2011 International Conference on Parallel Processing

Accelerating Sparse Matrix Vector Multiplication in Iterative Methods Using GPU

Kiran Kumar Matam
CSTAR, IIIT-Hyderabad
Gachibowli, Hyderabad, India 500 032
kiranm@research.iiit.ac.in

Kishore Kothapalli
CSTAR, IIIT-Hyderabad
Gachibowli, Hyderabad, India 500 032
kkishore@iiit.ac.in

***Abstract*—Multiplying a sparse matrix with a vector (`spmv` for short) is a fundamental operation in many linear algebra kernels. Having an efficient `spmv` kernel on modern architectures such as the GPUs is therefore of principal interest. The computational challenges that `spmv` poses are significantly different compared to that of the dense linear algebra kernels. Recent work in this direction has focused on designing data structures to represent sparse matrices so as to improve the efficiency of `spmv` kernels. However, as the nature of sparseness differs across sparse matrices, there is no clear answer as to which data structure to use given a sparse matrix.**

In this work, we address this problem by devising techniques to *understand* the nature of the sparse matrix and then choose appropriate data structures accordingly. By using our technique, we are able to improve the performance of the `spmv` kernel on an Nvidia Tesla GPU (C1060) by a factor of up to 80% in some instances, and about 25% on average compared to the best results of Bell and Garland [3] on the standard dataset (cf. Williams et al. *SC'07*) used in recent literature.

We also use our `spmv` in the conjugate gradient method and show an average 20% improvement compared to using HYB `spmv` of [3], on the dataset obtained from the The University of Florida Sparse Matrix Collection [9].

I. Introduction

In recent years, GPUs have emerged as a viable high performance computing platform due to their performance per unit cost, and performance per watt. For instance, for under $400, one can buy a GPU that can deliver more than a TFLOP of computing power while requiring less than 250 W of power. It is therefore no surprise that 4 out of the top 10 supercomputers [21] use GPUs along with other computing elements. However, this comes at the expense of reinterpreting applications in a highly multithreaded manner. GPUs are good at exploiting massive data parallelism, and are well-suited for applications with regular memory access patterns and high arithmetic intensity. For several regular applications, this approach has been very successful [19], [16]. However, most applications in high performance computing are irregular in nature. Examples include list ranking [27], graph algorithms [18], sparse matrix computations [10], among others. In recent works, efforts are on to arrive at efficient implementations of irregular algorithms on the GPUs [27], [3], and on other recent multi-core architectures such as the Cell [2], [25], [17].

Computations involving sparse matrices, in particular, multiplying a sparse matrix with a vector, is an important operation in several numerical computations. This operation, denoted `spmv`, finds applications in solving systems of linear equations using iterative methods like the conjugate gradient method [10], GMRES [10], iterative methods for finding eigenvalues and eigenvectors of sparse matrices [10], and the like. These methods in turn find applications in many areas of Computer Science such as information extraction, image processing, and the like. These iterative methods consist of both dense linear algebra kernels which can be efficiently implemented, and sparse matrix computations such as `spmv` which are the bottleneck. The importance of `spmv` can be judged by the fact that most multi-core architectures support an optimized library routine for `spmv` [11], [7].

It is therefore not surprising that efforts to improve the performance of `spmv` on the GPUs have been in research focus recently. Some of the difficulties that `spmv` poses on architectures such as the GPUs are irregular memory accesses, load balance amongst threads, coalesced reading of the matrix, among others. In addition, sparse matrices arising in different application settings exhibit varying kinds of sparsity. This makes it difficult to propose general techniques that would be suitable to improve the performance of `spmv` on architectures such as the GPUs. Further, `spmv` kernels may be highly sensitive to even slight architectural changes, and the optimization points very significantly across generations of GPUs [14].

In this context, Bell and Garland [3] proposed data structures to store the sparse matrix so as to improve the performance of `spmv` on the GPUs. They [3] also present the results on the dataset consisting of 14 sparse matrices identified by Williams et al. [25]. These set of 14 sparse matrices arise from various application domains such as quantum chemistry to web graphs and they have attained a benchmark status for recent research on sparse matrices on GPUs [6], [14]. Bell and Garland [3] in their work use representations such as the compressed sparse row (CSR), the ELL, coordinate format (COO), and a hybrid representation that combines the benefits of ELL and COO (For a description of these formats, we refer the reader to [3]). Other prominent works [14], [6], [26] extend the techniques

0190-3918/11 $26.00 © 2011 IEEE
DOI 10.1109/ICPP.2011.82

of [3] by working with blocked/striped representations and studying sparse matrices representing power law graphs to enhance the performance. In most of these works, the choice of the data structures is influenced by the sparsity nature of the matrices considered.

In this work, we propose a new methodology to choose the right data structures to represent sparse matrices. Our methodology uses parameters that attempt to balance the load amongst threads and also identify the right kind of representation, or a combination of representations, to use for storing the sparse matrix. Our experimental results indicate that our technique improves the performance of the `spmv` kernel by an average of 25% when compared to the best results of Bell and Garland [3], on the sparse matrices from the dataset described by Williams et al. in [25] which are presently the benchmark for comparing `spmv` performance.

A. Related Work

Considering the importance of `spmv` and optimizations one has to do at various levels to get good performance on a specific architecture there has been great amount of research on `spmv`. Vuduc [23] extensively studied `spmv` optimizations and auto tuning `spmv` kernel for sequential machines. Williams et al. [25] studied `spmv` on different multi-core architectures, an AMD dual core, an Intel quad-core, STI Cell and the highly multi-threaded Sun Niagara2. They present optimization strategies especially effective for multi-core environments. Their optimization strategies can be classified to low-level code optimizations and data structure optimizations that largely address single-core performance and parallelization optimizations to improve multi-core performance. Blelloch et al. [4], [5] studied `spmv` on vector machines. However, as the architecture and the programming model of GPUs is very different from that of other architectures, many of the optimization strategies on other multi-core platforms may not apply to `spmv` on GPU.

Bell and Garland [3] proposed `spmv` kernels for different sparse matrix formats. They also proposed a new GPU suitable sparse matrix storage format namely HYB which is a combination on ELL and COO. They show that CSR and HYB kernel using texture cache perform well for most of the cases. Baskaran et al. [1] proposed `spmv` kernel for CSR format which gives similar results.

There are also other works that study the sparsity pattern of the matrix on GPU. Choi et al. [6] proposed the BELLPACK data structure which is better suited for matrices which have dense block-substructures. BELLPACK is an extension of ELLPACK with explicit storage of dense blocks to compress data structure and row permutations to avoid unevenly distributed workloads. Yang et al. [26] proposed optimizations for power law graphs which use the texture cache in a better way and increase the memory locality thereby increasing the performance of `spmv`.

Monakov et al. [13] implement blocked `spmv` on GPU. In another work Monakov et al. [14] propose a sparse matrix data structure called Sliced ELLPACK in which a slice of the matrix, a set of adjacent rows, are stored in ELL format. The sizes of the slices may vary. Each slice is assigned to a block of threads in CUDA. Load balancing of threads is achieved by assigning multiple threads to a row if required. But their approach is susceptible to picking up the right slice size for a block.

Vazquez et al. [22] also used the idea of assigning multiple threads to a row to balance the computation per thread. They store the matrix in the ELLPACK-R format which is ELL format along with an array containing the lengths of each row. This approach is best suited for matrices that give good performance when stored in the ELL format.

Anirudh et al. [12] consider using a combination of the CSR and the ELL formats for storing the matrix. They store the rows with nonzero elements fewer than the warp size in ELL format and the remaining rows in CSR format. The applicability of this method to `spmv` is not studied in their paper.

B. Our Results

In this paper, we propose to study the nature of sparsity of a sparse matrix and then use the result of this study to improve the performance of the `spmv` kernel on the GPU. Our work is motivated by the fact that the `spmv` kernel on the GPU can suffer from (i) lack of load balance amongst threads, (ii) memory access coherence in reading the matrix, (iii) the overheads associated with auxiliary information, and the like. Since `spmv` is used in several iterative methods running for hundreds of iterations, a structural analysis of the sparse matrix is a viable option to be explored. In this work, we aim for such a structural analysis of sparse matrices. In summary, our contributions in this paper can be summarized as follows.

- A methodology based on quantities such as the average number of non-zero elements in each row, the deviation in the number of non-zero elements in each row, etc. to effectively balance the load amongst threads in an `spmv` kernel is presented.
- Experimental validation with respect to the benchmark suite of sparse matrices first introduced by Williams et al. in [25] is reported. Our experimental results indicate an average improvement of 25% on an Nvidia C1060 compared to the best results of Bell and Garland [3]. Our experiments were conducted on both an Nvidia Tesla C2050 (Fermi) and an Nvidia Tesla C1060 (Tesla), and on both single precision and double precision calculations.
- Application of our `spmv` method to the commonly used conjugate gradient method is studied. Our experimental results indicate an average improvement of 20% when compared between conjugate gradient method using our

spmv and HYB spmv of [3] on the dataset obtained from the The University of Florida Sparse Matrix Collection [9].

II. Sparse Matrix Formats on GPU

In this section, we discuss spmv kernels using some of the popular sparse matrix formats on GPU. For examples and detailed discussion on the various formats, we refer the reader to [3].

A. Compressed Sparse Row (CSR) Format

The advantage of the CSR format is that it allows a wide variety of matrices to be represented without any wastage of space. A typical way to work with the CSR format in a GPU program is to assign one thread of computation for each row of the matrix. This approach, called the *scalar* CSR kernel in [3], however works well only when the number of nonzero elements per each row is small. Further, when threads access the arrays containing the nonzero elements of the matrix and their corresponding column indices, these accesses are not coalesced in most cases. In the same work [3], the authors describe a *vector* CSR approach wherein a warp of threads are assigned to each row of the matrix. This is known to improve coalescing of the accesses of threads in a half-warp to the arrays containing the nonzero elements of the matrix and their corresponding column indices. Other techniques to improve performance include padding to guarantee alignments [1].

The main drawback of this format however is that on highly unstructured sparse matrices, i.e., sparse matrices that exhibit no particular sparsity pattern, the scalar CSR kernel suffers from huge load imbalance. While this problem may be offset in the vector CSR kernel, the vector CSR kernel works best when each row has at least 32 nonzero elements. Thus, the performance of CSR is highly dependent on the input matrix.

B. Coordinate (COO) Format

This format allows one to represent a variety of sparsity patterns without any wastage in space. When working with this format on GPU, one possibility is to assign one thread to each nonzero element. Each thread therefore reads a row index, a column index, and the corresponding data. It also reads the corresponding x value, and performs one multiplication. Then, a segmented reduction operation is required to add the appropriate values. Segmented reduction is an efficient operation on architectures such as the GPU, and is available as a library routine in CUDPP [8]. In [3], the authors modify the library routine using specific observations that are applicable to this setting. The main draw back when using this format is its high computational intensity (bytes/flop ratio) compared to other formats.

	Bytes/Flop	
Format	32-bit	64-bit
COO	8	12
CSR	6	10
ELL	6	10

Table I
Bytes/Flop ratio of different formats

C. ELL Format

This format suits matrices where the variation in row sizes is small, but for other matrices it can result in space wastage. When working with this format on GPU, one possibility is to assign one thread to each row. This approach was used in [3]. Assuming that the data is stored in a column major order, the accesses of the data and cols matrix are coalesced. This method works well when the variation in the number of nonzero elements among rows is less.

D. Hybrid Format

Of the above formats, the COO and the ELL are suitable in slightly complementary situations. This led [3] to devise a hybrid format, which we summarize as follows. Consider the situation that most rows have roughly k nonzero elements and there may be atypical rows that have more than k nonzero elements. Once such a k is determined, the ELL format is used to store k elements per row. This does not require too much padding and hence minimizes space wastage. The remaining entries are stored in the COO format.

Notice that by examining the matrix and computing statistical quantities such a value of k can be found easily [3]. Computing using the hybrid format is essentially a combination of computing using the ELL and the COO format.

III. Methodology

In this section, we describe our methodology to improve the performance of spmv on GPUs. We focus on improving the efficiency of spmv for general sparse matrices where information regarding sparsity is not known apriori. In the following, we use A to denote the matrix in an spmv operation, x to denote the vector that we multiply with A, and y as the result vector. Essentially, we are computing $y = A \cdot x$ where the matrix A is sparse. We first describe some of the challenges that the spmv kernel poses on architectures such as the GPU. Some of these challenges are:

- Reducing the irregular memory access of the x vector,
- Load balancing among threads,
- Improve the coalescing of reading the matrix A,
- Reducing the load overhead of auxiliary information for each matrix element, and
- Data structure concerns

Some of the observations we use in our approach are given below. As shown in Table I, `spmv` with the CSR format and the ELL format have low computational intensity (bytes/flop). Also the CSR Vector and the ELL kernel access the values and column indices of the A matrix in a coalesced manner. It can be seen that the CSR vector and the ELL offer a good scope for representing general purpose sparse matrices on GPUs. Further, the CSR and ELL formats are suitable under contrasting nature of sparsity. In our work, we combine both these representations in a novel way. To address the issue of load balance when using the CSR vector representation, we assign multiple warps to a row depending on the size of that row. When using the ELL format also, threads assigned to a row in ELL format can have load imbalance, so we assign multiple threads to a row according to the size of that row. But it is not immediate as to which format to use under what circumstances. As in other works [3], [6], we use texture memory for reducing the effect of irregular memory access for the x vector.

In this paper, we present a method for preprocessing the sparse matrix and using that preprocessed information for sparse matrix vector multiplication. We combine both the CSR and the ELL and propose a data structure that utilizes the advantages of both the formats. We find a threshold and use the CSR format for rows that have more nonzero elements than the threshold. The remaining rows are stored in ELL type format. We also find a threshold on the maximum number of nonzero elements for warps assigned to the CSR format and assign multiple warps to a row stored in the CSR format so that the threads are sufficiently load balanced. Similarly, we find a threshold on the maximum number of nonzero elements for a thread assigned to a row in the ELL type format and assign multiple threads to a row in the ELL type format. Details of our approach are presented below.

A. Preprocessing

In this section we describe the preprocessing details before launching the GPU kernel. Algorithm 1 shows the preprocessing algorithm in detail.

We store the matrix in an ELL type format and the CSR format. We consider parameters T, M, and L. Rows with less than T nonzero elements are stored in the ELL type format and the remaining rows are stored in the CSR format. Let the parameter M refer to the maximum number of elements each thread will process for the rows stored using ELL type format. If a row in the ELL type format has more than M nonzero elements, then we assign multiple threads to process that row. Let the parameter L refer to the maximum number of elements each warp will process for the rows stored using CSR format. If a row in CSR format has more than L nonzero elements, then we assign multiple warps to process that row.

Algorithm 1 Preprocessing(A, T, L, M)

1: //Let R be the set of rows of the matrix A. R_{csr} and R_{ell} be the set of rows that are in stored CSR and ELL type format respectively. Let r_{nnz} be the number of nonzero elements in a row r.
2: **for** each row $r \in R$ **do**
3: **if** $r_{nnz} \geq T$ **then**
4: $r_{nnz} \in R_{csr}$
5: **else**
6: $r_{nnz} \in R_{ell}$
7: **end if**
8: **end for**
9: Let $CSRmap$ be the array such that $CSRmap[i] =$ row index of i^{th} CSR row in the matrix A. Similarly $ELLmap$ be the array such that $ELLmap[i] =$ row index of i^{th} ELL row in the matrix A.
10: *//ELL Preprocessing*
11: Sort the rows in R_{ell} based on the number of nonzero elements in the row. Update $ELLmap$ accordingly.
12: From the above sorted rows, to a warp i assign a set of consecutive rows w_{rows_i}.
13: Let r_{max_i} be the maximum nonzero elements of a row in w_{row_i}.
14: Assign $\lceil \frac{r_{max_i}}{M} \rceil$ consecutive threads in warp i to rows in w_{row_i}.
15: Let $ELLdata$ and $ELLcols$ be the arrays that holds the elements and column indices of the rows in ELL warps.
16: **for** each warp i in R_{ell} **do**
17: store the row to which the starting thread of the i^th warp belongs to.
18: store the start index of $ELLdata$ that the warp needs to process.
19: transfer the values, column indices of the elements in w_{rows_i} rows to $ELLdata$ and $ELLcols$ respectively so that the warp can read the values from the arrays in a coalesced manner.
20: **end for**
21: *//CSR preprocessing*
22: Let $CSRvals$ and $CSRcols$ be the arrays that hold the values and column indices of the rows in R_{csr}.
23: **for** each row r in R_{csr} **do**
24: assign $\lceil \frac{r_{nnz}}{L} \rceil$ warps to the row r
25: Transfer the elements and corresponding column indices of the row r to $CSRvals$ and $CSRcols$ respectively.
26: for each warp assigned to row r, find the start and end index of its portion in the $CSRvals$ array.
27: **end for**

$$
\begin{aligned}
data &= \begin{bmatrix} 1 & 2 & 4 & 1 & 3 & 2 & 1 \end{bmatrix} \\
cols &= \begin{bmatrix} 1 & 3 & 4 & 1 & 2 & 5 & 6 \end{bmatrix} \\
ptr &= \begin{bmatrix} 1 & 4 & 8 \end{bmatrix} \\
ELLdata &= \begin{bmatrix} 1 & 2 & 1 & 3 & 4 & * & 2 & 1 \end{bmatrix} \\
ELLcols &= \begin{bmatrix} 1 & 3 & 1 & 2 & 4 & * & 5 & 6 \end{bmatrix}
\end{aligned}
$$

Figure 1. Let the warpsize be 4, the above 2 rows stored in CSR format be assigned to a warp and 2 threads be assigned to each row. $ELLdata$ and $ELLcols$ show the way we store the ELL rows. * refers to the padded element.

In Algorithm 1, lines 11-20 show our preprocessing method for the rows stored in the ELL type format. For the rows assigned to a warp in an ELL type format, values and the corresponding column indices are stored in an interleaved fashion among the rows. An example is shown in Figure 1. In GPU's, threads in the warp execute in synchronized manner. So in the ELL type format we assign a group of rows to a warp. We assign a group of shorter rows to one warp so that after the warp finishes its work it can continue with the other rows. We do extra padding to see that all the rows in the warp are of same length(r_{max_i}), i.e., the maximum row size in that warp, so that there can be coalesced memory access as in ELL. We assign an equal number of threads to a row in the warp so that we need to bring the size of the row and the row to which the thread belongs only once per warp. We see that threads of the same row are in a warp, so that we need no synchronization between different warps. Some of the threads in the warp may not be assigned to any row, but the number of such threads will be very low. As we assign multiple threads to a row, the thread index of a thread may not give us enough information as to which row it belongs to. This requires that we keep additional information along with each warp. For the first thread of a warp we store the row to which it belongs to. Given a thread index t and the row number to which the first thread of this warp belongs to, the row number to which thread t belongs can be computed on the fly. Similarly we need to store the starting index of the values of the matrix that warp needs to process.

In the Algorithm 1, lines 22-27 show our preprocessing method for the rows stored in the CSR format. The tuning of the parameters T, M, and L is considered in Section III-B.

B. Parameter tuning

In this section, we discuss how to choose the parameters T, M, and L given a matrix. Consider a row with k nonzero elements and one warp of 32 threads working for this row. In each iteration, each of the 32 threads perform one computation. This leaves $k \bmod 32$ elements for the last iteration in which only $k \bmod 32$ threads will be active. However, the fraction of inactive threads decreases as k increases because there will be more iterations where all threads are active. Therefore, it can be observed that CSR vector works best when the row sizes are large. We note that the ELL format tends to outperform the CSR vector format for rows with less than 256 elements. This also matches our intuition explained earlier in this paragraph. So, we see that $T \leq 256$. We want that most rows with less than 256 elements are processed using the ELL format. To determine T exactly, we compute the average of the sizes of rows with less than 256 elements. Hence, we pick T to be the multiple of 32 that is closest and greater than the above computed average.

Let us denote the number of elements that a thread (all the threads in a warp) has (have) to process as the *work load* of that thread (warp). We call a warp as a *CSR warp* (*ELL warp*) if the warp is processing a row using the CSR vector (*resp.* ELL) format. Recall from Algorithm 1 that L indicates the maximum work load of any CSR warp, and M indicates the maximum work load of any thread in an ELL warp. To determine L and M, we proceed as follows.

We first determine the maximum work load of any thread in an ELL warp. There are two considerations here. Firstly, in our `spmv` kernel, ELL warps have to fetch some additional book-keeping values that are required for the correct operation of the kernel. Given this, we note that each thread in such a warp has to process at least 6 elements so as to offset the above overhead with useful computation. A second consideration is to ensure that the work load variation across threads in ELL warps is kept at a minimum so as to improve load balance. One way to minimize the work load variation is to find the average work load of ELL warps. This average is already computed when determining the value of T in the previous paragraph. If this average is less than 6, then we can set M to be 6. If the above average is more than 32, we set M to be 32 so as to ensure that there is not much work load imbalance between threads processing rows with fewer than average number of elements and threads processing rows with average number of elements. So, $6 \leq M \leq 32$, and rows using ELL format with more than M elements are given multiple threads.

Since in our `spmv` kernel, both CSR warps and ELL warps are launched simultaneously, also CSR warps should use the above average as the maximum work load. This means that L can indeed be set to be equal to $M \cdot warpsize$ (since M is the maximum work load of a thread) and rows with more than L elements are given multiple warps. If all rows have more than 256 nonzero elements, we set the value of L to be equal to $32 \cdot warpsize$. Note that there will not be much work load imbalance as all the rows are greater than 256.

To test the validity of tuning parameters obtained by our methodology, we ran our program by varying the values of T, M, and L on matrices shown in the Figure 3. The values of T were varied from 8 to 512 in increments of 32, M were varied from 4 to 320 in increments of 16, and of L were varied from $4 \cdot warpsize$ to $64 \cdot warpsize$ in increments

of $8 \cdot warpsize$. We have observed that the results obtained by setting the parameters according to our methodology are close to the results obtained by the using the best possible values for parameters T, M, and L for that matrix.

C. Implementation details

We implement the `spmv` based on the CSR format and `spmv` based on the ELL type format in the same kernel, so that even if any of the format has few number of rows and does not fully occupy the GPU, it does not affect the occupancy of SMs. We see that warps execute either `spmv` based on CSR or ELL type format and there is no divergence among the threads in warp. We use a block size of 128 and 256 in Tesla C1060 and Tesla C2050 respectively so that the SMs are fully occupied. We use the texture cache for accessing the x vector so that the effect of irregular memory access due to the x vector is reduced. Bell and Garland [3] also show that using the texture cache for accessing the x vector is beneficial.

In the `spmv` based on the CSR format of the matrix, after each warp performs the multiplication and addition of the values, if it is the only warp assigned to a row we write the result to the resultant vector (y in $y = Ax$). If there are multiple warps assigned to the row we need to reduce the sum across all the warps in that row. We can reduce the sum for the warps in the same block and use atomic operations for the warps of the same row that are in different blocks. But the C1060 does not support atomic operations for float and double. So we write the result of the warp into another array and use a segmented reduction like algorithm as described in [3].

In the `spmv` based on the ELL type format where multiple threads can be assigned to a row, we need to perform a reduction across all the threads assigned to that row. After each thread computes its result, we perform a segmented reduction in shared memory across the threads in the warp and after that one of the thread in a row writes the result to the y vector.

IV. Results

In this section, we report the results of our experiments. The experiments were run on the following systems:

- **CPU:** An Intel Core i7 980x, with 12 MB cache, 12 GB RAM and a 6.4 GT/s Quick path interface, with maximum memory bandwidth of 25 GB/s.
- **Tesla C1060 GPU:** A Tesla C1060 which is one quarter of a Tesla S1070 computing system with 4 GB memory and 102 GB/s memory bandwidth. It is attached to an Intel Core i7 CPU, running CUDA Toolkit/SDK version 3.2.
- **Tesla C2050 GPU:** A Tesla C2050 GPU card with 3 GB memory and 144 GB/s memory bandwidth. It is attached to an Intel Core i7 CPU, running CUDA Toolkit/SDK version 3.2. [15].

In the following subsections we study our `spmv` method on the synthetic dataset and the standard dataset.

A. Experiments on Synthetic Datasets

To illustrate the efficacy of our approach, we consider a few experiments on synthetic data sets. The synthetic data sets correspond to highly unstructured sparse matrices. One kind of highly unstructured matrices are those where there are very few rows with very large number of nonzero elements compared to the rest of the rows.

We now perform experiments that illustrate the benefits of our ELL approach using synthetic data sets. When using the ELL format [3], each row is assigned one thread. In our approach, we assign multiple threads to each row depending on the number of nonzero elements in the row. We perform three experiments keeping the maximum number of nonzero elements across rows as 64, 96, and 128 in each experiment respectively. This is motivated by the fact that the ELL format works best for small row sizes. In each experiment we take three density functions. In these density functions, the percentage of rows with sizes in the first and last quarter of the maximum row size are varied. In the three density functions, percentage of rows with number of nonzero elements in the first quarter of the maximum row size are set to 60%, 38%, and 15% respectively. Similarly the percentage of rows with number of nonzero elements in the last quarter of the maximum row size are set to 10%, 32%, and 55% respectively. The column indexes for the nonzero elements in a row are chosen uniformly at random. In the experiments, all rows are processed using the ELL format only. As the Tesla C1060 can keep $1024 \times 32 = 32768$ threads active at any time, the matrices we generate have number of rows that is a multiple of 32768. The results of our experiment are shown in Figure 2. As can be seen, for all instances, load balancing does help in improving the throughput. Our approach also benefits from a simple reordering of rows based on the number of nonzero elements so that in a warp, most threads are load balanced. Notice that this reordering is very less time consuming as it involves only sorting of rows based on the number of nonzeros in a row. A similar experiment is performed to study our approach for matrices processed using the CSR format. We also observe an improvement in performance by assigning multiple warps to a large row in CSR format. For the `spmv` based on CSR format as the number of rows increases the effect of load imbalance tends to reduce, but the number of rows till which we can observe load imbalance is fairly large.

B. Experiments on the Standard Dataset from [25]

We use the standard dataset from the influential work of Williams et al. [25]. This dataset contains 14 sparse matrices arising from diverse fields such as finite element

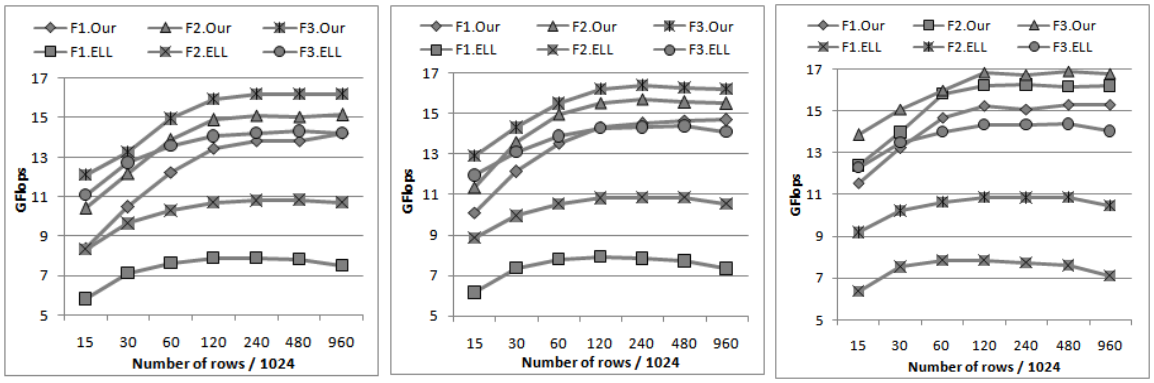

Figure 2. Performance comparison (in GFlops) on synthetic data sets. x−axis labels are (number of rows)/1024. The maximum row sizes in the plots are 64,96, and 128 respectively.The series Fi.Our (Fi.ELL) shows our `spmv` (resp. ELL `spmv` of [3]) method on i^{th} density function.

Matrix	Rows	NNZ	NNZ/Row
Dense	2,000	4,000,000	2000.0
Protein	36,417	4,344,765	119.3
FEM/Spheres	83,334	6,010,480	72.1
FEM/Cantilever	62,451	4,007,383	64.1
Wind Tunnel	217,918	11,634,424	53.3
FEM/Harbor	46,835	2,374,001	50.6
QCD	49,152	1,916,928	39.0
FEM/Ship	140,874	7,813,404	55.4
Economics	206,500	1,273,389	6.1
Epidemiology	525,825	2,100,225	3.9
FEM/Accelerator	121,192	2,624,331	21.6
Circuit	170,998	958,936	5.6
Webbase	1,000,005	3,105,536	3.1
LP	4,284	11,279,748	2632.9

Figure 3. List of sparse matrices. Number of columns and rows are equal for all the matrices except for the matrix LP, where the number of columns is equal to $1,092,610$.

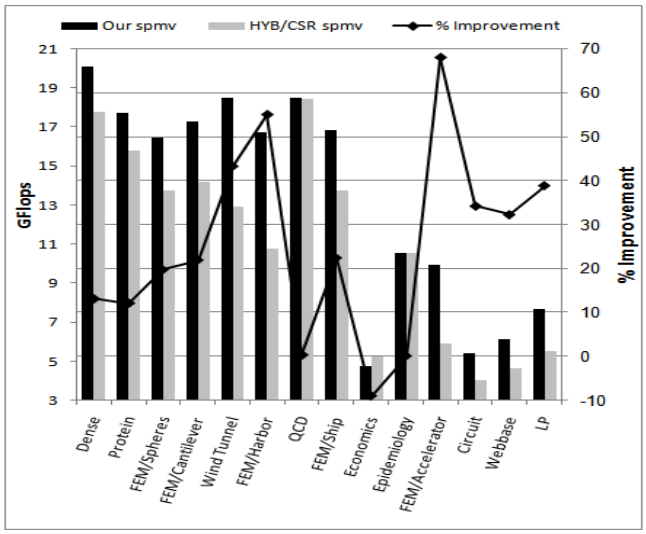

Figure 4. Performance comparison (in GFlops) on Tesla C1060 using single precision floating operations. *% Improvement* series is aligned to secondary axis (y-axis on the right side). Our spmv method is compared with the best results obtained by executing either CSR vector or HYB `spmv` kernel using texture cache of [3], which in turn are the best results of [3].

method-based modeling, circuit simulation, linear programming, a connectivity graph collected from a partial web crawl, among others. The list of matrices in this dataset is reproduced in Figure 3. The dataset contains matrices of varying nature of sparsity with some matrices having few nonzero elements per row, a dense matrix, and some highly unstructured matrices such as LP, and Webbase.

C. Results

As in prior works we use GFlops to measure the performance of our `spmv` method. We report the results by performing several runs for each kernel and taking the average over multiple runs. We compare our work with the kernels that use the texture cache for storing the vector x as this approach gives the best results. Figures 4–7 show the performance measured in terms of GFlops achieved on the matrices shown in Figure 3. The performance achieved is compared with that from the work of [3]. The label *HYB/CSR spmv* refers to the best performance reported from [3][1] obtained using either the CSR vector or the HYB format. The label *Our spmv* refers to the format we described in this paper.

We see that our `spmv` method is performing better on most of the matrices. In the matrix LP a large fraction of rows are stored in CSR format in our `spmv` method. The improvement in performance is due to assigning more warps to a row and hence better load balancing among them. In matrices FEM/Accelerator and Protein, a large fraction of rows are stored in both the CSR and the ELL formats. In other matrices where we see improvement in performance we see that most of the elements are stored in the ELL format and the improvement in performance is due to better

[1]The program corresponding to [3] is run by us in reporting the above results.

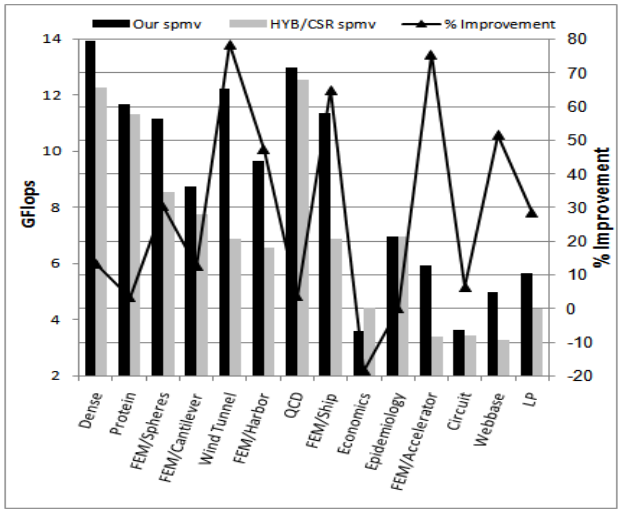

Figure 5. Performance comparison (in GFlops) on Tesla C1060 using double precision floating operations. *% Improvement* series is aligned to secondary axis (y-axis on the right side). Our spmv method is compared with the best results obtained by executing either CSR vector or HYB `spmv` kernel using texture cache of [3], which in turn are the best results of [3].

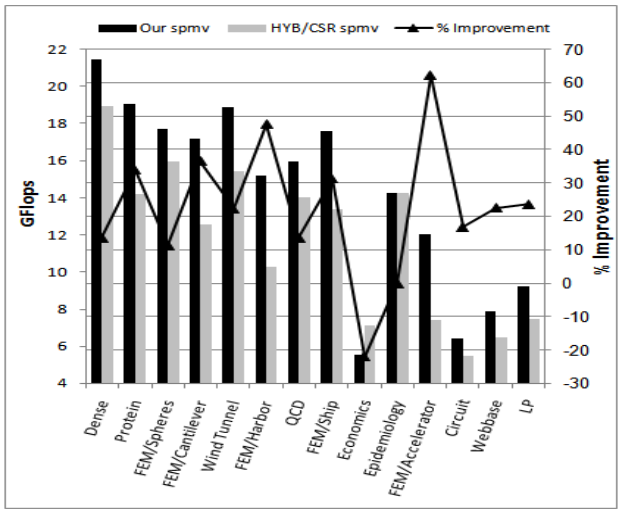

Figure 6. Performance comparison (in GFlops) on Tesla C2050 using single precision floating operations. *% Improvement* series is aligned to secondary axis (y-axis on the right side). Our spmv method is compared with the best results obtained by executing either CSR vector or HYB `spmv` kernel using texture cache of [3].

ELL storage and assigning more threads to an ELL row. For the matrix Epidemiolgy, in the preprocessing stage we see that all the rows have very few nonzero elements and the standard deviation of the number of nonzero zero elements in the rows is also very less. So in such cases where our method does not really benefit we use the HYB `spmv` of [3]. In the case of the matrix Economics, we see that in the the `spmv` based on HYB format most of the elements are stored in the ELL format which performs poorer than `spmv` based on CSR format due to reduced cache hit ratio.

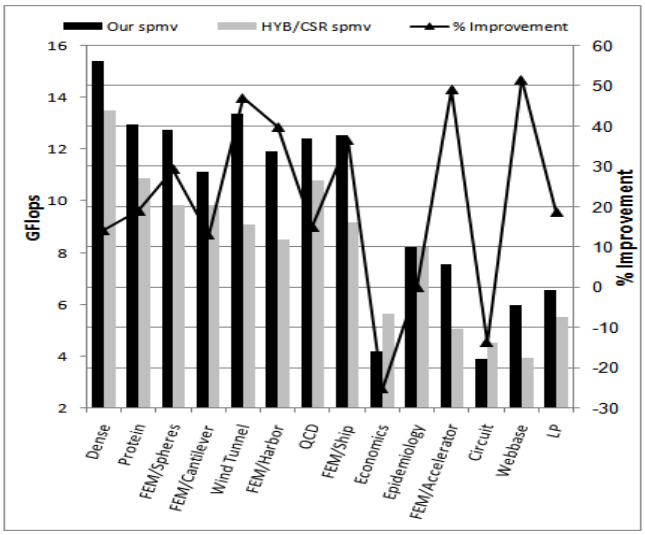

Figure 7. Performance comparison (in GFlops) on Tesla C2050 using double precision floating operations. *% Improvement* series is aligned to secondary axis (y-axis on the right side). Our spmv method is compared with the best results obtained by executing either CSR vector or HYB `spmv` kernel using texture cache of [3].

In our `spmv` method also most of the rows are stored in ELL format and due to the reduction in cache hit ratio our `spmv` kernel performs poorer than `spmv` based on CSR format.

The theoretical peak bandwidth between the device memory and the symmetric multiprocessors (SMs) on Tesla C1060 is 102 GB/s . On Tesla C1060, it was observed that our kernels are performing well on most of the matrices. But on some matrices the reduction in bandwidth usage is significantly due to texture cache misses. On Tesla C2050, it was observed that the bandwidth used by most of our kernels is below the theoretical peak of 144 GB/s. Some reasons for this could be that on Tesla C2050, our kernel uses the texture cache to store the x vector. Since Tesla C2050 supports a hardware managed L1 cache, it turns out that leaving the device to manage the access to the x vector via the L1 cache is a better choice. The same effect is seen on the CUSP kernels, corresponding to the work of [3], that run on Tesla C2050 without using texture cache.

The above discussion indicates that the optimization points on Tesla C1060 and Tesla C2050 are very different. We have attempted to suggest approaches that are applicable to both the architectures but some fine-tuning may be required in the end to achieve the best possible performance.

D. Comparison with other works

Other recent works on GPUs [6], [26], [13], [22] in the literature on `spmv` exploit particular sparsity patterns of the matrix and use data structures and optimizations suitable to that sparsity pattern. However for matrices that do not follow that sparsity pattern these works may perform poorly.

Monakov et al. [14] do optimizations for general sparse matrices. They present results on the same dataset used in

Section 3. However, a direct comparison with their work may not be apt as spmv is memory bound and the peak memory bandwidth between the architectures we work on, the Tesla C1060, and the GTX 280 used in [14] differ. But, to attempt a comparison, we consider the percentage improvements of our result and the result from [14] with respect to respective baseline implementations. In [14], the baseline used is HYB spmv from [3]. For the purposes of this section, we use HYB spmv [3] as the baseline. With such a comparison, we see that our kernels are performing better than those from [14] in 10 of the 14 instances of the dataset under consideration when using single precision operations. Further Monakov et al. [14] present the best results obtained after performing an exhaustive search for several configuration parameters including slice size, thread block size, reordering parameter and work per thread. In our approach, we attempt to compute such parameters without recourse to such exhaustive searching.

Bell et al. [3] in their paper show that GPU based spmv outperforms other multi-core architectures. Comparing with the Intel MKL library that is highly optimized for multi-cores on Intel Xeon 5550, 2.67GHz from a recent benchmarking effort of [20] our spmv kernel is about 2 to 8 times faster.

V. Discussion

In this section, we discuss certain issues that are relevant to our work.

A. Additional overheads

One point to note with respect to our work is the additional code and storage overhead introduced by our method. Unlike the CSR Vector proposed in [3], when multiple warps are assigned to the same row, we require that the partial results of these multiple warps be reduced in parallel to produce one value in the output vector y. This does add some overhead in terms of code, but the overhead is observed to be negligible. A similar overhead shows in our approach when using the ELL type format also where partial results from multiple threads assigned to the same row have to be reduced to produce one value in the output vector y. Again, this overhead is noticed to be very small. Another approach to do the parallel reduction is to use atomic operations. Since very few threads/warps compete for exclusive access, atomic operations may also offer a better choice in our setting.

Our approach also results in some data structure and operational space overhead. Data structure overhead comes when using the ELL type representation for some rows of a given sparse matrix. This overhead may be applicable in other works also [3]. Operational overhead in our approach may arise in some settings. For instance, when assigning multiple warps to each row, each warp has to know the starting and ending indices of columns in that row that will be processed by this warp. Similarly, when using the parallel reduction method to add the partial results from each warp, we need some auxiliary space to do the reduction. But, compared to the actual number of non-zero elements, this extra space is quite small. For instance, for the dataset shown in Figure 3 the storage overhead is on an average 2 % and is 6% at maximum.

Our approach may result in altering the texture cache miss ratio of accessing the x vector. This happens as we are reordering rows processed in the ELL type format by sorting them according to their size. Our approach stands to benefit from an improved texture cache model in future GPUs.

B. Preprocessing Time

Finally, we shift our attention to the time taken by our approach to arrange the sparse matrix in the appropriate data structures. We see that the time taken for preprocessing on CPU and to transfer the data structures to the GPU for the dataset in Figure 3 is on average equivalent to 45 calls to our spmv kernel on a Tesla C1060 GPU.

VI. Application to Conjugate Gradient Method

To illustrate the efficiency of our spmv method, we apply it to the commonly used conjugate gradient method (CG in short). CG method is an iterative method used for solving sparse symmetric positive definite linear systems. In each iteration spmv and few BLAS1 operations are used. In this comparison, our focus is to study the gains obtained by using the spmv kernel described in this paper. Variations to the CG method such as preconditioning [24] may only fasten the convergence of the CG method, but the overall algorithmic structure remains unchanged. For illustrative purposes, we therefore consider the basic CG method [24] and show the improvements obtained with the proposed spmv technique.

We perform the preprocessing stage of our spmv method on the CPU while the GPU simultaneously executes the iterations of the CG method using the HYB spmv from [3]. After preprocessing stage we transfer the data obtained by our methodology to the GPU and we use our spmv method in the remaining iterations of the CG method.

We study our spmv method on the dataset obtained from the The University of Florida Sparse Matrix Collection[9]. From this dataset, we consider all matrices which have more than 10^5 rows and are symmetric, positive definite with real valued entries. Figure 8 shows the performance comparison in terms of GFlops between CG method using our spmv kernel and using the HYB spmv from [3]. We see that on most matrices, CG method using our spmv method outperforms CG using HYB spmv method from [3].

References

[1] M. M. Baskaran and R. Borderland. Optimizing sparse matrix-vector multiplication on gpus. Technical report, IBM RC24704, 2008.

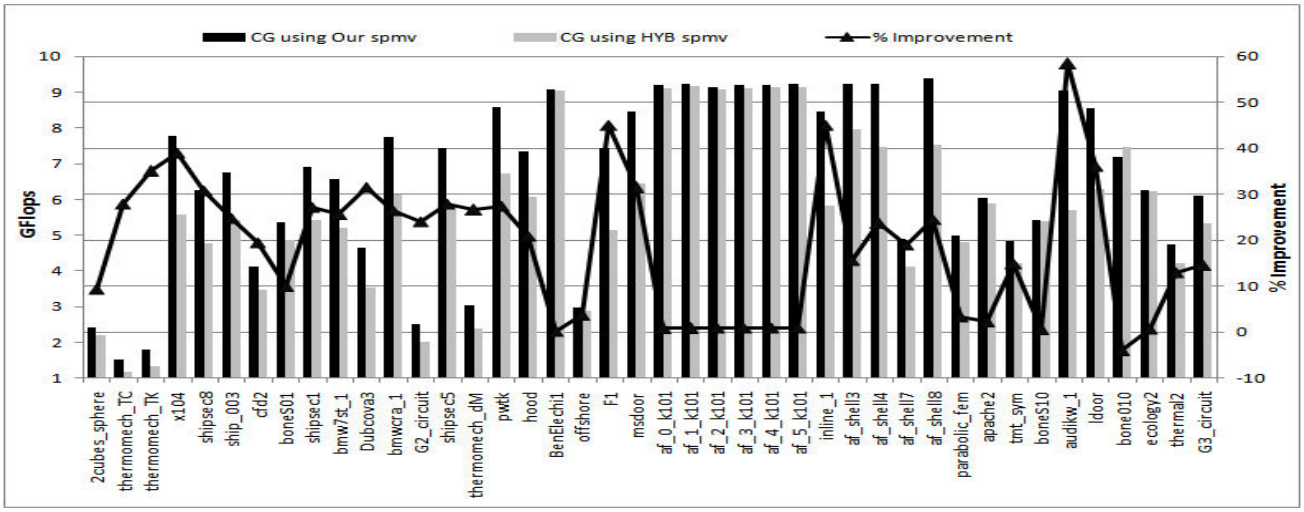

Figure 8. Performance comparison (in GFLOPS) between conjugate gradient method using our `spmv` method and using HYB `spmv` method of [3] on Tesla C1060 using double precision floating point operations. *% Improvement* series is aligned to secondary axis (y-axis on the right side).

[2] D.A. Bader, V. Agarwal, and K. Madduri, "On the Design and Analysis of Irregular Algorithms on the Cell Processor: A case study on list ranking," 21th IPDPS, Long Beach, CA, March 26-30, 2007.

[3] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In Proc. of SC '09, ACM, New York, NY, USA, , Article 18 , 11 pages.

[4] G. E. Blelloch, J. C. Hardwick, S. Chatterjee,J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In PPOPP' 93, pages 102–111, 1993.

[5] G. E. Blelloch, M. A. Heroux, and M. Zagha. Segmented operations for sparse matrix computation on vector multiprocessors. Technical report, Pittsburgh, PA, USA, 1993.

[6] Jee W. Choi, Amik Singh, and Richard W. Vuduc. 2010. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In Proc. PPoPP '10. ACM, New York, NY, USA, 115-126.

[7] Cusp-library, http://code.google.com/p/cusp-library/

[8] CUDPP: CUDA Data-Parallel Primitives Library. http://www.gpgpu.org/developer/cudpp/, 2009.

[9] T. A. Davis and Y. Hu, The University of Florida Sparse Matrix Collection, ACM Transactions on Mathematical Software (to appear), http://www.cise.ufl.edu/research/sparse/matrices.

[10] James W. Demmel. 1997. Applied Numerical Linear Algebra. Soc. for Industrial and Applied Math., Philadelphia, PA, USA.

[11] Intel Math Kernel Library, http://software.intel.com/en-us/articles/intel-mkl/

[12] A. Maringanti, V. Athavale, and S. Patkar. Acceleration of conjugate gradient method for circuit simulation using CUDA, in Proc. HiPC, pp. 438–444, 2009.

[13] Alexander Monakov and Avetisyan, Arutyun. Implementing Blocked Sparse Matrix-Vector Multiplication on NVIDIA GPUs, in Proc. of SAMOS 2009, pp. 289-297.

[14] Alexander Monakov, Arutyun Avetisyan, and Anton Lokhmotov. Automatically tuning sparse matrix-vector multiplication for GPU Architectures, in Proc. of HiPEAC, pp. 111–125, 2010.

[15] NVIDIA Corporation. CUDA: Compute unified device architecture programming guide. Technical report, NVIDIA.

[16] Nadathur Satish, Mark Harris, and Michael Garland. 2009. Designing efficient sorting algorithms for manycore GPUs. In Proc. of IPDPS '09. IEEE Computer Society, Washington, DC, USA, pp: 1-10.

[17] Daniele Paolo Scarpazza, Oreste Villa, and Fabrizio Petrini. 2008. Efficient Breadth-First Search on the Cell/BE Processor. IEEE Trans. Parallel Distrib. Syst. 19, 10 (October 2008), pp: 1381-1395.

[18] Pawan Harish, P. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In Proc. of HiPC 2007, Vol. 4873, pp. 197-208.

[19] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. In Proc. of ACM Symposium on Graphics hardware, pp: 97-106, 2007.

[20] Tesla C2050 Performance Benchmarks, Available at www.microway.com/pdfs/TeslaC2050-Fermi-Performance.pdf.

[21] The Top 500 Supercomputing sites. Available at http://www.top500.org.

[22] F. Vazquez, G. Ortega, J.J. Fernndez, E.M. Garzn, "Improving the Performance of the Sparse Matrix Vector Product with GPUs," cit, pp.1146-1151, 2010 10th IEEE International Conference on Computer and Information Technology, 2010

[23] R. Vuduc. Automatic performance tuning of sparse matrix kernels. PhD thesis, UC Berkeley, USA, December 2003.

[24] Y. Saad. 2003. Iterative Methods for Sparse Linear Systems (2nd ed.). Soc. for Industrial and Applied Math., Philadelphia, PA, USA.

[25] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2007. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In Proc. of SC '07. ACM, New York, NY, USA, , Article 38 , 12 pages.

[26] Xintian Yang, Srini Parthasarathy, and P. Sadayappan, "Fast Sparse Matrix-Vector Multiplication on GPUs: Implications for Graph Mining," 12 pp. OSU-CISRC-2/10-TR05.

[27] Zheng Wei, Joseph JaJa. Optimization of linked list prefix computations on multithreaded GPUs using CUDA. In Proc. IPDPS 2010, pp. 1-8.