actual GFLOP/s obtained and the straight lines indicate the smoothed GFLOP/s. We observe that SURAA provides better throughput with the increase in the variance of nonzero elements per row. On extrapolating the GFLOP/s, we observe that larger variations in nonzero elements per row will achieve higher GFLOP/s for SURAA compared to the next best CSR5, CSR, and HYB (note the upward slope for these schemes); and this behavior is highly desirable because higher variations usually decrease SpMV performance. We also observe that larger variations decrease the performance of the WPK2, WPK1, and SELL-P techniques.

Similarly, Figure 11 depicts the speedup of SURAA against CSR5, HYB, SELL-P, WPK1, WPK2, and CSR, with respect to the variance of the nonzero elements per row for all the 26 matrices in the benchmark suite. As in Figure 10, the symbols indicate the actual SURAA speedups obtained and the straight lines indicates the smoothed speedups. Note that the  $npr$  variance and the speedups are plotted in  $\log_{10}$ . We observe in the figure that the speedup of SURAA over SELL-P, WPK1, CSR, HYB, WPK2, and CSR5 is increasing with the increase in the variance of the number of non-zeros per row. As seen and discussed before, this indicates an excellent behavior for a sparse matrix computation tool. The rate of increase in speedup of SURAA against SELL-P, WPK1 and CSR is relatively high compared to the rest of the schemes and this has made the plots for HYB, WPK2, and CSR5 appear as

straight lines. To allow a closer look at the SURAA performance, Figure 12 depicts the same graphs as in Figure 11 but excludes the SELL-P, WPK1, and CSR data. The speedup scale now does not use log10. It can now be observed that the SURAA speedup against CSR5 and HYB show an increase for an increasing  $npr$  variance though the rate is lower than the other SpMV computation schemes. The highest rate of increase of SURAA speedup is over the HYB scheme followed by WPK2 and CSR5.



**Figure 10.** Throughput against  $npr$  variance for the benchmark suite comprising 26 matrices (compared schemes: CSR5, WPK1, WPK2, HYB, SELL-P, CSR, and SURAA).



**Figure 11.** SURAA speedup against  $npr$  variance for the 26 benchmark matrices (compared for CSR5, WPK1, WPK2, HYB, SELL-P, and CSR).

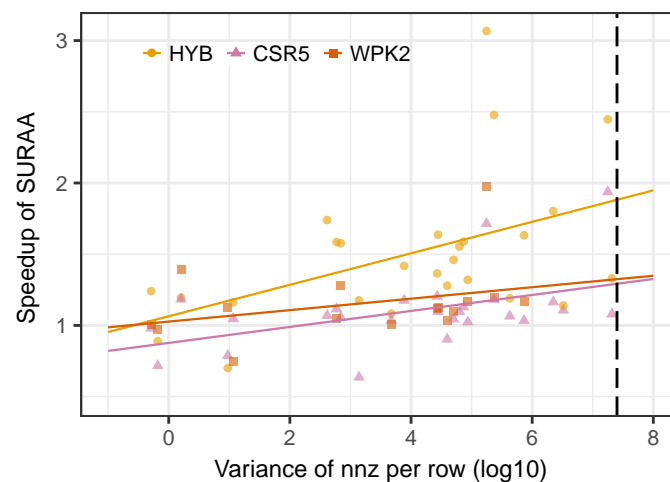
Figure 13 illustrates the GFLOP/s performance of BSR, SELL-P, WPK1, WPK2, CSR, HYB, CSR5, and SURAA, respectively, against the  $npr$  variance and the total number of nonzero elements,  $nnz$ .

Note the colour legend in the figure depicting colours ranging from blue to green, yellow, orange and red, representing GFLOP/s in increasing order from near zero (SELL-P) to maximum 36.70 (SURAA). More yellow/orange/red means higher throughput. Blue and green colours represent lower throughput.

BSR provides a low performance for the majority of the cases as shown by the blue shades that cover majority of the figure. The highest GFLOP/s results attained by BSR are 12.95 and 12.75, for the matrices *nemeth23* and *TSOPF\_RS\_b2383*, which have a block structure.

SELL-P shows a performance in the range of 10 to 26.8 GFLOP/s for smaller  $nnz$  values and  $npr$  variance varying from low to medium. However, a higher  $npr$  variance results in poor performance with GFLOP/s in the range of 0 to 5. We had discussed this earlier while evaluating SELL-P individual

performance. A look at Figures 6 and 7 will reveal that SELL-P mostly has produced very low throughput for high *npr* variance matrices such as *ASIC\_680k*.



**Figure 12.** Speedup against the *npr* variance for the 26 benchmark matrices (compared for CSR5, WPK2, and HYB).

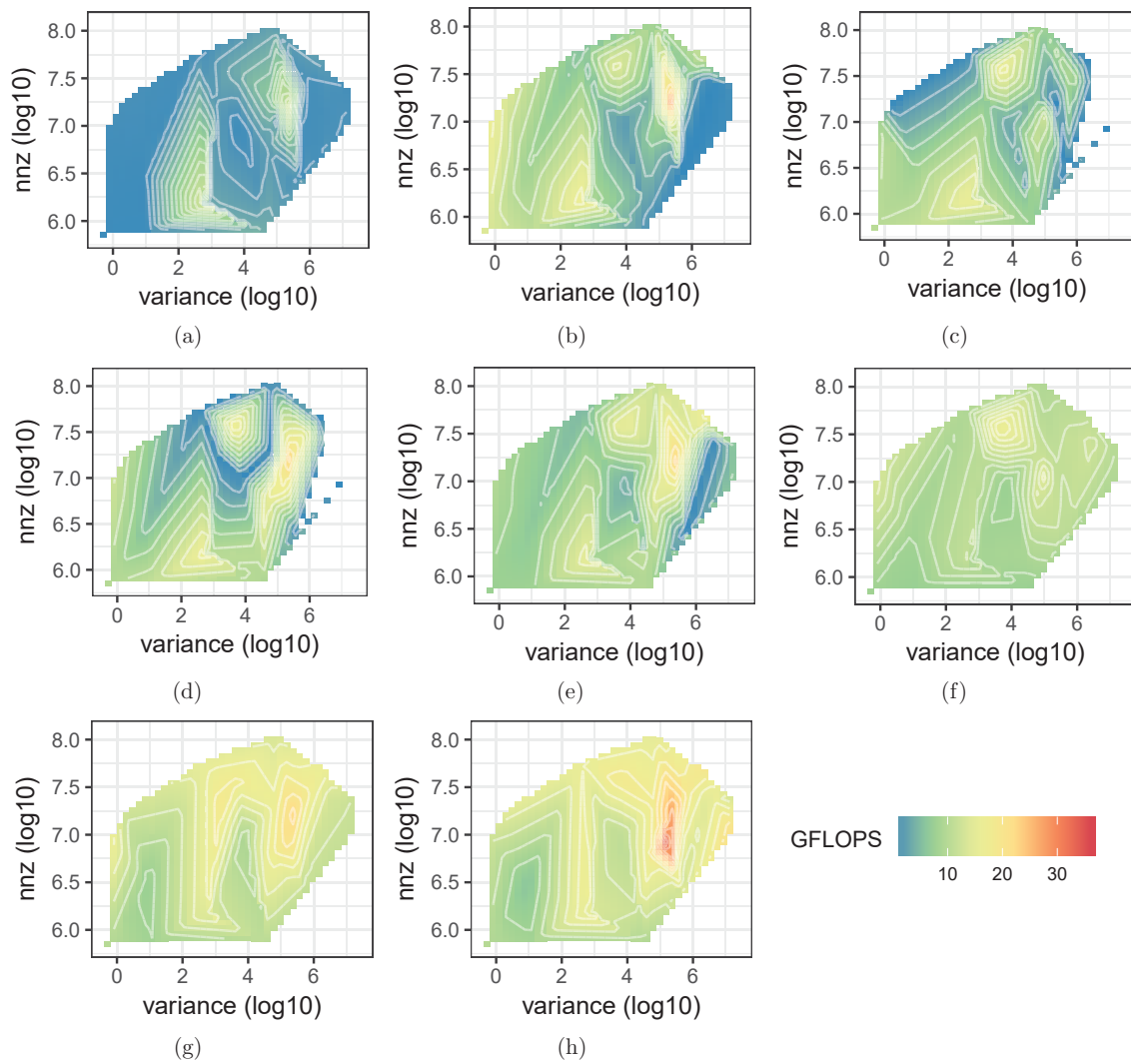
WPK1 delivers relatively low performance against the increasing *npr* variance as evident by mostly green and blue colors though some yellow patches can be seen towards lower and upper middle (medium *npr* variance and low/high *nnz*) sections of the plot representing higher GFLOP/s of 17.5. WPK2 also delivers low performance against the increasing *npr* variance. This can be seen by mostly green and blue colors though the yellow patches are larger for WPK2 compared to the WPK1. This seems surprising at the first look because WPK2 has delivered a superior performance overall after SURAA and CSR5 (max throughput is 22.78; see Table 4). A closer look at Figures 6 and 7 reveals that both WPK1 and WPK2 did not successfully execute some of the high *npr* variance matrices including *spal\_004*, *12month1*, and *rail4284*.

CSR provides average performance (depicted by the yellow color; see the GFLOP/s legend) in two scenarios: (1) for matrices with low *npr* variance and low *nnz* and (2) for medium *npr* variance and large *nnz*. For high variance scenarios, CSR provides low performance, indicated by the blue and green shaded regions in the figure. CSR performance for low variance scenarios is also poor. The blue color indicates that at high *npr* variance and high *nnz* CSR provides low performance.

For HYB, we note a general trend that the performance increases with the increasing *nnz* and the *npr* variance. However, we do note light green shades towards the right of the plot which indicate slightly poor performance. For the majority of the part, the HYB performance is almost constant within the range of 10 to 18 GFlops.

CSR5 shows a general trend of an increasing performance as the *nnz* and *npr* variance increases. It provides an average performance of 15 to 23 GFLOP/s as seen by the yellowish shade in the figure. Low to average performance (5 to 15 GFLOP/s) can be observed in three scenarios: (1) high *npr* variance and medium to high *nnz*, (2) low *nnz* and medium *npr* variance, and (3) low *npr* variance and low to medium *nnz*.

Finally, SURAA shows an excellent behavior in that it delivers high performance with the increase in the *npr* variance and *nnz*. The patch in red color on the right of the plot shows the highest performance for the associated range of *npr* variance and *nnz* depicted by the red color. SURAA achieves the highest GFLOP/s of 36.70 among all the schemes as can also be seen in Figure 7.



**Figure 13.** Throughout with respect to  $npr$  variance and  $nnz$  for (a) BSR; (b) SELL-P; (c) WPK1; (d) WPK2; (e) CSR; (f) HYB; (g) CSR5; and (h) SURAA.

### 5.5. SURAA: Parametric Configuration

The matrix, *spal\_004*, is a rectangular matrix with fewer rows and a large number of dense columns. It also has a fairly high  $npr$  variance. Due to the high density, it is highly suitable for dynamic parallelism. In the results presented in this paper, the parameter *max\_child\_threads* is set to 8192 for *spal\_004*. During the course of conducting our experiments, we found that configuring this parameter to 2048, 4096, and 8192 varies the performance of the SURAA tool. The typical trend was that the highest value provided the best performance for *spal\_004* due to the increase in dynamic parallelism. To illustrate, *spal\_004* achieved 19.5 GFLOP/s in our experiments with 8192 kernels, while it provided 13.60 GFLOP/s when the parameter was set to 2048. This indicates that tweaking of the *max\_child\_threads* affects the performance of the SpMV computation. However, larger values for *max\_child\_threads* cause larger overheads due to the spawning of the dynamic kernels, and, therefore, setting it to higher values should be done on a necessity basis. In our benchmark dataset, for 23 of the 26 matrices, there are no more than 2048 rows, in total, whose  $npr$  value is greater than 32. The three matrices are *spal\_004*, *rail4284* and *12month1*, with *spal\_004* the biggest exception containing 8192 rows which have more than 32  $npr$ . Therefore, we have set *max\_child\_threads* to 2048 for all the matrices except for the *spal\_004* matrix. An automatic configuration of the tool for this parameter can be done during the set up phase by calculating the number of rows with  $npr \geq 32$  and setting

the *max\_child\_threads* parameter to an appropriate value. In future work, we will further investigate this behavior.

### 5.6. Preprocessing Cost

The preprocessing time introduced due to the conversion of the input matrix in the base format (usually CSR/COO) to a more advanced format could be more than three or four orders of magnitude higher than the time required for executing a single sparse matrix vector multiplication [77]. Applications that utilize the same sparse matrix for multiple executions of SpMV operations such as iterative linear solvers amortize the preprocessing cost. However, when the iterations are small in the linear iterative solver or when the matrix structure changes frequently as in dynamic graph applications such as PageRank, the preprocessing cost can adversely affect the performance gains obtained while using a specific storage format.

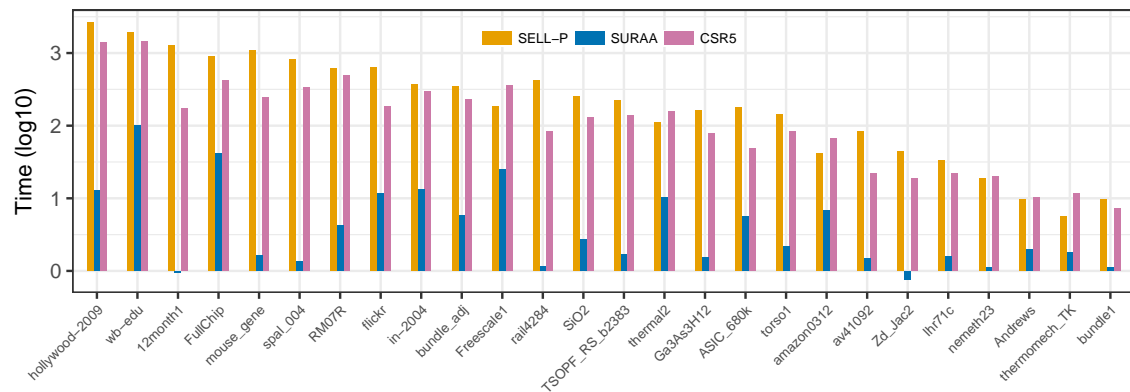
The major operation involved in converting CSR to SURAA is sorting the number of nonzero elements in each row. The sorting process in SURAA by default is performed on the GPU side unlike the SELL-P and CSR5 conversion routines in the MAGMA library. Technically executing on the CPU is not as bad as it sounds, rather the time it takes depends upon the type of operations involved. Conversions involving auxiliary structures are difficult to implement on the device side. We use the radix sort by the key algorithm provided by the thrust library on the device side for faster sorting of the non-zero elements per row. The rest of the processing such as application of FD rule and grouping for execution are not time intensive.

In our experiments, for the purpose of a fair comparison, we assume that the initial format of all the matrices before the conversion is CSR. Use of CSR/COO as an initial stage for the data structures have been discussed in [85]. Since the conversion cost between COO and CSR is negligible and SURAA loads matrices as CSR by default, we selected CSR as the initial state of the storage format. Moreover, our conversion time includes the sorting time (required also by SELL-P), data transfer time from the host to device, as well as the time for other processing required for each format.

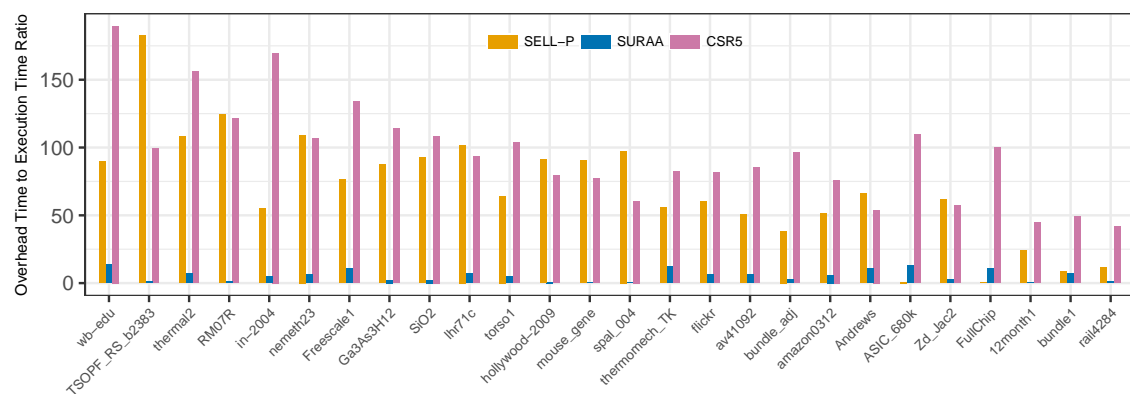
Figure 14 depicts the normalized overhead time taken for SELL-P, CSR-5, and SURAA for our 26 benchmark matrices. The time is obtained empirically by executing the codes for SURAA, SELL-P and CSR-5. We can observe that SURAA has the lowest conversion overhead compared to SELL-P and CSR-5. For a better understanding, we calculate the ratio of the measured conversion time to a single SpMV execution (rounded to the next integer), which denotes the number of sparse matrix vector operations that could have been possible in the overhead time. Figure 15 presents the ratio of the overhead in format conversion to the time for a single SpMV execution for SURAA, SELL-P, and CSR5. For the benchmark set of 26 matrices, the average ratio is 69, 5, and 96 for SELL-P, SURAA, and CSR5, respectively. This implies that on an average 69, 5, and 96 SpMV execution could have been performed in the time taken for format conversion. CSR5 has a higher overhead than SELL-P while converting graph based applications especially circuit simulation based matrices. The lowest ratio we obtained for SURAA is 0.26 (*12month1*), which is rounded to 0 and the largest ratio is 14 (*wb-edu*).

Figure 16 illustrates the absolute time taken for device based radix sort used in SURAA for sorting the matrix based on the number of nonzero values. We can observe that the graph based matrices such as *FullChip*, *wb-edu*, and *Freescale* take more time than other matrices. The average time for sorting the tested matrices is less than 3 ms.

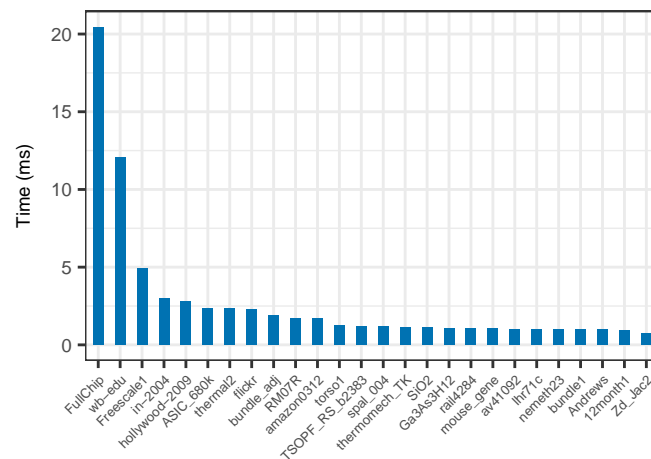




**Figure 14.** SURAA: Effective format conversion cost comparison with CSR-5 and SELL-P.



**Figure 15.** SURAA: A comparison of ratio of the conversion overhead time to the SpMV execution time for SELL-P, SURAA, and CSR5.



**Figure 16.** SURAA: Radix sort computation time on GPU for SURAA.

### 5.7. Sparse Iterative Solver Performance

Iterative solution of sparse linear equation systems can be considered as of prime importance due to its application in various important areas such as solving finite differences of PDEs, high accuracy surface modeling, finding steady-state probability vector of Markov chains, webranking, inventory control and manufacturing systems, queuing systems, fault modeling, and weather forecasting. This is a main focus of our research. Iterative methods converge based on the properties of the matrices. Many matrices take more than 1000 iterations to converge. The gain reported in the paper will lead to much larger time savings.

## 6. Conclusions

In this paper, we presented and evaluated SURAA, a new load balanced scheme for SpMV storage and computations that employ dynamic parallel kernel for the segments with larger rows, CSR vector kernel for segments with number of non-zeros per row greater than or equal to 32, and CSR scalar kernel for rows length shorter than 32. To further balance the load and reduce idle threads, the threads are distributed adaptively to the rows based on their lengths. We also achieve a coalesced memory access and accelerate the SpMV computations for a range of matrices with diverse sparsity features. The sparse matrix data structure is created by sorting the rows of the matrix on the basis of the nonzero elements per row (*npr*) and forming segments of equal size (containing approximately an equal number of nonzero elements per row) using the Freedman–Diaconis (FD) rule. To the best of our knowledge, no other work exists that used the FD rule for SpMV computations the way we used it, or used adaptive kernels the way we have done in this paper.

We conducted experiments to assess the performance of SURAA on K20 GPUs and compared with the best techniques from the best CUSP (open source) and cuSPARSE (commercial) libraries. A benchmark dataset was formulated using widely used benchmarks comprising 26 matrices from 13 diverse domains. SURAA was compared with a total of seven other SpMV storage and computation techniques using three widely used performance metrics; throughput (GFLOP/s), speedup, and effective memory bandwidth (GB/s).

SURAA outperformed, overall, all other seven SpMV computation methods with the average speedup varying between  $40\times$  for SELL-P and  $1.15\times$  for WPK2 and  $1.1\times$  for CSR5, with the average collective speedup of  $13.99\times$ . SURAA provided the best throughput for 20 out of the 26 benchmark matrices. The highest speedups achieved by SURAA against other schemes for any single matrix varied between  $562\times$  against SELL-P and  $1.94\times$  against CSR5. SURAA provided the highest *mean* throughput (GFLOP/s) of 15.75, followed by 14.28 by CSR5, and the highest *max* throughput of 36.70 GFLOP/s followed by 26.8 by SELL-P. SURAA also provided the highest effective memory bandwidth of 226.25 GB/s compared to the second best 160.75 GB/s by SELL-P. Moreover, SURAA has shown a much lower setup overhead, equal to five SpMV execution time, compared to 69 and 95.5 SpMV executions for SELL-P and CSR5. The performance analysis of SURAA revealed that it consistently delivers the highest throughput among all the compared schemes with the increase in the *npr variance* and *nnz*, a property which is highly desirable.

While the speedup of SURAA against CSR5 and WPK2 was small, SURAA provided the highest throughput for 20 out of 26 matrices. More importantly, none of the schemes except SURAA delivered consistently high performance on all of the performance benchmarks. SURAA code can be found at the github online repository (see <http://github.com/stormvirux/suraa>).

In our future work, we shall strive to achieve parametric improvements for the various parameters that have been used in our work such as the number of segments and the number of child threads for parent kernel and use larger datasets to get a better understanding. We will also look to further optimize the scalar, vector, and dynamic kernels. We have achieved significant progress in addressing the major challenges of GPU SpMV computing including coalesced memory access and load balancing. Future work will look into further improving our method based on these challenges. SURAA did not perform for low *npr variance* matrices as well as it did for the high *npr variance* matrices. Future work will also look into improving this behavior of the tool. We will investigate further optimization of scalar vector and dynamic kernels using significantly larger data sets.

**Author Contributions:** Conceptualization, T.M. and R.M.; methodology, T.M. and R.M.; software, T.M.; validation, T.M. and R.M.; formal analysis, T.M. and R.M.; investigation, T.M. and R.M.; resources, R.M., I.K. and A.A.; data curation, T.M.; writing—original draft preparation, T.M.; writing—review and editing, R.M.; visualization, T.M. and R.M.; supervision, R.M.; project administration, R.M. and A.A.; funding acquisition, R.M., A.A. and I.K.

**Funding:** This research was funded by the Deanship of Scientific Research (DSR) at the King Abdulaziz University (KAU), Jeddah, Saudi Arabia, under Grant No. G-651-611-38.

**Acknowledgments:** The authors acknowledge with thanks the technical and financial support from the Deanship of Scientific Research (DSR) at the King Abdulaziz University (KAU), Jeddah, Saudi Arabia, under Grant No. G-651-611-38. The experiments reported in this paper were performed on the Aziz supercomputer at KAU. We are thankful to the anonymous reviewers whose comments have greatly improved the quality of this paper.

**Conflicts of Interest:** The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

## References

1. Asanovic, K.; Bodik, R.; Catanzaro, B.C.; Gebis, J.J.; Husbands, P.; Keutzer, K.; Patterson, D.A.; Plishker, W.L.; Shalf, J.; Williams, S.W.; et al. *The Landscape of Parallel Computing Research: A View from Berkeley*; Technical Report UCB/EECS-2006-183; EECS Department, University of California: Berkeley, CA, USA, 2006.
2. Dziekonski, A.; Mrozowski, M. Block Conjugate-Gradient Method With Multilevel Preconditioning and GPU Acceleration for FEM Problems in Electromagnetics. *IEEE Antennas Wirel. Propag. Lett.* **2018**, *17*, 1039–1042. [[CrossRef](#)]
3. Afzal, A.; Ansari, Z.; Faizabadi, A.R.; Ramis, M. Parallelization strategies for computational fluid dynamics software: State of the art review. *Arch. Comput. Methods Eng.* **2017**, *24*, 337–363. [[CrossRef](#)]
4. Golovashkin, D.L.; Vorotnokova, D.G.; Kochurov, A.V.; Malysheva, S.A. Solving finite-difference equations for diffractive optics problems using graphics processing units. *Opt. Eng.* **2013**, *52*, 091719. [[CrossRef](#)]
5. Yan, C.; Zhao, G.; Yue, T.; Chen, C.; Liu, J.; Li, H.; Su, N. Speeding up the high-accuracy surface modelling method with GPU. *Environ. Earth Sci.* **2015**, *74*, 6511–6523. [[CrossRef](#)]
6. Mehmood, R.; Crowcroft, J. *Parallel Iterative Solution Method for Large Sparse Linear Equation Systems*; Technical Report UCAM-CL-TR-650; University of Cambridge, Computer Laboratory: Cambridge, UK, 2005.
7. Mehmood, R.; Parker, D.; Kwiatkowska, M. *An Efficient BDD-Based Implementation of Gauss–Seidel for CTMC Analysis*; Technical Report CSR-03-13; School of Computer Science, University of Birmingham: Birmingham, UK, 2003.
8. Kwiatkowska, M.; Mehmood, R.; Norman, G.; Parker, D. A Symbolic Out-of-Core Solution Method for Markov Models. *Electron. Notes Theor. Comput. Sci.* **2002**, *68*, 589–604. [[CrossRef](#)]
9. 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: Second Joint International Workshop PAPM-PROBMIV 2002 Copenhagen, Denmark, 25–26 July 2002 Proceedings*; Hermanns, H., Segala, R., Eds.; Springer: Berlin/Heidelberg, Germany, 2002; Volume 2399, Chapter 9, pp. 135–151. [[CrossRef](#)]
10. Kwiatkowska, M.; Parker, D.; Zhang, Y.; Mehmood, R. Dual-Processor Parallelisation of Symbolic Probabilistic Model Checking. In *Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, Volendam, The Netherlands, 8 October 2004; pp. 123–130. [[CrossRef](#)]
11. Mehmood, R. *A Survey of Out-Of-Core Analysis Techniques in Stochastic Modelling*; Technical Report CSR-03-7; School of Computer Science, University of Birmingham: Birmingham, UK, 2003.
12. Garrappa, R.; Moret, I.; Papolizio, M. Solving the time-fractional Schrödinger equation by Krylov projection methods. *J. Comput. Phys.* **2015**, *293*, 115–134. [[CrossRef](#)]
13. Page, L.; Brin, S.; Motwani, R.; Winograd, T. *The PageRank Citation Ranking: Bringing Order to the Web*; Technical Report; Stanford InfoLab: Stanford, CA, USA, 1999.
14. Kamvar, S.D.; Haveliwala, T.H.; Manning, C.D.; Golub, G.H. Extrapolation Methods for Accelerating PageRank Computations. In *Proceedings of the 12th International Conference on World Wide Web*, Budapest, Hungary, 20–24 May 2003; pp. 261–270. [[CrossRef](#)]
15. Langville, A.N.; Meyer, C.D. A Survey of Eigenvector Methods for Web Information Retrieval. *SIAM Rev.* **2005**, *47*, 135–161. [[CrossRef](#)]
16. Buzacott, J.A.; Shanthikumar, J.G. *Stochastic Models of Manufacturing Systems*; Prentice Hall: Englewood Cliffs, NJ, USA, 1993; Volume 4.
17. Kim, B.; Kim, J. Stability of a two-class two-server retrial queueing system. *Perform. Eval.* **2015**, *88–89*, 1–17. [[CrossRef](#)]

18. Kim, B.; Kim, J. A single server queue with Markov modulated service rates and impatient customers. *Perform. Eval.* **2015**, *83–84*, 1–15. [[CrossRef](#)]
19. Buchholz, P. A class of hierarchical queueing networks and their analysis. *Queueing Syst.* **1994**, *15*, 59–80. [[CrossRef](#)]
20. Ching, W.K. Iterative Methods for Queuing Systems with Batch Arrivals and Negative Customers. *Bit Numer. Math.* **2003**, *43*, 285. [[CrossRef](#)]
21. Ching, W.K.; Huang, X.; Ng, M.K.; Siu, T.K. Queueing Systems and the Web. *Markov Chains* **2013**, *189*, 47–76. [[CrossRef](#)]
22. Ching, W.K.; Huang, X.; Ng, M.K.; Siu, T.K. Manufacturing and Re-manufacturing Systems. *Markov Chains Models Algorithms Appl.* **2013**, *189*, 1–243. [[CrossRef](#)]
23. Mehmood, R.; Lu, J.A. Computational Markovian analysis of large systems. *J. Manuf. Technol. Manag.* **2011**, *22*, 804–817. [[CrossRef](#)]
24. Stewart, W.J.; Atif, K.; Plateau, B. The numerical solution of stochastic automata networks. *Eur. J. Oper. Res.* **1995**, *86*, 503–525. [[CrossRef](#)]
25. Chan, R.H.; Ching, W.K. Circulant preconditioners for stochastic automata networks. *Numer. Math.* **2000**, *87*, 35–57. [[CrossRef](#)]
26. Heffes, H.; Lucantoni, D. A Markov Modulated Characterization of Packetized Voice and Data Traffic and Related Statistical Multiplexer Performance. *IEEE J. Sel. Areas Commun.* **1986**, *4*, 856–868. [[CrossRef](#)]
27. Mehmood, R.; Alturki, R.; Zeadally, S. Multimedia applications over metropolitan area networks (MANs). *J. Netw. Comput. Appl.* **2011**, *34*, 1518–1529. [[CrossRef](#)]
28. El-Gorashi, T.E.H.; Pranggono, B.; Mehmood, R.; Elmirghani, J.M.H. A data Mirroring technique for SANs in a Metro WDM sectioned ring. In Proceedings of the International Conference on Optical Network Design and Modeling, Vilanova i la Geltru, Spain, 12–14 March 2008; pp. 1–6. [[CrossRef](#)]
29. Pranggono, B.; Mehmood, R.; Elmirghani, J.M.H. Performance Evaluation of a Metro WDM Multi-channel Ring Network with Variable-length Packets. In Proceedings of the IEEE International Conference on Communications, Glasgow, UK, 24–28 June 2007; pp. 2394–2399. [[CrossRef](#)]
30. Pranggono, B.; Mehmood, R.; Elmirghani, J.M.H. Data Mirroring for Metro WDM Storage Area Networks. In Proceedings of the 9th International Conference on Transparent Optical Networks, Rome, Italy, 1–5 July 2007; Volume 4, pp. 88–91. [[CrossRef](#)]
31. El-Gorashi, T.E.; Pranggono, B.; Mehmood, R. A Mirroring Strategy for SANs in a Metro WDM Sectioned Ring Architecture under Different Traffic Scenarios. *J. Opt. Commun.* **2008**, *29*, 89–97. [[CrossRef](#)]
32. Mehmood, R.; Crowcroft, J.; Elmirghani, J.M.H. A Parallel Implicit Method for the Steady-State Solution of CTMCs. In Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation, Monterey, CA, USA, 11–14 September 2006; pp. 293–302. [[CrossRef](#)]
33. Bylina, J.; Bylina, B. A Markovian Queueing Model of a WLAN Node. *Comput. Netw.* **2011**, *160*, 80–86. [[CrossRef](#)]
34. Bylina, J.; Bylina, B.; Karwacki, M. A Markovian Model of a Network of Two Wireless Devices. *Comput. Netw.* **2012**, *291*, 411–420. [[CrossRef](#)]
35. Bianchi, G. Performance analysis of the IEEE 802.11 distributed coordination function. *IEEE J. Sel. Areas Commun.* **2000**, *18*, 535–547. [[CrossRef](#)]
36. Park, P.; Marco, P.D.; Soldati, P.; Fischione, C.; Johansson, K.H. A generalized Markov chain model for effective analysis of slotted IEEE 802.15.4. In Proceedings of the IEEE 6th International Conference on Mobile Adhoc and Sensor Systems, Macau, China, 12–15 October 2009; pp. 130–139. [[CrossRef](#)]
37. Muhammed, T.; Mehmood, R.; Albeshri, A. Enabling reliable and resilient IoT based smart City Applications. In *Smart Societies, Infrastructure, Technologies and Applications. SCITA 2017. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering (LNICST)*; Mehmood, R., Bhaduri, B., Katib, I., Chlamtac, I., Eds.; Springer: Cham, Switzerland, 2018; Volume 224, pp. 169–184. [[CrossRef](#)]
38. Bustamam, A.; Burrage, K.; Hamilton, N.A. Fast Parallel Markov Clustering in Bioinformatics Using Massively Parallel Computing on GPU with CUDA and ELLPACK-R Sparse Format. *IEEE/ACM Trans. Comput. Biol. Bioinform.* **2012**, *9*, 679–692. [[CrossRef](#)] [[PubMed](#)]
39. Altowaijri, S.; Mehmood, R.; Williams, J. A Quantitative Model of Grid Systems Performance in Healthcare Organisations. In Proceedings of the International Conference on Intelligent Systems, Modelling and Simulation, Liverpool, UK, 27–29 January 2010; pp. 431–436. [[CrossRef](#)]

40. Muhammed, T.; Mehmood, R.; Albeshri, A.; Katib, I. UbeHealth: A Personalized Ubiquitous Cloud and Edge-Enabled Networked Healthcare System for Smart Cities. *IEEE Access* **2018**, *6*, 32258–32285. [\[CrossRef\]](#)
41. Mehmood, R. Towards understanding intercity traffic interdependencies. In Proceedings of the 2007 14th World Congress on Intelligent Transport Systems (ITS), Beijing, China, 9–13 October 2007; Volume 3, pp. 1793–1799.
42. Mehmood, R.; Nekovee, M. Vehicular AD HOC and grid networks: Discussion, design and evaluation. In Proceedings of the 14th World Congress on Intelligent Transport Systems, Beijing, China, 9–13 October 2007; Volume 2, pp. 1555–1562.
43. Lafferty, J.D.; McCallum, A.; Pereira, F.C.N. *Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data*; Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2001; pp. 282–289.
44. 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.* **2017**, *37*, 75–104. [\[CrossRef\]](#)
45. Mehmood, R. Serial Disk-Based Analysis of Large Stochastic Models. In *Validation of Stochastic Systems: A Guide to Current Research*; Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.P., Siegle, M., Eds.; Springer: Berlin/Heidelberg, Germany, 2004; pp. 230–255. [\[CrossRef\]](#)
46. Usman, S.; Mehmood, R.; Katib, I. Big data and HPC convergence: The cutting edge and outlook. In *International Conference on Smart Cities, Infrastructure, Technologies and Applications (SCITA 2017)*; Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST; Springer: Cham, Switzerland, 2018; Volume 224, pp. 11–26. [\[CrossRef\]](#)
47. Mehmood, R.; Graham, G. Big Data Logistics: A health-care Transport Capacity Sharing Model. *Procedia Comput. Sci.* **2015**, *64*, 1107–1114. [\[CrossRef\]](#)
48. Alyahya, H.; Mehmood, R.; Katib, I. Parallel Sparse Matrix Vector Multiplication on Intel MIC: Performance Analysis. In *Smart Societies, Infrastructure, Technologies and Applications*; Mehmood, R., Bhaduri, B., Katib, I., Chlamtac, I., Eds.; Springer International Publishing: Cham, Switzerland, 2018; pp. 306–322.
49. Mehmood, R.; Bhaduri, B.; Katib, I.; Chlamtac, I. (Eds.) Smart Societies, Infrastructure, Technologies and Applications. In Proceedings of the First International Conference, SCITA 2017, Jeddah, Saudi Arabia, 27–29 November 2017. [\[CrossRef\]](#)
50. Saad, Y. *Iterative Methods for Sparse Linear Systems*, 2nd ed.; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2003.
51. Wu, K.; Truong, N.; Yuksel, C.; Hoetzlein, R. Fast Fluid Simulations with Sparse Volumes on the GPU. *Comput. Graph. Forum* **2018**, *37*, 157–167. [\[CrossRef\]](#)
52. Dalton, S.; Bell, N.; Olson, L.; Garland, M. Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations, Version 0.5.0. 2014. Available online: <https://cusplibrary.github.io/> (accessed on 22 June 2018).
53. cuSPARSE, NVIDIA Developers, 2018. Available online: <https://developer.nvidia.com/cusparses> (accessed on 22 June 2018).
54. Freedman, D.; Diaconis, P. On the histogram as a density estimator: L2 theory. *Z. Wahrscheinlichkeitstheorie Verw. Geb.* **1981**, *57*, 453–476. [\[CrossRef\]](#)
55. Birgé, L.; Rozenholc, Y. How many bins should be put in a regular histogram. *ESAIM PS* **2006**, *10*, 24–45. [\[CrossRef\]](#)
56. Mehmood, R. Disk-Based Techniques for Efficient Solution of Large Markov Chains. Ph.D. Thesis, School of Computer Science, University of Birmingham, Birmingham, UK, 2004.
57. Saad, Y. SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations-Version 2. 1994. Available online: <https://www-users.cs.umn.edu/~saad/software/SPARSKIT/> (accessed on 19 July 2018).
58. Bell, N.; Garland, M. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In Proceedings of the SC '09: Proceedings Conference on High Performance Computing Networking, Storage and Analysis, Portland, OR, USA, 14–20 November 2009. [\[CrossRef\]](#)
59. Wang, J.; Yalamanchili, S. Characterization and analysis of dynamic parallelism in unstructured GPU applications. In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), Raleigh, NC, USA, 26–28 October 2014; pp. 51–60. [\[CrossRef\]](#)



60. Zhang, P.; Holk, E.; Matty, J.; Misurda, S.; Zalewski, M.; Chu, J.; McMillan, S.; Lumsdaine, A. Dynamic Parallelism for Simple and Efficient GPU Graph Algorithms. In Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms, Austin, TX, USA, 15 November 2015; pp. 11:1–11:4. [\[CrossRef\]](#)
61. NVIDIA. *CUDA C Programming Guide*; NVIDIA: Santa Clara, CA, USA, 2018.
62. Tang, X.; Pattnaik, A.; Jiang, H.; Kayiran, O.; Jog, A.; Pai, S.; Ibrahim, M.; Kandemir, M.T.; Das, C.R. Controlled Kernel Launch for Dynamic Parallelism in GPUs. In Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA), Austin, TX, USA, 4–8 February 2017; pp. 649–660. [\[CrossRef\]](#)
63. Wang, J.; Rubin, N.; Sidelnik, A.; Yalamanchili, S. Dynamic Thread Block Launch: A Lightweight Execution Mechanism to Support Irregular Applications on GPUs. *SIGARCH Comput. Arch. News* **2015**, *43*, 528–540. [\[CrossRef\]](#)
64. Chen, G.; Shen, X. Free launch: Optimizing GPU dynamic kernel launches through thread reuse. In Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Waikiki, HI, USA, 5–9 December 2015; pp. 407–419. [\[CrossRef\]](#)
65. Cevahir, A.; Nukada, A.; Matsuoka, S. Fast Conjugate Gradients with Multiple GPUs. In *Computational Science—ICCS 2009: 9th International Conference Baton Rouge, LA, USA, 25–27 May 2009 Proceedings, Part I*; Allen, G., Nabrzyski, J., Seidel, E., van Albada, G.D., Dongarra, J., Sloot, P.M.A., Eds.; Springer: Berlin/Heidelberg, Germany, 2009; pp. 893–903. [\[CrossRef\]](#)
66. Li, R.; Saad, Y. GPU-accelerated preconditioned iterative linear solvers. *J. Supercomput.* **2013**, *63*, 443–466. [\[CrossRef\]](#)
67. Abu-Sufah, W.; Karim, A.A. An Effective Approach for Implementing Sparse Matrix-Vector Multiplication on Graphics Processing Units. In Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems, Liverpool, UK, 25–27 June 2012; pp. 453–460. [\[CrossRef\]](#)
68. Choi, J.W.; Singh, A.; Vuduc, R.W. Model-driven Autotuning of Sparse Matrix-vector Multiply on GPUs. In Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Bangalore, India, 9–14 January 2010; pp. 115–126. [\[CrossRef\]](#)
69. Vazquez, F.; Fernandez, J.J.; Garzon, E.M. A new approach for sparse matrix vector product on NVIDIA GPUs. *Concurr. Comput. Pract. Exp.* **2011**, *23*, 815–826. [\[CrossRef\]](#)
70. Kreutzer, M.; Hager, G.; Wellein, G.; Fehske, H.; Basermann, A.; Bishop, A.R. Sparse Matrix-vector Multiplication on GPGPU Clusters: A New Storage Format and a Scalable Implementation. In Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, Shanghai, China, 21–25 May 2012; pp. 1696–1702.
71. Vazquez, F.; Fernandez, J.J.; Garzon, E.M. Automatic tuning of the sparse matrix vector product on GPUs based on the ELLR-T approach. *Parallel Comput.* **2012**, *38*, 408–420. [\[CrossRef\]](#)
72. Dziekonski, A.; Lamecki, A.; Mrozowski, M. A memory efficient and fast sparse matrix vector product on a GPU. *Prog. Electromag. Res.* **2011**, *116*, 49–63. [\[CrossRef\]](#)
73. Monakov, A.; Lokhmotov, A.; Avetisyan, A. Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. In *High Performance Embedded Architectures and Compilers, Proceedings of the 5th International Conference, HiPEAC 2010, Pisa, Italy, 25–27 January 2010*; Patt, Y.N., Foglia, P., Duesterwald, E., Faraboschi, P., Martorell, X., Eds.; Springer: Berlin/Heidelberg, Germany, 2010; pp. 111–125. [\[CrossRef\]](#)
74. Hartwig, A.; Tomov, S.; Dongarra, J. *Implementing a Sparse Matrix Vector Product for the SELL-C/SELL-C- $\sigma$  Formats on NVIDIA GPUs*; Technical Report; University of Tennessee: Knoxville, TN, USA, 2014.
75. Kreutzer, M.; Hager, G.; Wellein, G.; Fehske, H.; Bishop, A.R. A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units. *SIAM J. Sci. Comput.* **2014**, *36*, C401–C423. [\[CrossRef\]](#)
76. Maggioni, M.; Berger-Wolf, T. AdELL: An Adaptive Warp-Balancing ELL Format for Efficient Sparse Matrix-Vector Multiplication on GPUs. In Proceedings of the 2013 42Nd International Conference on Parallel Processing, Lyon, France, 1–4 October 2013; pp. 11–20. [\[CrossRef\]](#)



77. Ashari, A.; Sedaghati, N.; Eisenlohr, J.; Parthasarathy, S.; Sadayappan, P. Fast Sparse Matrix-vector Multiplication on GPUs for Graph Applications. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans, LA, USA, 16–21 November 2014; pp. 781–792. [\[CrossRef\]](#)
78. Wong, J.; Kuhl, E.; Darve, E. A new sparse matrix vector multiplication graphics processing unit algorithm designed for finite element problems. *Int. J. Numer. Methods Eng.* **2015**, *102*, 1784–1814. [\[CrossRef\]](#)
79. Liu, W.; Vinter, B. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In Proceedings of the 29th ACM on International Conference on Supercomputing, Newport Beach, CA, USA, 8–11 June 2015; pp. 339–350. [\[CrossRef\]](#)
80. Blelloch, G.E.; Heroux, M.A.; Zagha, M. *Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors*; Technical Report; DTIC Document: Fort Belvoir, VA, USA, 1993.
81. Hou, K.; Feng, W.; Che, S. Auto-Tuning Strategies for Parallelizing Sparse Matrix-Vector (SpMV) Multiplication on Multi- and Many-Core Processors. In Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Lake Buena Vista, FL, USA, 29 May–2 June 2017; pp. 713–722. [\[CrossRef\]](#)
82. Flegar, G.; Anzt, H. Overcoming Load Imbalance for Irregular Sparse Matrices. In Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms, Denver, CO, USA, 12–17 November 2017; pp. 2:1–2:8. [\[CrossRef\]](#)
83. Anzt, H.; Sawyer, W.; Tomov, S.; Luszczek, P.; Yamazaki, I.; Dongarra, J. Optimizing Krylov Subspace Solvers on Graphics Processing Units. In Proceedings of the IEEE International Parallel Distributed Processing Symposium Workshops, Phoenix, AZ, USA, 19–23 May 2014; pp. 941–949. [\[CrossRef\]](#)
84. Yamazaki, I.; Anzt, H.; Tomov, S.; Hoemmen, M.; Dongarra, J. Improving the Performance of CA-GMRES on Multicores with Multiple GPUs. In Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, 19–23 May 2014; pp. 382–391. [\[CrossRef\]](#)
85. Langr, D.; Tvrdik, P. Evaluation Criteria for Sparse Matrix Storage Formats. *IEEE Trans. Parallel Distrib. Syst.* **2016**, *27*, 428–440. [\[CrossRef\]](#)
86. Davis, T.A.; Hu, Y. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* **2011**, *38*, 1:1–1:25. [\[CrossRef\]](#)
87. Wong, E.J.; Kuhl, E.D. Warppkernel—GPU Sparse Matrix Vector Product Library. 2015. Available online: <https://github.com/thejonwong/warppkernel> (accessed on 12 April 2018).
88. NVIDIA. *CUDA C Best Practises Guide*; NVIDIA: Santa Clara, CA, USA, 2018.



© 2019 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).