J. Parallel Distrib. Comput. 76 (2015) 3–15

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

A model-driven blocking strategy for load balanced sparse matrix–vector multiplication on GPUs

Arash Ashari *, Naser Sedaghati, John Eisenlohr, P. Sadayappan

Department of Computer Science and Engineering, The Ohio State University, United States

HIGHLIGHTS

- A novel blocking strategy that reduces thread divergence and improves load balance.
- Enhanced performance modeling for selection of a key blocking parameter.
- An efficient auto-tuning technique to optimize performance.
- Comprehensive experimental evaluation and integrating with a real system; PETSc.
- A multi-GPU algorithm for SpMV with experimental evaluation.

ARTICLE INFO

Article history:
Received 28 February 2014
Received in revised form
31 October 2014
Accepted 4 November 2014
Available online 12 November 2014

Keywords:
SpMV
GPU
CUDA

ABSTRACT

Sparse Matrix–Vector multiplication (SpMV) is one of the key operations in linear algebra. Overcoming thread divergence, load imbalance and un-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 two-dimensional blocking mechanism that addresses these challenges effectively. It reduces thread divergence by reordering and blocking 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 approach 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. Furthermore, when partitioning the input matrix, BRC achieves near linear speedup on multiple GPUs.

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 [15] and CUDA [18,9], scientists from a broad range of disciplines have started leveraging the computation throughput of GPUs. Sparse Matrix–Vector multiplication (SpMV) has received much attention since it is a core kernel used in many algorithms, such as iterative methods for solving large sparse linear systems of equations.

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 causes (a) irregular and un-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 [11] and CUSP [10,6,7] are two of the most widely-used CUDA libraries that support different sparse matrix formats; Diagonal (DIA), ELLPACK (ELL), Compressed Sparse Row (CSR), Coordinate (COO), and also hybrid (HYB), which combines ELL and COO. HYB splits the matrix into two parts: (1) a dense part well-suited to ELL and (2) a sparse part, suited to the 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 [5,8,28] have proposed

* Corresponding author.
E-mail address: ashari@cse.ohio-state.edu (A. Ashari).

http://dx.doi.org/10.1016/j.jpdc.2014.11.001
0743-7315/© 2014 Elsevier Inc. All rights reserved.

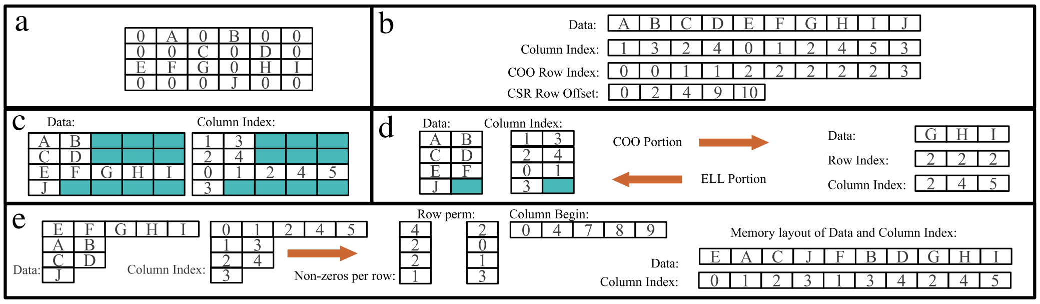

Fig. 1. Sparse matrix formats: (a) original matrix A, equivalent (b) COO, (c) ELL, (d) HYB, (e) JDS.

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 do not 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 tackles the intra-warp thread divergence problem introduced by CSR based formats, and the redundant computation and data transfer problem introduced by the ELL based format. We also address synchronization overhead caused by the reduction/atomic operations in COO. The proposed format, blocked row–column (BRC), leverages the sparsity characteristics of the matrix and transforms it to a hybrid format such that 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 the block along matrix column. It achieves 96% of the performance obtained from exhaustive search in a bounded domain of this parameter.

On an NVIDIA GTX Titan (Kepler GK110) GPU, the CUDA implementation of auto-tuned BRC is up to 4.3× and 3.4× (and average 1.8× and 1.4×) faster than HYB in single and double precision, respectively. BRC also achieves a maximal speedup of 4.1× and 2.5× (and average 2.4× and 1.9×) over CSR (the most commonly used format) for single and double precision, respectively. Compared to the COO format, BRC achieves a maximal speedup of 6.4× and 6.9× (and average 4× and 3.9×) for single and double precision, respectively. Integrating BRC into PETSc [4,3] shows that using BRC as the SpMV kernel reduces the total runtime by 16% and 70% for *ILU(0)* and *Polynomial* preconditioners, respectively, when compared to the PETSc AIJ–CUSP format which uses NVIDIA CUSP [10,6,7] for SpMV.

The rest of the paper is organized as follows: Section 2 reviews the existing SpMV formats and related work. In Section 3, we describe the BRC format. Section 4 describes the evaluation methodology and Section 5 presents the results. Section 6 discusses the integration of BRC with PETSc and Section 7 discusses scalability of BRC using multiple devices. We discuss preprocessing overhead of BRC and formats that perform data layout transformation or auto-tuning in Section 8, and conclude in Section 9.

2. Background and related work

In this paper, we target the SpMV problem in the form $y = y + Ax$, where y and x are two one-dimensional vectors and A is a two-dimensional sparse matrix. By using a special representation of a sparse matrix that only stores the nonzero elements, the number of operations as well as the memory requirements can be significantly reduced from that required with a standard dense matrix representation. A number of formats/optimizations have been proposed for sparse matrix–vector multiplication. Commonly used sparse matrix formats have been reviewed in [6,7,25]. In this section, we provide a brief review of various existing formats.

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

In order to illustrate different formats, Fig. 1(a) shows an example of a sparse matrix with 10 non-zero elements (A to J) distributed unevenly across 4 rows and 6 columns. Thus 14 of the 24 elements of the matrix are known to have a zero value. The COO format for this matrix is shown in Fig. 1(b). When matrix A is in COO format, the SpMV kernel assigns every non-zero element to a separate GPU thread. As a result, an atomic operation is used to collect contributions of different threads (mapped to the same row) and to finalize the reduced results in vector y [7]. One major drawback of the 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. A highly unbalanced distribution has a very negative impact on performance because of unbalanced execution across threads. Even though approaches such as segmented reduction [7,24] decrease this overhead, the performance achievable with the COO format is limited [7].

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 store the row index of every element. Instead, it stores only the row offsets, as shown by Fig. 1(b). 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 : $RowOffset[i] \leq r < RowOffst[i+1]$. This format saves both memory space and load because only the start and end index of each row are stored. With the CSR format, one approach (Scalar-CSR) is to make each thread responsible for calculating the product of a matrix row with the vector x. But this kernel suffers from thread divergence and un-coalesced access to the sparse matrix elements. An alternate approach (Vector-CSR [6,7]) improves performance by having all threads of a warp collectively process a row. But this algorithm wastes resources when rows have far fewer non-zero elements than a warp size. In the most recent and efficient implementation of Vector-CSR in the cuSPARSE [11] and CUSP [10,6,7] libraries, the idea of segmented reduction is employed, where threads of a warp span multiple rows when the average number of non-zeros per row is small.

ELLPACK (ELL) format: ELLPACK (ELL) [21] 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.