An Efficient Two-Dimensional Blocking Strategy for Sparse Matrix-Vector Multiplication on GPUs

Arash Ashari, Naser Sedaghati, John Eisenlohr, and P. Sadayappan
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{ashari,sedaghat,eisenloh,saday@cse.ohio-state.edu}

ABSTRACT

Sparse matrix-vector multiplication (SpMV) is one of the key operations in linear algebra. Overcoming thread divergence, load imbalance and non-coalesced and indirect memory access due to sparsity and irregularity are challenges to optimizing SpMV on GPUs.

In this paper we present a new blocked row-column (BRC) storage format with a novel two-dimensional blocking mechanism that effectively addresses the challenges: it reduces thread divergence by reordering and grouping rows of the input matrix with nearly equal number of non-zero elements onto the same execution units (i.e., warps). BRC improves load balance by partitioning rows into blocks with a constant number of non-zeros such that different warps perform the same amount of work. We also present an efficient autotuning technique to optimize BRC performance by judicious selection of block size based on sparsity characteristics of the matrix. A CUDA implementation of BRC outperforms NVIDIA CUSP and cuSPARSE libraries and other state-of-the-art SpMV formats on a range of unstructured sparse matrices from multiple application domains. The BRC format has been integrated with PETSc, enabling its use in PETSc's solvers.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*Parallel Programming*; G.1.0 [**Mathematics of Computing**]: Numerical Analysis—*Parallel Algorithms*

Keywords

BRC; CUDA; GPU; SpMV

ICS'14, June 10–13 2014, Munich, Germany.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2642-1/14/06 ...$15.00.
http://dx.doi.org/10.1145/2597652.2597678 .

1. INTRODUCTION

In the last decade, there has been a growing trend in the use of many-core throughput-oriented architectures in scientific computing. In particular, with the emergence of programmer-friendly APIs such as OpenCL [12] and CUDA [6, 14], scientists from a broad range of disciplines have started leveraging the computation throughput of GPUs. Sparse Matrix-Vector multiplication (SpMV) is one of the computational operations that have received much attention since it is a core kernel used in many algorithms such as iterative methods for solving large-scale linear systems. Recently, the high performance conjugate gradient (HPCG) benchmark, in which SpMV is one of the main kernels, was announced as a new benchmark for ranking high performance computing Systems [9].

GPUs are very well suited for dense matrix computations, but several challenges are faced in achieving high performance for sparse matrix computations. In the case of SpMV ($y = y + Ax$), sparsity and irregularity of the matrix A cause a) irregular and non-coalesced accesses to both matrix A and vector x, b) load imbalance among threads and warps, and c) thread divergence at the warp level. cuSPARSE [8] and CUSP [4, 7] are two widely-used CUDA libraries that support different sparse matrix formats, e.g. Diagonal (DIA), ELLPACK (ELL), Compressed Sparse Row (CSR), Coordinate (COO), and also hybrid (HYB) which combines ELL and COO. As the best-performing format, HYB splits the matrix into two parts: a compressed part that contains typical number of non-zeros per row well-suited to ELL, and a sparser part with the remaining non-zeros, suited to COO format. However HYB suffers from performing redundant computations (inherent in the ELL part) and also redundant data transfer (due to the padded elements). Several studies, [3, 5, 23], proposed enhanced formats that work better than HYB for certain types of matrices. All these methods achieve coalesced accesses for the matrix A, but lack the generality of HYB and so cannot outperform HYB in general. The main shortcoming of existing methods is the lack of an adaptive format that can tune itself for different matrix structures and achieve consistently superior performance across matrices from various application domains.

In this paper we propose a new adaptive format that addresses intra-warp thread divergence, redundant computation and redundant data transfer introduced by the ELL format. We also address synchronization overhead caused by the reduction/atomic operations in COO. The proposed format, blocked row-column (BRC), is a hybrid sparse matrix representation with the property that in an SpMV operation each warp is assigned the same number of rows (a block of 32 rows) and all the rows have equal non-zero elements less than or equal to a tile size. By using a dense structured blocked format, BRC alleviates thread divergence and redundant computation while achieving a load balanced execution. We also propose an auto-tuning framework for the model parameter – width of block along matrix column. It achieves 96% of the performance obtained from exhaustive search in a bounded domain of this parameter.

On an NVIDIA Kepler GPU, the CUDA implementation of auto-tuned BRC is up to 4.8× and 4.3× (and average 2.7× and 2.5×) faster than the HYB in single and double precision, respectively. BRC also achieves a maximal speedup of 3.6× and 2.2× (and average 2× and 1.7×) over the CSR (the most commonly used format) for single and double precision, respectively. Integrating BRC into PETSc [1, 2] shows that using BRC as the SpMV kernel reduces the total runtime by 16% and 70% for *ILU(0)* and *Polynomial* preconditioners, respectively and when compared to the PETSc AIJ-CUSP format which uses NVIDIA CUSP [7] for SpMV.

The rest of the paper is organized as follows: Section 2 reviews the existing SpMV formats. In Section 3, we describe the BRC format. Section 4 describes evaluation methodology and Section 5 presents the results. Section 6 describes integration of BRC with PETSc. Related work is discussed in Section 7. We conclude in Section 8.

2. BACKGROUND

In this paper, we target the SpMV problem in the form $y = y + Ax$ where y and x are one-dimensional vectors and A is a two-dimensional sparse matrix. Due to sparsity of the matrix, there are many computations and memory usage that can be ignored (e.g. zero elements) in order to save memory and computational resources. To do so, many formats/optimizations have been proposed. Bell and Garland [4] and Vuduc [21] have reviewed some of the existing formats. In this section, we briefly review those that are related to our proposal and against which we have compared our results.

In order to illustrate different formats, we use an example sparse matrix A with 10 non-zero elements (namely, A to J) distributed unevenly across 4 rows and 6 columns (thus leaving total number of 14 zero elements):

$$A = \begin{bmatrix} 0 & A & 0 & B & 0 & 0 \\ 0 & 0 & C & 0 & D & 0 \\ E & F & G & 0 & H & I \\ 0 & 0 & 0 & J & 0 & 0 \end{bmatrix} \tag{1}$$

2.1 Coordinate (COO) Format

COO is a very simple format in which the sparse matrix A is transformed into three dense vectors: *data* that contains only the non-zero data values, *column index* that contains the column index of the elements corresponding to data vector, and *row index* that contains the row index of the elements corresponding to data vector

Figure 1-a shows matrix A in COO format. The SpMV kernel that receives matrix in COO format assigns every non-zero element to a separate GPU thread. As a result, atomic operation is used to collect contributions of different threads (mapped to the same row) and to finalize the reduced results in vector y [4]. One major drawback to COO format is the use of atomic operations especially when uneven distribution of non-zero elements per row causes some rows to be denser than others. Such a distribution has a drastic impact on performance because of unbalanced executions across threads. Even though improvement attempts such as segmented reduction [4, 20] have been proposed to decrease this overhead, it is still not completely invariant to the distribution of the non-zero elements per row [4].

2.2 Compressed Sparse ROW (CSR) Format

CSR works at the granularity of threads per row(s). This format is similar to COO with the difference that CSR does not need to keep the row indices. Instead, it keeps only the row offsets, as shown by Figure 1-a. In this format, non-zero elements of row i and corresponding column indices are located respectively in the data and column index vectors at index $r : \mathit{RowOffset}[i] \leq r < \mathit{RowOffst}[i+1]$. This way, we save both memory space and load, because for all elements in each row, we keep only the start and end offset of that row.

2.3 ELLPACK (ELL) Format

ELLPACK (ELL) [17] is another format that works at the granularity of thread per row but with the expense of redundant memory usage, data transfer and computation power. In this format, first the non-zero elements of the matrix in each row are compressed, then each row is padded (with extra "0" elements) such that all the rows have the same size as the row with largest number of non-zero elements. Along with the padded data matrix, there is a column matrix that holds the corresponding column index of non-zero elements. Figure 1-b shows the ELL format of our example matrix. While ELL format achieves high performance on dense matrices or on matrices with nearly equal numbers of non-zero elements per row, it suffers from redundant memory usage due to padded rows, redundant computation and data transfer for other matrices (i.e. with variant number of non-zero elements per row).

2.4 HYB Format

Hybrid COO-ELL [4] is a hybrid of COO and ELL in which the ELL part is a complete $row \times k$ matrix. If a row has less than k non-zero elements, it is padded with 0s. And if a row has more than k non-zeros, then the remaining elements are packed into a COO format. k is the maximum value, such that, there is at least $R = max(4096, \frac{M}{3})$ rows with k or more non-zero elements (M is the total number of non-zero elements of the matrix). For a given $k = 2$, Figure 1-c shows the corresponding HYB format of the example matrix A.

2.5 Jagged Diagonal Storage (JDS) Format

JDS [18, 19] is designed to eliminate the redundant memory usage, data transfer and computation of the ELL format. In JDS, rows are rearranged in decreasing order of number of non-zero elements. In this format, Matrix A is represented by the following components (Figure 1-d): *data* that contains non-zero values of the matrix A, ordered by number of non-zeros in each row; *perm*, a vector whose elements indicate the original place of each permuted row; *column begin* that contains the beginning index of each column in the new format; *column index* that holds the column index of non-zero elements, corresponding to vector data; and *non-zeros* that holds the number of non-zeros of each row. This format

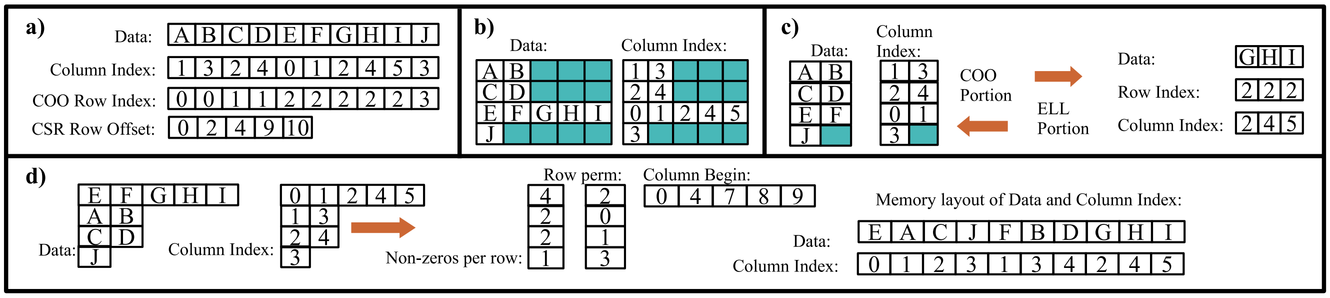
a)
Data: A B C D E F G H I J
Column Index: 1 3 2 4 0 1 2 4 5 3
COO Row Index: 0 0 1 1 2 2 2 2 2 3
CSR Row Offset: 0 2 4 9 10
b)
Data:
Column Index:
c)
Data:
Column Index:
COO Portion
ELL Portion
Data: G H I
Row Index: 2 2 2
Column Index: 2 4 5
d)
Data:
Column Index:
Non-zeros per row:
Row perm:
Column Begin: 0 4 7 8 9
Memory layout of Data and Column Index:
Data: E A C J F B D G H I
Column Index: 0 1 2 3 1 3 4 2 4 5

Figure 1: Sparse matrix A transformed into different formats: a) COO/CSR, b) ELL, c) HYB, and d) JDS.

works well when the number of non-zero elements of consecutive rows (in the ordered version) that are packed into a warp/block are the same. However, it suffers from thread divergence when rows with different numbers of non-zero elements are assigned to the same warp/block.

Note that in ELL and JDS (and the ELL portion of HYB), data is stored in column-major order so that SpMV has coalesced accesses. This layout is shown explicitly in Figure 1-d. Moreover, the cost of transforming Matrix A into a customized format is assumed to be amortized and thus can be ignored because is paid once, before launching a large-scale iterative algorithm such as conjugate gradient solver [11].

3. THE BRC FORMAT

ELL format only works well for matrices with equal numbers of non-zeros per row; otherwise, it suffers from redundant computation (for padded rows). Moreover, for matrices with large numbers of non-zeros (e.g. *WEB*, *LP* in Table 1), size of the GPU device memory becomes a serious issue for ELL [22, 4]. Also, padding overhead becomes worse when performing double-precision computation. COO performs a minimum number of computations, but because of the fine-grained element-to-thread mapping it must perform (usually expensive) atomic (write) operations. The contention due to the increasing number of atomic writes (as the number of non-zeros per row increases) causes COO to fail for large matrices. Although segmented reduction [20] can decrease this overhead in very large matrices, it still suffers high computation due to reduction process. JDS assigns one row to each thread and thus, its performance is dependent on the matrix structure and distribution of elements. This is because branch divergence will not be improved when different (sorted) rows, mapped to threads in the same warp, work on different numbers of elements. Moreover, JDS doesn't promise load-balanced execution when different warps work on rows with different numbers of non-zeros.

To address these problems, we propose the BRC (blocked row-column) format with a novel two-dimensional blocking (i.e. grouping) mechanism in which row-blocking of the matrix A is done to reduce thread divergence and column-blocking to improve load-balance.

3.1 Blocked Row

We first propose row-blocking in which the row permutation (based on the number of non-zeros in each row) of JDS is combined with ELL padding mechanism. However, unlike ELL, we do not pad the entire matrix. Instead, after permuting the rows, we group them into blocks of consecutive rows. Then, in each block we adaptively pad rows based on the number of non-zeros in the first row of that block. Note that, since rows are sorted, first row of each block will have the maximum number of non zeros in that block. This way, we decrease the amount of memory usage, redundant computation and data transfer. In addition, by assigning one warp to each block, we remove in-warp thread divergence.

Figure 2-a shows the example matrix A in blocked-row format. In this figure, the array labeled "Data" contains the actual non-zeros and padded elements; "Column Index" contains the corresponding column indices; "Non-zeros per block" holds the number of non-zeros in each row of the block; "Block begin" holds the beginning address of each block; and "Row perm" keeps the original row of the permuted rows. $B1$, the number of rows in a block that maps to a warp (more description later in this section), is the only parameter used for row-blocking. When SpMV is executed for a matrix in blocked-row format, each block is processed by one warp and each warp may process multiple blocks in order to hide data transfer latency and to have load-balanced execution for all SMs. To avoid extra accesses to vector x in global memory, we use the same column index of the previous rows in the block for the zero-padded rows (those elements will be cached when executing global loads from preceding threads in the same thread block).

3.2 Blocked Row-Column (BRC)

In sparse matrices with unstructured rows, *Blocked-row* format produces blocks in which first row may have a much larger number of non-zeros than the last row. In addition to redundant computation, data transfer and memory usage, this also causes warp-level load imbalance that may not be addressed even by assigning multiple blocks to each warp. Furthermore, blocks that pack longer rows will become a performance bottleneck. To solve this problem, we propose to add another reformatting step after row-blocking in which the matrix A is blocked again along the column dimension (V-Blocks in Figure 2-b). This format is called *blocked row-column* (BRC).

In BRC, we group neighboring rows into smaller blocks such that the maximum number of non-zeros is controlled by a parameter (i.e. $B2$). To do so, we scan the sparse matrix from the first row (rows are already ordered) and create each block by grouping $B1$ rows and packing maximum of $B2$ non-zeros of each row. By scanning the matrix row-wise and then column-wise (*H-Block* and *V-Block* in Figure 2, respectively), each block will have a size of $B1 \times T$ where:

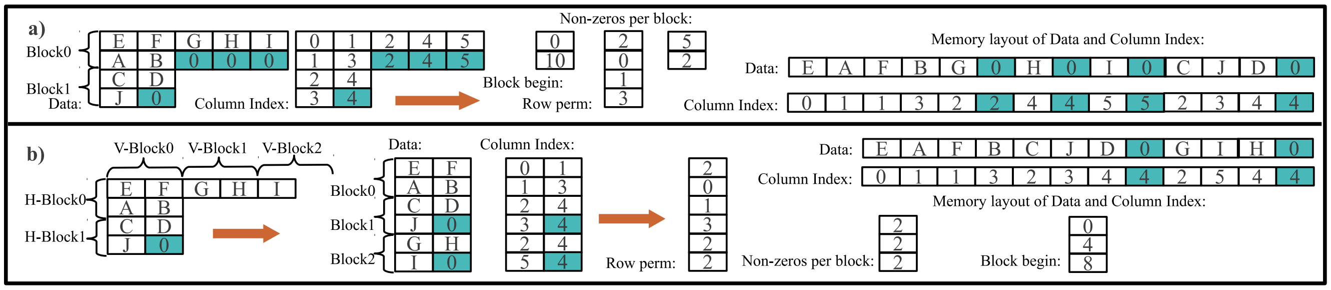

Figure 2: The proposed BRC format: a) blocked-row, b) blocked row-column.

$$T = min(B2, max_{i \in block}(\text{non-zeros of row i})) \qquad (2)$$

Such a blocking scheme is completely adaptive and accounts for the matrix characteristics (e.g. sparsity) as well.

Figure 2-b shows the example matrix A transformed into BRC format. After constructing each block, we decrease the number of non-zeros of the rows by T (if one row has been packed n times into the block, we decrease its non-zeros by $n \times T$). When building a block, if there are not enough rows with non-zeros to be filled up, we use rows from the next vertical block. For example, when scanning rows in *V-Block1*, we only have one row to use for building *Block2*. Therefore, the remaining rows of *Block2* are filled by rows from *V-Block2*. And if the last block is not full, we pad it with zeros. As in row-blocking, we use the column index of the previous row in the same block for the padded elements. Moreover, as in all previous formats, we put the data and column index into a column-based one-dimensional array. Figure 2-b also shows the final memory layout of BRC. Unlike blocked-row where each row is processed by only one thread, BRC may require a row to be processed by multiple threads. Therefore, we use atomic write (*atomicAdd*()) to update vector y.

```
__global__ void brc_spmv (
  float* y, float* data, float* x, int* nnz_per_block,
  int* column_index, int* block_offset, int* row_perm,
  int num_blocks, int rep, int B1) {

  int tid = blockIdx.x*blockDim.x+threadIdx.x;
  int cb = tid / B1; // current block
  int b_row = tid % B1;
  for (i=0; i< rep; i++){
    float tmp = 0.0;
    for (j=0; j < nnz_per_block[cb]; j++){
      int index = block_offset[cb]+j*B1+b_row;
      tmp += data[index]*x[column_index[index]];
    }
    atomicAdd(row_perm[tid], tmp);
    tid += gridDim.x * blockDim.x;
    cb += gridDim.x * blockDim.x / B1;
  }
}
```

Figure 3: An example kernel in BRC format

Algorithm 1 demonstrates transformation of a matrix from CSR to BRC. We also present an example CUDA kernel in BRC format in Figure 3 in which *rep* corresponds to the coarsening factor (described later in this section). In this kernel, each iteration of the i (i.e. coarsening) loop takes care of a padded block of size $B1 \times T$. In the j loop, each thread takes care of one row in the block and performs T corresponding computations where $T = nnz\_per\_block[cb]$.

3.3 Model Parameters

The first parameter needs to be set for BRC is $B1$ (block height). Since each thread computes one row in a block, $B1$ is important for the following two reasons.

1. Consecutive threads will access consecutive elements in memory (respect memory coalescing)
2. Having row segments with the same number of non-zeros (including padded zeros) means threads in the same warp will perform same amount of computation, and thus control divergence will be removed.

For example, in order to have coalesced DRAM accesses, and for a given burst size of 64 bytes, we set $B1$ to be multiple of 16, for single-precision computation. However, if there is only one warp per thread block, $B1$ has to be chosen as the warp size (e.g. 32 in our experiments). For all the experiments in this paper, we have set $B1$ to be 32. The reason is, for cases with uneven distribution of non-zeros of the rows in one block (e.g. difference between max and min is high), larger $B1$ introduces redundant memory accesses and also computations. Therefore, when standard deviation in the number of non-zeros is small, $B1$ can be chosen to be as big as the thread block size.

The second parameter to be set is $B2$ (block width). If the matrix is row-wise sparse (i.e. averages a small number of non-zero elements per row), it is better to have a smaller block along columns to avoid redundant computation and load imbalance. When the matrix is denser, $B2$ can be bigger in order to avoid the extra cost of atomic data writes. This sizing depends on the matrix sparsity and especially to the mean number of non-zeros per row, standard deviation of non-zeros and size of the matrix (i.e. number of rows and columns). To respect this characteristic, we set $B2$ using the following heuristic:

$$B2 = min(C \times round(\mu + \sigma), MaxNZ) \qquad (3)$$

in which C is a constant, μ is the mean number of non-zeros per row, σ is the standard deviation and $MaxNZ$ is the maximum non-zeros per row. Depending on σ, we choose $B2$ to be close to the μ value. Note that having a small σ means that we have a matrix in which most of the rows have the same number of non-zeros and that number is close to μ. If σ is large, then we choose $B2$ to be larger as well. However, note that we do not necessarily pad all the blocks with $B2$ elements along the row dimension. Our method is

Algorithm 1: BRC Format Transformation

```
input  : CSR matrix: $data$, $column\_index$, $row\_offset$
output: BRC matrix: $data$, $column\_index$, $row\_perm$, $nnz\_per\_block$, $block\_offset$, $num\_blocks$
begin
    Extract $\mu$, $\sigma$ and vector $nnz\_per\_row$ from $row\_offset$;
    Let $Qrows$ be a queue contains list of rows (sorted based on $nnz\_per\_row$) with its permutation;
    $num\_block = 0$; $block\_offset[0] = 0$;
    $B1 = 32$; $rep = 1$; $B2 =$ "Equation 3";
    if $\sigma > \mu$ then
        $rep = MaxOfCoalescing$
    while $Qrows \neq \emptyset$ do
        Extract a block of size $B1$ ($current\_block$) as follows and add it to the list of blocks;
        begin
            $current\_block = \emptyset$;
            for $i = 1 : B1$ do
                $r\_ptr = Qrows.front$;
                $r\_nnz = r\_ptr.NNZ$;
                add row $r\_ptr$ with $min(r\_nnz, B2)$ to $current\_block$;
                add original $row\_index$ to $row\_perm$;
                delete $Qrows.front$;
                if $r\_nnz - B2 > 0$ then
                    $r\_ptr.NNZ = r\_ptr.NNZ - B2$;
                    insert $r\_ptr$ back to $Qrows$;
                if $Qrows = \emptyset$ then
                    break;
            store $current\_block \rightarrow data$ into $BRC \rightarrow data$ (in column order);
            set $BRC \rightarrow column\_index$ accordingly;
            set $nnz\_per\_block[num\_block]$ as the max non_zero in rows of $current\_block$;
            pad the smaller rows in $current\_block$ with zeros and column as previous row accordingly;
            add dummy rows if necessary (i.e. loop exited with $i < B1$);
            $num\_block++$;
            $block\_offset[num\_block] = block\_offset[num\_block - 1] + B1 * nnz\_per\_block[num\_block - 1]$;
```

adaptive and each block chooses its block width based on the value T defined by the Equation 2. That means blocks at the top of the row-order may have a much bigger size than blocks at the bottom due to bigger block width. Meanwhile, Equation 3 respects the load balance among the threads, because $\mu + \sigma$ presents a measure of where the density of the number of non-zeros in each row is focused. Therefore, when we transform the matrix into BRC, in each vertical-block scan of the matrix, threads in each block and also neighboring blocks get almost equal amount of work. If a block has a lower load compared to others, it means that block has all the remaining non-zero elements of some rows which have been scanned completely. In the result section, we show that Equation 3 is experimentally accepted.

Such a $B2$ value is theoretically acceptable unless it is so large that performing SpMV on each row becomes a bottleneck (i.e. when there are rows with large number of non-zeros close to $B2$). In those cases, our solution is to break down the $B2$ further. To do so, we apply the following refinement to our $B2$ selection: we choose its value as the minimum among what has been calculated in Equation 3 and a predefined maximum $B2$ size. Such a maximum can be chosen depending on the device characteristic, computation type and latency of the computation and data transfer. In our case, we have used a maximum of 200 by a simple search. We also apply a final refinement to our parameter selection to deal with the load imbalance issue. In particular, when $\mu < \sigma$, there are big blocks at the top rows and small blocks around the bottom rows, resulting in load imbalance on the warps. To handle this case, we apply a level of thread coarsening and assign each thread more than one row to work on.

4. EVALUATION METHODOLOGY

In order to evaluate the effectiveness of the BRC format, we selected a diverse set of sparse matrices that have been used in previous studies [22, 4]. These matrices are in Matrix Market Coordinate Format [15]. Table 1 shows the $B2$ value chosen for each matrix along with other characteristics of each matrix: min, max, mean (μ) and standard deviation (σ) of the number of non-zeros per row. Our auto-tuning model sets $B2$ such that it falls into the range of min and max, is close to μ and covers a broad range of non-zeros. Since our method is adaptive at the warp level, the padding cost is negligible. As Table 1 shows, if σ is small, then $B2$ is equal to μ. If μ is too large, $B2$ is chosen to be a number small enough such that each thread processes a limited number of elements and does not become a computation bottleneck. The thread block size was set to 128, so that there are enough warps to mask the latency of data transfer in each block.

We report performance in terms of computation rate (as number of floating point operations per second in GFLOPs). Each SpMV experiment was repeated 50 times and the average (arithmetic mean) is reported. We exclude the time spent transferring data between host and device and other one-time SpMV data transformation overhead that is performed on the CPU side (for compatibility reasons). However, we later also report performance in a real use scenario with PETSc, including all data conversion and transfer overheads, where it is shown that these overheads are negligible.

The experiments are run on two different generations of NVIDIA GPUs, $GTX580$ (Fermi) and $GTXTitan$ (Kepler), each hosted by an Intel Core i7 CPU. Details of the GPUs peak are listed in the Table 2. We used NVIDIA compiler ($nvcc$) version 5.5, with all the general optimizationsenabled

Matrix	Abbrv.	Rows	Columns	Total NZ	Mean NZ (μ)	SD NZ (σ)	Min NZ	Max NZ	B2	HYB-k
FEM/Cantilever	CAN	62,451	62,451	4,007,383	64	14	1	78	78	75
FEM/Spheres	SPH	83,334	83,334	6,010,480	72	19	1	81	91	81
FEM/Accelerator	ACC	121,192	121,192	2,624,331	22	14	8	81	36	23
Dense	DEN	2,000	2,000	4,000,000	2000	0	2000	2000	200	0
Economics	ECO	206,500	206,500	1,273,389	6	4	1	44	10	7
Epidemiology	EPI	525,825	525,825	2,100,225	4	0	2	4	4	4
Protein	PRO	36,417	36,417	4,344,765	119	32	18	204	151	138
Wind Tunnel	WIN	217,918	217,918	11,634,424	53	5	2	180	58	54
QCD	QCD	49,152	49,152	1,916,928	39	0	39	39	39	39
LP	LP	4,284	1,092,610	11,279,748	2633	4209	1	56181	200	23
FEM/Harbor	HAR	46,835	46,835	2,374,001	51	28	4	145	79	55
Circuit	CIR	170,998	170,998	958,936	6	4	1	353	10	5
FEM/Ship	SHI	140,874	140,874	7,813,404	55	11	24	102	66	54
Webbase	WEB	1,000,005	1,000,005	3,105,536	3	25	1	4700	28	2

Table 1: Set of matrices used in our experiments (NZ: non-zero, SD: standard deviation (σ), B2: width of vertical blocks, HYB-k: number of columns in ELL portion of the HYB format).

GPU Model	GTX 580	GTX Titan
Architecture	Fermi (GF110)	Kepler (K20X)
Compute capability	2.0	3.5
Multiprocessors, cores per MP	16 , 32	14 , 192
Warp size	32	32
Max threads per block/MP	1024/1536	1024/2048
Shared memory (KB) per block	48	48
L2 cache (KB)	768	1536
Total global memory (MB)	1535	6143
Peak off-chip BW (GB/s)	192	288
Peak SP TFLOPS (FMA)	1.5	4.5

Table 2: GPUs hosted the experiments.

BRC vs.	Fermi		Kepler	
	SP	DP	SP	DP
CSR	1.71 ×	1.19 ×	1.96 ×	1.64 ×
HYB	1.67 ×	1.45 ×	2.68 ×	2.47 ×
JDS	1.64 ×	1.30 ×	4.91 ×	3.54 ×
RG12 [16]	1.19 ×	1.08 ×	2.30 ×	1.69 ×

Table 3: Average BRC speedup.

using *-O3*. The input vector x was placed in texture memory to improve the accesses when locally cached in SM. As the baseline, we use the best performance among the HYB and CSR formats (from cuSPARSE [8] and CUSP [7] libraries), JDS format [18], and also improved CSR recently developed by Reguly and Giles [16]. We also compared BRC against a state-of-the-art format recently developed for the Intel Xeon Phi many-core system [13].

5. EXPERIMENTAL RESULTS

In the first experiment, we compared BRC with CSR, HYB, JDS and the enhanced CSR approach recently developed by Reguly and Giles [16]. For BRC, $B1$ was set to 32, and $B2$ set according to Equation 3. Figure 4 shows the performance obtained (in GFLOPs) with the different SpMV formats, for single and double precision computation, on the Kepler GPU. BRC outperforms all other SpMV formats, except for one case (matrix EPI) where HYB is slightly better. *EPI* is a very sparse matrix with rows that mostly have 4 non-zero elements, with a few rows with some 2 or 3 non-zeros. With the HYB format, this matrix with $\mu = 4$ and $\sigma \simeq 0$ is represented entirely using the ELL format (no COO part), which is the best match for such a sparse matrix. This avoids the penalty of reading row permutations and uncoalesced memory write that BRC requires due to the row reordering. Figure 5 shows performance on the Fermi GPU. Here again the BRC format is generally superior, except for a few cases (ECO, EPI, CIR, and WEB). These are the cases where the average number of non-zero elements per row is very small ($max = 6$) and thus there is not much to do due to the low bandwidth utilization of the GPU global memory. In the case of LP matrix, we are limited by the overhead of the atomic add operation. This can be improved by reduction. Later in this section, we report on additional experiments that shed light on the performance with these matrices. Table 3 summarizes the average speedup of BRC over other SpMV formats, for single (SP) and double (DP) precision computation on the two GPUS.

To test the effectiveness of the model for selecting the $B2$ parameter, we first compare performance of BRC by choosing different values for $B2$, either from the range of $[min, max]$ number of non-zeros in each matrix, or from Equation 3. Figure 6 shows this result for double precision on the Kepler GPU. In this case, there is no thread coarsening. As the figure shows, increasing $B2$ improves the performance until it reaches the range of "$\mu + \sigma$". After that point, higher values of $B2$ either have negligible impact (i.e. small fluctuations) or hurt the performance. Also, except for three cases (CIR, ECO, LP), the value of $B2$ chosen by BRC gives the best performance. In the case of LP, BRC thread coarsening improves performance when $B2$ is chosen by Equation 3, while it does not improve performance when $B2 = \{1k, 2k, 4k\}$. And in the case of CIR and ECO, the difference is very small.

We also performed an exhaustive search in a bounded domain of model parameters of size 24. The model had an efficiency of 96% compared to the result of this bounded search. This search is similar to the optimal method proposed by Reguly and Giles [16]. The results suggest that using such a search would help find the optimal model parameters in a few runs. Such a parameter search can be used when the number of iterations of SpMV is large in comparison with the search size. Figure 7 compares BRC to the optimal performance achieved by the enhanced CSR [16]. It can be observed that BRC outperforms [16] on all the test matrices. This was also true for other matrices tested by Reguly and Giles [16], which are not included in Figure 7.

Furthermore, we compared BRC against the EBS scheme of Liu et al. [13] (state-of-the-art many-core CPU solution to SpMV). Figure 8 shows performance of BRC and KNC-adaptive (referred in the figure as *LSCD13*) on the set of

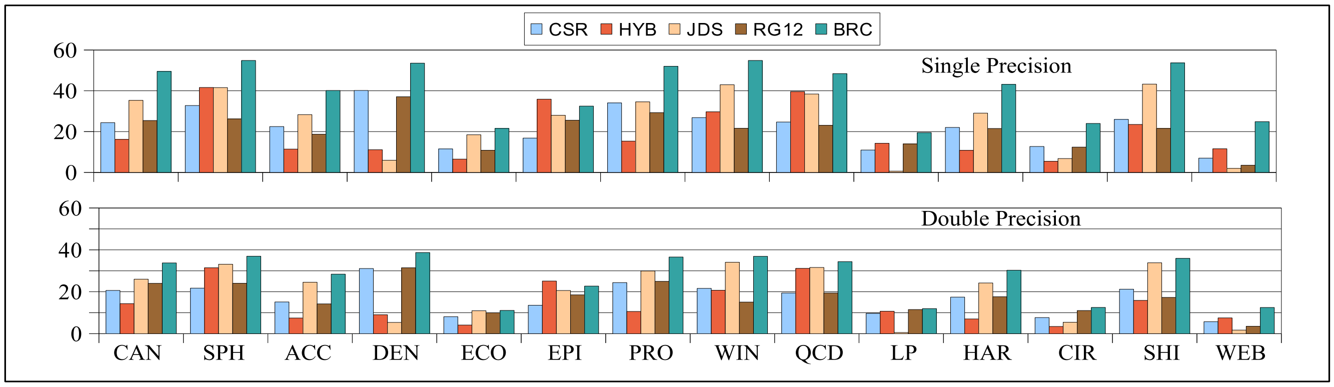

Figure 4: BRC performance (GFLOPs) on Kepler vs. other formats (*RG12* refers to Reguly and Giles [16]).

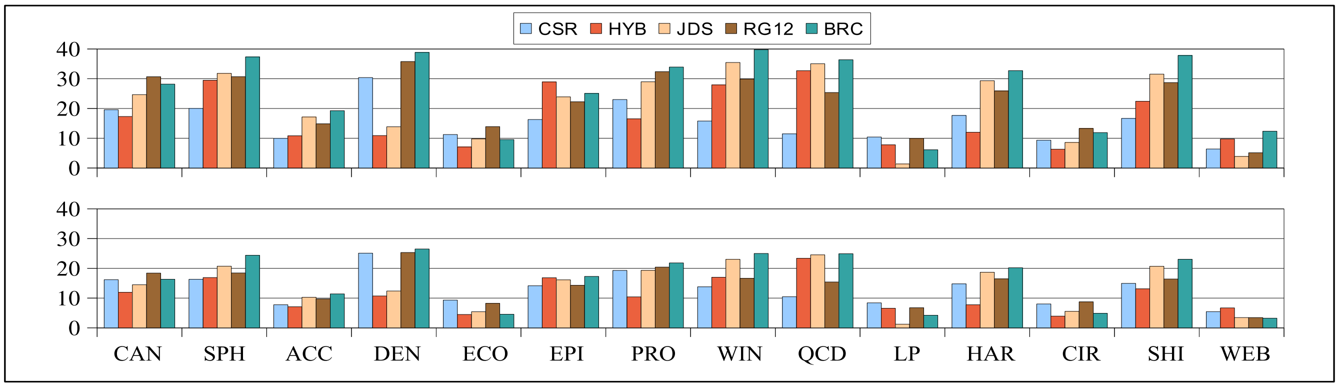

Figure 5: BRC performance (GFLOPs) on Fermi vs. other formats (*RG12* refers to Reguly and Giles [16]).

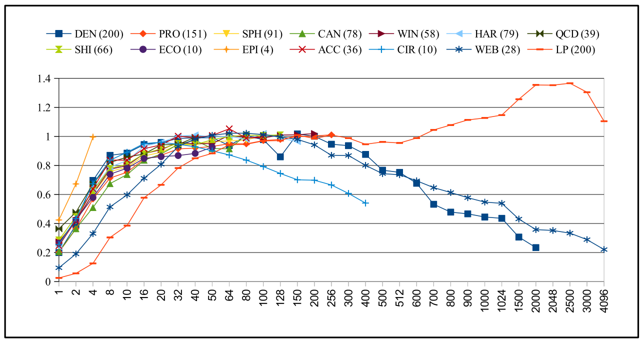

Figure 6: Change in BRC performance for different values of B2 (X-axis), compared to the baseline BRC (i.e. B2 from Equation 3, shown in the legend).

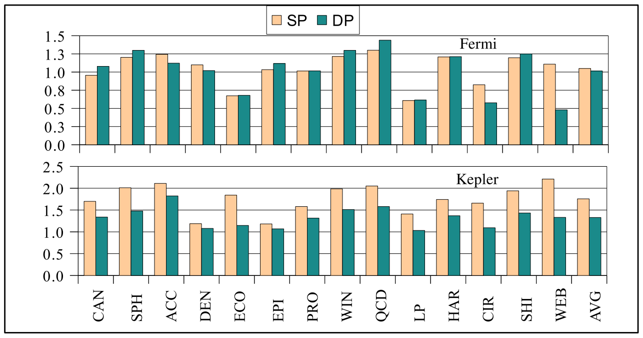

Figure 7: Auto-tuned BRC speedup vs RG12 [16].

matrices that have been tested in [13]. *LSCD13* data in this figure are extracted from *Figure 9* in [13]. As shown by the figure, BRC is 1.27× faster than EBS. However, EBS performs better on matrix *circuits5M*, a large sparse matrix with average of 11 non-zeros in $5M$ rows that are mostly located around the diagonal. In this case, EBS leverages locality of vector x in each column partition. EBS is also a bit faster on matrices *rail4284* and *spal-004*, which are matrices with much smaller number of rows in comparison with columns, and large μ and σ. We next describe how a hybrid of ELL and BRC can perform better with such matrices.

In another experiment, we studied replacing either of the COO or ELL parts of the HYB format by BRC. The last column in Table 1 shows the column size of the ELL part, chosen by HYB. Furthermore, Table 4 provides more information on how HYB partitions the matrix into COO and ELL parts. In this table, the first column shows the fraction of non-zeros that fall into the COO portion (CNZ). The second column shows the fraction of the total time taken by COO computation (CTR). The third and fourth columns show the time of COO and ELL in ms, respectively (CTA and ETA). As Table 4 shows, in the case of matrices which

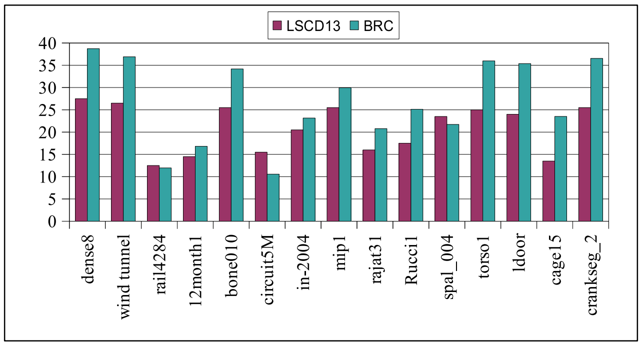

Figure 8: GFLOPs for BRC vs Liu et al. [13].

Matrix	CNZ(%)	CTR(%)	CTA(ms)	ETA(ms)
CAN	0.323	38.189	0.181	0.293
SPH	0	0	0	0.413
ACC	17.248	52.727	0.254	0.228
DEN	100	100	0.672	0
ECO	18.879	69.635	0.268	0.117
EPI	0	0	0	0.149
PRO	4.026	41.016	0.220	0.316
WIN	0.561	23.602	0.2	0.647
QCD	0	0	0	0.121
LP	99.147	98.640	2.782	0.038
HAR	18.673	61.616	0.247	0.154
CIR	21.775	74.862	0.238	0.080
SHI	6.850	38.363	0.270	0.433
WEB	35.788	75.702	0.508	0.163

Table 4: Timing analysis for the CUSP HYB

are less regular, COO takes care of a high percentage of non-zeros and thus takes most of the total time. However, in cases where ELL can finish up computation for most of the non-zeros, the COO part plays a significant role as well. For example, in *Cantilever* matrix, COO has only 0.3% of the non-zeros but takes 38% of the total time. In *Wind Tunnel*, COO has 0.56% of the non-zeros while it takes 23.6% of the total time. Therefore, it is very important to use formats that can substitute the COO portion.

We individually substituted both ELL and COO portion of the HYB format with BRC and then compared the performance with COO and ELL. Figure 9 shows the result of applying BRC on the COO and ELL portions separately. It also shows the performance of the original COO and ELL in HYB. For the COO portion, it may be seen that BRC always outperforms COO. This implies that BRC can always be substituted for COO in the HYB format for improved performance. However, this observation does not hold for the ELL portion. In the matrices *ECO, EPI, LP*, and *WEB*, ELL works better than BRC. These are matrices that either have zero standard deviation of non-zeros like *EPI*, or have very high standard deviation (in comparison with mean) of non-zeros and a big portion of them can be separated and put into ELL format. This observation suggests that in the cases where HYB outperforms BRC, a hybrid of BRC and ELL is likely to perform even better. This could be done via empirical auto-tuning for cases where the same matrix is used multiple times.

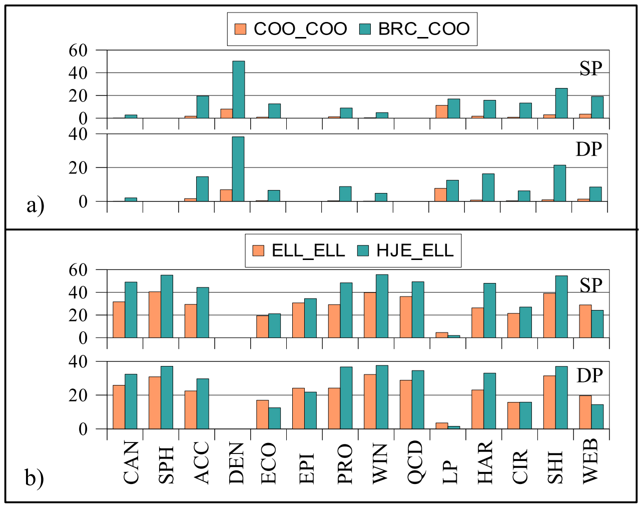

Figure 9: Performance in GFLOPs for a) COO portion of HYB solved using COO (COO_COO) or BRC (BRC_COO) and b) ELL portion of HYB solved using ELL (ELL_ELL) or BRC (BRC_ELL)

6. INTEGRATION OF BRC IN PETSC

In order to facilitate the practical use of the developments reported in this paper, we have incorporated it into the widely used PETSc framework [1, 2]. PETSc has an object-oriented architecture and supports many implementations of the abstract Matrix type. One of these implementations is called the AIJ matrix type and is PETSc's version of the CSR matrix format for sparse matrices. We have implemented the BRC matrix format in PETSc by deriving it from the AIJ type. Our BRC type inherits all functionality implemented in AIJ except the creation and sparse matrix vector multiply functions. The creation routine creates the BRC format version of the sparse matrix and stores the data on the GPU while the sparse matrix vector multiply function uses the BRC format of the matrix. When, for example, an ILU preconditioner is needed for a BRC matrix, the L and U factors are created just as they are for the AIJ superclass and are stored in the same format. The AIJ code is used for the forward and backward substitution steps when using the ILU preconditioner in an iterative solver. However, sparse matrix vector multiplications are performed on the GPU using the BRC format.

To test this implementation we ran one of the PETSc example simulations which must solve systems of non-linear equations. For each iteration of this non-linear solve a system of linear equations must be solved using an iterative method with the user's choice of preconditioner. We ran the simulation using the BRC matrix type and a couple of different preconditioners. During the simulation we measured the elapsed time spent in the routines MatMult, which performs the sparse matrix vector multiplication, and the routine PCApply, which applies the preconditioner. These are called during each step of the iterative solver which was chosen to be GMRES. The time spent in these routines is tabulated in Table 5. We compare the results of running the simulation with three different matrix types – AIJ, CUSP and BRC. The CUSP matrix type is also derived from the AIJ type but stores the matrix data on the GPU in the CSR

Platform	Precon	Format	Iter 1	Iter 2	Conv 1	Conv 2	SpMV1	SpMV 2	PC 1	PC 2	Total
Fermi	ILU(0)	AIJ	657	770	—	—	3.048	3.571	3.756	4.400	26.864
		CUSP	657	770	—	—	1.068	1.248	5.383	6.290	17.345
		BRC	657	770	0.044	0.044	0.412	0.483	4.376	5.117	13.891
	Polynomial	AIJ	299	345	—	—	1.435	1.602	37.680	43.461	91.269
		CUSP	299	345	—	—	0.500	0.558	10.359	11.957	25.783
		BRC	299	345	0.045	0.043	0.194	0.217	4.454	5.137	11.994
Kepler	ILU(0)	AIJ	657	770	—	—	3.113	3.649	3.667	4.298	27.600
		CUSP	657	770	—	—	1.218	1.427	4.453	5.201	15.143
		BRC	657	770	0.040	0.039	0.248	0.290	4.201	4.914	12.683
	Polynomial	AIJ	299	345	—	—	1.467	1.638	38.534	44.446	94.639
		CUSP	299	345	—	—	0.494	0.551	10.078	11.634	24.831
		BRC	299	345	0.040	0.039	0.110	0.123	2.657	3.065	7.540

Table 5: Performance (sec) of PETSc ex19 on two GPU platforms

format and a GPU implementation of sparse-matrix multiplication using NVIDIA's CUSP library. The CUSP matrix type also uses the AIJ format for its ILU preconditioner but overrides the sparse matrix vector multiplication function. For both the CUSP and BRC types, the time spent in SpMV includes the time to transfer the vector values from main memory to GPU memory before the multiplication and to transfer the resulting vector back to main memory. The AIJ implementation of SpMV is done on the CPU.

Table 5 shows the time for running this PETSc example with 2 different preconditioners for 3 matrix formats on two different machines, one with a Fermi GPU and one with a Kepler GPU. The columns labeled *Iter 1* and *Iter 2* show the number of GMRES iterations required for each of the two linear solves. *Conv 1* and *Conv 2* show the time to convert the Jacobian matrix to BRC format while the *SpMV* and *PC* columns show the time in sparse matrix vector multiplication and application of the preconditioner, respectively. As can be seen, for ILU the time to apply the PC is roughly the same for all three matrix types but for the polynomial preconditioner the two GPU formats are better, with BRC being superior. This is because the application of the polynomial preconditioner is based on matrix vector multiplication. The time reported in the *SpMV* columns shows the benefit of the BRC format. The time spent in conversion to the BRC format is more than made up for by the speedup of SpMV.

7. RELATED WORK

There is an extensive amount of work in the literature on customizing sparse matrix formats and optimizing SpMV on different platforms. Vuduc [21] has studied SpMV on single-core CPUs and presented an automated system for generating efficient implementations of SpMV on these platforms. Williams et al. [22] moves toward multi-core platforms with the implementation of parallel SpMV kernels. With the emergence of heterogeneous computing and many-core systems, SpMV kernels have been studied and optimized for GPUs as well. Bell and Garland have implemented sparse matrix formats in CUDA [4, 7, 8]. They have also proposed HYB (hybrid of ELL and COO), which is generally the fastest format for a broad class of unstructured matrices.

Baskaran and Bordawekar [3] have optimized Compressed Sparse Row (CSR) so that it performs about the same (modestly faster with more memory cost) as CSR-Vector [4] (in which each warp takes care of a row with segmented reduction). However in general, it does not reach the same performance as CUSP-HYB [4] on unstructured matrices.

Choi et al. [5] also combine the idea of blocking, CSR and ELLPACK (ELL), and present BCSR and BELLPACK. These formats of SpMV outperform HYB in matrices with dense block substructure but on average their auto-tuned version is still behind HYB for general unstructured and sparse matrices. Although we have not compared our method with theirs on the same machine with the same set of matrices, our average speed up over HYB is higher than the max they have reported with exhaustive search on their model.

Yang et al. [23] present a fast SpMV for matrices that present large graphs with power-low characteristics. They combine ideas from Transposed Jagged Diagonal Storage (TJDS) [10] with COO and tiling. They first order columns of the matrix based on non-zero elements in each column and then they tile the columns and work on each tile separately. They do a row ordering on each tile separately and assign the same amount of work to each warp in this order. The last step is a COO-like process that has the overhead of atomic memory write/reduction. It differs from our algorithm in that our format does not need column-based ordering and our method guarantees blocks to have same number of non-zero elements (rarely with the help of padding). Furthermore, our method is general and not limited to very big matrices with power-law characteristics.

Liu et al. [13] propose ELLPACK Sparse Block (ESB) for the Intel Xeon Phi Co-processor, code-named Knights Corner (KNC), that partitions the matrix coarsely by rows and columns into large sparse blocks and applies ELL on those blocks. They first partition the matrix column-wise, sort each column partition separately and block it row-wise. Then each block is saved in ELL format. Their result shows that on average ESB outperforms cuSPARSE on NVIDIA Tesla K20X. However, it still suffers from redundant computation, overhead of separate sorting per column partition, overhead of memory space and load of row permutation per column partition. Furthermore, on the same device, BRC shows superior performance in comparison with ESB. Figure 8 shows performance of BRC on the same set of matrices has been used in [13].

Reguly and Giles [16] present an auto-tuning algorithm on top of CSR format in which they choose the size of a thread group that works on a given row depending on the available resources on the device and matrix specification. This method also selects the number of rows for a thread to work on. It improves on CSR performance but all rows are treated similarly, and thus this approach lacks a complete load balance on threads and warps for matrices with diverse sparsity of rows. We have compared BRC against [16] and our result shows that BRC outperforms it.

8. CONCLUSION AND FUTURE WORK

In this paper, we have presented BRC, a 2D row-column blocked sparse matrix format for GPUs. BRC outperforms NVIDIA CUSP and cuSPARSE libraries, JDS and other formats. The key feature of the BRC format is its adaptivity to the matrix characteristic. This format performs fewer redundant memory accesses and computations than the ELL format, while it eliminates warp-level thread divergence. Our auto-tuning parameter selection sets the blocking parameters so that we have a balanced work distribution among all the warps.

We also studied the COO and ELL portions of HYB separately and showed that BRC can improve the COO part that is the bottleneck of the HYB format. A hybrid format of ELL and BRC works better than either pure BRC or pure HYB in matrices with $0 < \sigma \leq \frac{2}{3}\mu$. In future work, ELL and BRC can be combined to make a more effective hybrid format. In this format, ELL would represent the dense portion of the matrix, while BRC handles the rest. However, parameter tuning needs to be studied further. For example, in the case of banded matrices with few non-zeros per row, but the same number across all rows (e.g. the QCD and Epidemiology matrices), we can have a value k such that ELL covers all the matrix. In other cases, most of the load goes to the BRC part.

In our auto-tuning model, we fixed the size of vertical blocks based on the matrix characteristics. However, scanning the rows, we know that after building the first vertical block, the remaining rows are mostly among the top-ordered rows. As we scan these blocks (from left to the right), we move toward the top part of the row order. Therefore, one can increase the $B2$ size after scanning each block in order to decrease the number of blocks. Another similar idea for further investigation is to choose $B2$ depending on the row order.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their suggestions that helped improve the quality of this paper. We also thank Reguly and Giles [16] for providing us with the source code of their implementation.

10. REFERENCES

[1] S. Balay, J. Brown, , K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.4, Argonne National Laboratory, 2013.

[2] S. Balay, J. Brown, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc Web page, 2013. http://www.mcs.anl.gov/petsc.

[3] M. M. Baskaran and R. Bordawekar. Optimizing sparse matrix-vector multiplication on gpus. In *Technical report, IBM Research Report RC24704 (W0812-047)*, 2008.

[4] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[5] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP)*, January 2010.

[6] CUDA. A parallel computing platform and programming model invented by nvidia. https://developer.nvidia.com/cuda-home-new.html.

[7] CUSP. The nvidia library of generic parallel algorithms for sparse linear algebra and graph computations on cuda architecture gpus. https://developer.nvidia.com/cusp.

[8] cuSPARSE. The nvidia cuda sparse matrix library. https://developer.nvidia.com/cusparse.

[9] J. Dongarra and M. A. Heroux. Toward a new metric for ranking high performance computing systems. *UTK EECS Tech Report and Sandia National Labs Report SAND2013-4744*, June 2013.

[10] A. Ekambaram and E. Montagne. An alternative compressed storage format for sparse matrices. In *ISCIS*, pages 196–203, 2003.

[11] R. Helfenstein and J. Koko. Parallel preconditioned conjugate gradient algorithm on gpu. *Journal of Computational and Applied Mathematics*, 236(15):3584–3590, 2012.

[12] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.

[13] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey. Efficient sparse matrix-vector multiplication on x86-based many-core processors. *International Conference on Supercomputing*, pages 273–282, 2013.

[14] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *ACM Queue*, 6(2):40–53, 2008.

[15] N. I. of Standards and Technology. The matrix market format.

[16] I. Reguly and M. Giles. Efficient sparse matrix-vector multiplication on cache-based gpus. In *Innovative Parallel Computing (InPar)*, pages 1–12, 2012.

[17] D. M. Y. Roger G. Grimes, David Ronald Kincaid. *ITPACK 2.0: User's Guide*. 1980.

[18] Y. Saad. Krylov subspace methods on supercomputers. *SIAM J. SCI. STAT. COMPUT*, 10:1200–1232, 1989.

[19] Y. Saad. Sparskit: a basic tool kit for sparse matrix computations - version 2. 1994.

[20] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for gpu computing. In *Graphics Hardware*, pages 97–106, 2007.

[21] R. W. Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, University of California, January 2004.

[22] S. Williams, L. Oliker, R. W. Vuduc, J. Shalf, K. A. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178–194, 2009.

[23] X. Yang, S. Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on gpus: implications for graph mining. *Proc. VLDB Endow.*, 4(4):231–242, 2011.