Exemplar based approaches on faces

Thesis submitted in partial fulfillment
of the requirements for the degree of

MS by Research
in
Computer Science & Engineering

by

Mallikarjun B R
201307593
kiran.raj@research.iiit.ac.in

Center for Visual Information Technology
International Institute of Information Technology
Hyderabad - 500 032, India
February 2016

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled "Exemplar based approaches for faces" by Mallikarjun B R, has been carried out under my supervision and is not submitted elsewhere for a degree.

Date	Adviser: Dr. Kishore Kothapalli

To My Family

Acknowledgments

Acknowledgments

Abstract

Multiplying two sparse matrices, denoted spmm, is a fundamental operation in linear algebra with several applications. Hence, efficient and scalable implementation of spmm has been a topic of immense research. Recent efforts are aimed at implementations on GPUs, multicore architectures, FPGAs, and such emerging computational platforms. Owing to the highly irregular nature of spmm, it is observed that GPUs and CPUs can offer comparable performance (Lee et al. [39]).

In this paper, we study CPU+GPU heterogeneous algorithms for spmm where the matrices exhibit a scale-free nature. Focusing on such matrices, we propose an algorithm that multiplies two sparse matrices exhibiting scale-free nature on a CPU+GPU heterogeneous platform.

Our experiments on a wide variety of real-world matrices from standard datasets show an average of 25% improvement over the best possible algorithm on a CPU+GPU heterogeneous platform. We show that our approach is both architecture-aware, and workload-aware.

The architectural trend towards heterogeneity has pushed heterogeneous computing to the fore of parallel computing research. Heterogeneous algorithms, often carefully handcrafted, are designed for several important problems from parallel computing such as sorting, graph algorithms, matrix computations, and the like. A majority of these algorithms follow a work partitioning approach where the input is divided into appropriate sized parts so that individual devices can process the right parts of the input. Such a division is done by means of thresholds. However, identifying the right value of the threshold is usually non-trivial and may require extensive empirical search. Such an extensive empirical search may potentially offset any gains accrued out of heterogeneous algorithms.

In this paper, we propose a simple and effective technique to identify the required thresholds in heterogeneous algorithms. Our technique is based on sampling and therefore can adapt to the algorithm used and the input instance. Our technique is generic in its applicability as we will demonstrate in this paper.

We validate our technique on two problems: finding the connected components of a graph and multiplying two scale-free sparse matrices. For these two problems, we show that using our method, we can find the required threshold that is $\pm$ 5% and $\pm$ 7.5% away from the best possible threshold, respectively. Along the way, we design a novel heterogeneous algorithm for sparse matrix multiplication when the matrices are scale-free in nature. This algorithm outperforms the existing best known algorithms for sparse matrix multiplication by 22% on average on a wide variety of matrices drawn from standard datasets.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Relevance of Heterogeneous Computing

1.2 Sparse Matrix Computations

1.2.1 Sparse Matrix Sparse Matrix Multiplication

1.2.2 Sparse Matrix Dense Matrix Multiplication

1.3 Importance of Load Balancing

1.4 Contributions

Chapter 2

Fiducial detection

In this chapter we present different types of matrix multiplications, sparse matrix storage formats, scale-free matrices, heterogeneous platforms, data-sets which were used in this thesis. Also we discuss some previous works that focus on workload balancing in a heterogeneous platform.

2.1 Matrix Multiplication Formulation

Let A, B and C be three matrices with sizes $M \times P$, $P \times N$ and $M \times N$ respectively such that $C = A \times B$. There are four different types of formulations to multiply two matrices. They are Row-Column formulation, Row-Row formulation, Column-Row formulation and Column-Column formulations. All these four formulations are briefly explained with an example.

2.1.1 The Row-Column Formulation

In the Row-Column formulation, to get one element in C, we multiply a row in the A matrix with a column in the B matrix, i.e., $C(i, j) = A(i, :) \times B(:, j)$ for $i = 1, 2, ...M$, and $j = 1, 2, ...N$. This is the standard matrix multiplication approach. For a given i, j, let $I(i, j)$ denote the set of index k such that both the elements $A(i, k)$ and $B(k, j)$ are nonzero. Then, $C(i, j) = \sum_{k \in I(i,j)} A(i, k) \times B(k, j)$. However, to obtain $I(i, j)$, we need to access all the elements in the i^{th} row of A and j^{th} column of B. Therefore, we bring in elements which may not contribute to the output. In the worst case, we would access the entire row i of A and a column j of B whereas $I(i, j) = \Phi$. Hence, this approach is not suited for sparse matrices in general.

2.1.2 The Row-Row Formulation

In the Row-Row formulation, to compute the i^{th} row in C, $C(i, :)$, we multiply each element in $A(i, :)$ with corresponding row in B. We then add all the scaled B rows to get the $C(i, :)$. Thus, $C(i, :) = \sum_{j \in A(i,:)} A(i, :) \times B(j, :)$. In this formulation, we access only the elements which contribute to the output. The working of the Row-Row formulation is shown below.

2.1.2.1 Example of Row-Row Formulation

$$Let\ A = \begin{bmatrix} 3 & 4 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 2 \end{bmatrix}\ and\ B = \begin{bmatrix} 0 & 2 & 4 \\ 0 & 1 & 0 \\ 4 & 0 & 0 \\ 6 & 1 & 0 \end{bmatrix}$$

$$C(1,:) = 3\times\begin{bmatrix} 0 & 2 & 4 \end{bmatrix}+4\times\begin{bmatrix} 0 & 1 & 0 \end{bmatrix}+1\times\begin{bmatrix} 4 & 0 & 0 \end{bmatrix}+0\times\begin{bmatrix} 6 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 4 & 10 & 12 \end{bmatrix}$$
$$C(2,:) = 2\times\begin{bmatrix} 0 & 2 & 4 \end{bmatrix}+0\times\begin{bmatrix} 0 & 1 & 0 \end{bmatrix}+0\times\begin{bmatrix} 4 & 0 & 0 \end{bmatrix}+1\times\begin{bmatrix} 6 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 6 & 5 & 8 \end{bmatrix}$$
$$C(3,:) = 0\times\begin{bmatrix} 0 & 2 & 4 \end{bmatrix}+1\times\begin{bmatrix} 0 & 1 & 0 \end{bmatrix}+1\times\begin{bmatrix} 4 & 0 & 0 \end{bmatrix}+0\times\begin{bmatrix} 6 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 4 & 1 & 0 \end{bmatrix}$$
$$C(4,:) = 0\times\begin{bmatrix} 0 & 2 & 4 \end{bmatrix}+0\times\begin{bmatrix} 0 & 1 & 0 \end{bmatrix}+1\times\begin{bmatrix} 4 & 0 & 0 \end{bmatrix}+2\times\begin{bmatrix} 6 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 16 & 2 & 0 \end{bmatrix}$$
$$C = (C(1,:);C(2,:);C(3,:);C(4,:)) = \begin{bmatrix} 4 & 10 & 12 \\ 6 & 5 & 8 \\ 4 & 1 & 0 \\ 16 & 2 & 0 \end{bmatrix}$$

2.1.3 The Column-Row Formulation

In the Column-Row formulation, for $i = 1, 2, ..., P$, we multiply the i^{th} column of A with the i^{th} row of B to get a matrix $C_i = A(:, i) \times B(i, :)$. The output matrix C is sum of all such matrices obtained, i.e., $C = \sum_{i=1} NC_i$. In this formulation also, we access only the elements which contribute to the output. An Example is given below.

2.1.3.1 Example of Column-Row Formulation

$$Let\ A = \begin{bmatrix} 3 & 4 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 2 \end{bmatrix}\ and\ B = \begin{bmatrix} 0 & 2 & 4 \\ 0 & 1 & 0 \\ 4 & 0 & 0 \\ 6 & 1 & 0 \end{bmatrix}$$

Let C_i denote the matrix obtained by multiplication of i^{th} column of A and i^{th} row of B.

$$C_1 = \begin{bmatrix} 3 & 2 & 0 & 0 \end{bmatrix}^T \times \begin{bmatrix} 0 & 2 & 4 \end{bmatrix} = \begin{bmatrix} 0 & 6 & 12 \\ 0 & 4 & 8 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$C_2 = \begin{bmatrix} 4 & 0 & 1 & 0 \end{bmatrix}^T \times \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 4 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$C_3 = \begin{bmatrix} 1 & 0 & 1 & 1 \end{bmatrix}^T \times \begin{bmatrix} 4 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 4 & 0 & 0 \\ 0 & 0 & 0 \\ 4 & 0 & 0 \\ 4 & 0 & 0 \end{bmatrix}$$

$$C_4 = \begin{bmatrix} 0 & 1 & 0 & 2 \end{bmatrix}^T \times \begin{bmatrix} 6 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 6 & 1 & 0 \\ 0 & 0 & 0 \\ 12 & 2 & 0 \end{bmatrix}$$

$$C = C_1 + C_2 + C_3 + C_4 = \begin{bmatrix} 4 & 10 & 12 \\ 6 & 5 & 8 \\ 4 & 1 & 0 \\ 16 & 2 & 0 \end{bmatrix}$$

2.1.4 The Column-Column Formulation

The Column-Column formulation is similar to the Row-Row formulation. Here column elements of B are used to scale the corresponding columns of A. An example is given below.

2.1.4.1 Example of Column-Column Formulation

$$Let\ A = \begin{bmatrix} 3 & 4 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 2 \end{bmatrix}\ and\ B = \begin{bmatrix} 0 & 2 & 4 \\ 0 & 1 & 0 \\ 4 & 0 & 0 \\ 6 & 1 & 0 \end{bmatrix}$$

$C(:,1) = 0\times\begin{bmatrix} 3 & 2 & 0 & 0 \end{bmatrix}^T + 0\times\begin{bmatrix} 4 & 0 & 1 & 0 \end{bmatrix}^T + 4\times\begin{bmatrix} 1 & 0 & 1 & 1 \end{bmatrix}^T + 6\times\begin{bmatrix} 0 & 1 & 0 & 2 \end{bmatrix}^T = \begin{bmatrix} 4 & 6 & 4 & 16 \end{bmatrix}^T$

$C(:,2) = 2\times\begin{bmatrix} 3 & 2 & 0 & 0 \end{bmatrix}^T + 1\times\begin{bmatrix} 4 & 0 & 1 & 0 \end{bmatrix}^T + 0\times\begin{bmatrix} 1 & 0 & 1 & 1 \end{bmatrix}^T + 1\times\begin{bmatrix} 0 & 1 & 0 & 2 \end{bmatrix}^T = \begin{bmatrix} 10 & 5 & 1 & 2 \end{bmatrix}^T$

$C(:,3) = 4\times\begin{bmatrix} 3 & 2 & 0 & 0 \end{bmatrix}^T + 0\times\begin{bmatrix} 4 & 0 & 1 & 0 \end{bmatrix}^T + 0\times\begin{bmatrix} 1 & 0 & 1 & 1 \end{bmatrix}^T + 0\times\begin{bmatrix} 0 & 1 & 0 & 2 \end{bmatrix}^T = \begin{bmatrix} 12 & 8 & 0 & 0 \end{bmatrix}^T$

$$C = (C(:,1), C(:,2), C(:,3)) = \begin{bmatrix} 4 & 10 & 12 \\ 6 & 5 & 8 \\ 4 & 1 & 0 \\ 16 & 2 & 0 \end{bmatrix}$$

2.2 Sparse Matrix Storage Formats

In this section we describe some of sparse matrix storage formats which we have used in our work.

2.2.1 Compressed Sparse Row (CSR) Format

This is the popular storage format. This format stores only required elements and does not make any assumptions about sparsity pattern of the matrix. Let A be a sparse matrix with dimensions $M \times N$ and has nnz non-zeros. In this format we use three arrays say $data, rowPtr, colPtr$ to store the matrix. The $data$ array contains only non-zero elements of matrix. The $colPtr$ stores column indices corresponding to the non-zeros in $data$ array. In the $rowPtr$ we store starting and ending indices of each row in $data$ array. $rowPtr[i], rowPtr[i+1]$ indicates the starting and ending indices of i^{th} row of A in $data$, $colPtr$ arrays. The sizes of $rowPtr$, $colPtr$, $data$ are $m+1$, nnz, nnz respectively. An example of CSR format is given in below.

Let $A = \begin{bmatrix} 1 & 3 & 0 & 0 \\ 0 & 0 & 7 & 5 \\ 0 & 4 & 8 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix}$ be the input matrix. Then the CSR representation is as follows,

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

$cols = \begin{bmatrix} 0 & 1 & 2 & 3 & 1 & 2 & 1 \end{bmatrix}$

$rowPtr = \begin{bmatrix} 0 & 2 & 4 & 6 & 7 \end{bmatrix}$

2.2.2 Coordinate (COO) Format

This is also another popular sparse matrix storage format. Let A be a sparse matrix with dimensions $M \times N$ and has nnz non-zeros. Similar to CSR format, it also uses three arrays to store the given matrix. These three arrays are $data$, $rowIndex$, $colIndex$. Similar to CSR format, $data$ array stores only non-zeros values of matrix A. $rowIndex$, $colIndex$ stores row value, column value corresponding to the non-zeros in $data$ array respectively. So the three arrays are each of size nnz each. An example of COO format is given below.

Let $A = \begin{bmatrix} 1 & 3 & 0 & 0 \\ 0 & 0 & 7 & 5 \\ 0 & 4 & 8 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix}$ be the input matrix. Then the COO representation is as follows,

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

$cols = \begin{bmatrix} 0 & 1 & 2 & 3 & 1 & 2 & 1 \end{bmatrix}$

$rows = \begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 3 \end{bmatrix}$

2.3 Scale-free matrices

2.3.1 Real-world graphs

2.3.2 Synthetic graphs

2.4 Platform

2.4.1 CPU: OpenMP - Application Program Interface

2.4.2 GPU: CUDA - Application Program Interface

2.4.3 Heterogeneous Computing

2.4.4 Experimental Platform

2.5 Datasets

2.6 Related Work

2.6.1 Sparse Matrix Sparse Matrix Multiplication (SPMM)

2.6.2 Sparse Matrix Dense Matrix Multiplication (CSRMM)

Chapter 3

Frontalization

In this chapter we present different types of matrix multiplications, sparse matrix storage formats, scale-free matrices, heterogeneous platforms, data-sets which were used in this thesis. Also we discuss some previous works that focus on workload balancing in a heterogeneous platform.

3.1 Matrix Multiplication Formulation

Let A, B and C be three matrices with sizes $M \times P$, $P \times N$ and $M \times N$ respectively such that $C = A \times B$. There are four different types of formulations to multiply two matrices. They are Row-Column formulation, Row-Row formulation, Column-Row formulation and Column-Column formulations. All these four formulations are briefly explained with an example.

3.1.1 The Row-Column Formulation

In the Row-Column formulation, to get one element in C, we multiply a row in the A matrix with a column in the B matrix, i.e., $C(i, j) = A(i, :) \times B(:, j)$ for $i = 1, 2, ...M$, and $j = 1, 2, ...N$. This is the standard matrix multiplication approach. For a given i, j, let $I(i, j)$ denote the set of index k such that both the elements $A(i, k)$ and $B(k, j)$ are nonzero. Then, $C(i, j) = \sum_{k \in I(i,j)} A(i, k) \times B(k, j)$. However, to obtain $I(i, j)$, we need to access all the elements in the i^{th} row of A and j^{th} column of B. Therefore, we bring in elements which may not contribute to the output. In the worst case, we would access the entire row i of A and a column j of B whereas $I(i, j) = \Phi$. Hence, this approach is not suited for sparse matrices in general.

3.1.2 The Row-Row Formulation

In the Row-Row formulation, to compute the i^{th} row in C, $C(i, :)$, we multiply each element in $A(i, :)$ with corresponding row in B. We then add all the scaled B rows to get the $C(i, :)$. Thus, $C(i, :) = \sum_{j \in A(i,:)} A(i, :) \times B(j, :)$. In this formulation, we access only the elements which contribute to the output. The working of the Row-Row formulation is shown below.

3.1.2.1 Example of Row-Row Formulation

$$Let\ A = \begin{bmatrix} 3 & 4 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 2 \end{bmatrix}\ and\ B = \begin{bmatrix} 0 & 2 & 4 \\ 0 & 1 & 0 \\ 4 & 0 & 0 \\ 6 & 1 & 0 \end{bmatrix}$$

$$C(1,:) = 3\times\begin{bmatrix} 0 & 2 & 4 \end{bmatrix}+4\times\begin{bmatrix} 0 & 1 & 0 \end{bmatrix}+1\times\begin{bmatrix} 4 & 0 & 0 \end{bmatrix}+0\times\begin{bmatrix} 6 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 4 & 10 & 12 \end{bmatrix}$$

$$C(2,:) = 2\times\begin{bmatrix} 0 & 2 & 4 \end{bmatrix}+0\times\begin{bmatrix} 0 & 1 & 0 \end{bmatrix}+0\times\begin{bmatrix} 4 & 0 & 0 \end{bmatrix}+1\times\begin{bmatrix} 6 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 6 & 5 & 8 \end{bmatrix}$$

$$C(3,:) = 0\times\begin{bmatrix} 0 & 2 & 4 \end{bmatrix}+1\times\begin{bmatrix} 0 & 1 & 0 \end{bmatrix}+1\times\begin{bmatrix} 4 & 0 & 0 \end{bmatrix}+0\times\begin{bmatrix} 6 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 4 & 1 & 0 \end{bmatrix}$$

$$C(4,:) = 0\times\begin{bmatrix} 0 & 2 & 4 \end{bmatrix}+0\times\begin{bmatrix} 0 & 1 & 0 \end{bmatrix}+1\times\begin{bmatrix} 4 & 0 & 0 \end{bmatrix}+2\times\begin{bmatrix} 6 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 16 & 2 & 0 \end{bmatrix}$$

$$C = (C(1,:);C(2,:);C(3,:);C(4,:)) = \begin{bmatrix} 4 & 10 & 12 \\ 6 & 5 & 8 \\ 4 & 1 & 0 \\ 16 & 2 & 0 \end{bmatrix}$$

3.1.3 The Column-Row Formulation

In the Column-Row formulation, for $i = 1, 2, ..., P$, we multiply the i^{th} column of A with the i^{th} row of B to get a matrix $C_i = A(:,i) \times B(i,:)$. The output matrix C is sum of all such matrices obtained, i.e., $C = \sum_{i=1} NC_i$. In this formulation also, we access only the elements which contribute to the output. An Example is given below.

3.1.3.1 Example of Column-Row Formulation

$$Let\ A = \begin{bmatrix} 3 & 4 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 2 \end{bmatrix}\ and\ B = \begin{bmatrix} 0 & 2 & 4 \\ 0 & 1 & 0 \\ 4 & 0 & 0 \\ 6 & 1 & 0 \end{bmatrix}$$

Let C_i denote the matrix obtained by multiplication of i^{th} column of A and i^{th} row of B.

$$C_1 = \begin{bmatrix} 3 & 2 & 0 & 0 \end{bmatrix}^T \times \begin{bmatrix} 0 & 2 & 4 \end{bmatrix} = \begin{bmatrix} 0 & 6 & 12 \\ 0 & 4 & 8 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$C_2 = \begin{bmatrix} 4 & 0 & 1 & 0 \end{bmatrix}^T \times \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 4 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$C_3 = \begin{bmatrix} 1 & 0 & 1 & 1 \end{bmatrix}^T \times \begin{bmatrix} 4 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 4 & 0 & 0 \\ 0 & 0 & 0 \\ 4 & 0 & 0 \\ 4 & 0 & 0 \end{bmatrix}$$

$$C_4 = \begin{bmatrix} 0 & 1 & 0 & 2 \end{bmatrix}^T \times \begin{bmatrix} 6 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 6 & 1 & 0 \\ 0 & 0 & 0 \\ 12 & 2 & 0 \end{bmatrix}$$

$$C = C_1 + C_2 + C_3 + C_4 = \begin{bmatrix} 4 & 10 & 12 \\ 6 & 5 & 8 \\ 4 & 1 & 0 \\ 16 & 2 & 0 \end{bmatrix}$$

3.1.4 The Column-Column Formulation

The Column-Column formulation is similar to the Row-Row formulation. Here column elements of B are used to scale the corresponding columns of A. An example is given below.

3.1.4.1 Example of Column-Column Formulation

$$Let\ A = \begin{bmatrix} 3 & 4 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 2 \end{bmatrix}\ and\ B = \begin{bmatrix} 0 & 2 & 4 \\ 0 & 1 & 0 \\ 4 & 0 & 0 \\ 6 & 1 & 0 \end{bmatrix}$$

$C(:,1) = 0\times\begin{bmatrix} 3 & 2 & 0 & 0 \end{bmatrix}^T + 0\times\begin{bmatrix} 4 & 0 & 1 & 0 \end{bmatrix}^T + 4\times\begin{bmatrix} 1 & 0 & 1 & 1 \end{bmatrix}^T + 6\times\begin{bmatrix} 0 & 1 & 0 & 2 \end{bmatrix}^T = \begin{bmatrix} 4 & 6 & 4 & 16 \end{bmatrix}^T$

$C(:,2) = 2\times\begin{bmatrix} 3 & 2 & 0 & 0 \end{bmatrix}^T + 1\times\begin{bmatrix} 4 & 0 & 1 & 0 \end{bmatrix}^T + 0\times\begin{bmatrix} 1 & 0 & 1 & 1 \end{bmatrix}^T + 1\times\begin{bmatrix} 0 & 1 & 0 & 2 \end{bmatrix}^T = \begin{bmatrix} 10 & 5 & 1 & 2 \end{bmatrix}^T$

$C(:,3) = 4\times\begin{bmatrix} 3 & 2 & 0 & 0 \end{bmatrix}^T + 0\times\begin{bmatrix} 4 & 0 & 1 & 0 \end{bmatrix}^T + 0\times\begin{bmatrix} 1 & 0 & 1 & 1 \end{bmatrix}^T + 0\times\begin{bmatrix} 0 & 1 & 0 & 2 \end{bmatrix}^T = \begin{bmatrix} 12 & 8 & 0 & 0 \end{bmatrix}^T$

$$C = (C(:,1), C(:,2), C(:,3)) = \begin{bmatrix} 4 & 10 & 12 \\ 6 & 5 & 8 \\ 4 & 1 & 0 \\ 16 & 2 & 0 \end{bmatrix}$$

3.2 Sparse Matrix Storage Formats

In this section we describe some of sparse matrix storage formats which we have used in our work.

3.2.1 Compressed Sparse Row (CSR) Format

This is the popular storage format. This format stores only required elements and does not make any assumptions about sparsity pattern of the matrix. Let A be a sparse matrix with dimensions $M \times N$ and has nnz non-zeros. In this format we use three arrays say $data, rowPtr, colPtr$ to store the matrix. The $data$ array contains only non-zero elements of matrix. The $colPtr$ stores column indices corresponding to the non-zeros in $data$ array. In the $rowPtr$ we store starting and ending indices of each row in $data$ array. $rowPtr[i], rowPtr[i+1]$ indicates the starting and ending indices of i^{th} row of A in $data$, $colPtr$ arrays. The sizes of $rowPtr$, $colPtr$, $data$ are $m+1$, nnz, nnz respectively. An example of CSR format is given in below.

Let $A = \begin{bmatrix} 1 & 3 & 0 & 0 \\ 0 & 0 & 7 & 5 \\ 0 & 4 & 8 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix}$ be the input matrix. Then the CSR representation is as follows,

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

$cols = \begin{bmatrix} 0 & 1 & 2 & 3 & 1 & 2 & 1 \end{bmatrix}$

$rowPtr = \begin{bmatrix} 0 & 2 & 4 & 6 & 7 \end{bmatrix}$

3.2.2 Coordinate (COO) Format

This is also another popular sparse matrix storage format. Let A be a sparse matrix with dimensions $M \times N$ and has nnz non-zeros. Similar to CSR format, it also uses three arrays to store the given matrix. These three arrays are $data$, $rowIndex$, $colIndex$. Similar to CSR format, $data$ array stores only non-zeros values of matrix A. $rowIndex$, $colIndex$ stores row value, column value corresponding to the non-zeros in $data$ array respectively. So the three arrays are each of size nnz each. An example of COO format is given below.

Let $A = \begin{bmatrix} 1 & 3 & 0 & 0 \\ 0 & 0 & 7 & 5 \\ 0 & 4 & 8 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix}$ be the input matrix. Then the COO representation is as follows,

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

$cols = \begin{bmatrix} 0 & 1 & 2 & 3 & 1 & 2 & 1 \end{bmatrix}$

$rows = \begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 3 \end{bmatrix}$

3.3 Scale-free matrices

3.3.1 Real-world graphs

3.3.2 Synthetic graphs

3.4 Platform

3.4.1 CPU: OpenMP - Application Program Interface

3.4.2 GPU: CUDA - Application Program Interface

3.4.3 Heterogeneous Computing

3.4.4 Experimental Platform

3.5 Datasets

3.6 Related Work

3.6.1 Sparse Matrix Sparse Matrix Multiplication (SPMM)

3.6.2 Sparse Matrix Dense Matrix Multiplication (CSRMM)

Chapter 4

Conclusions and Future Work

Conclusion

Related Publications

- *Kiran Raj Ramamoorthy, Dip Sankar Banerjee, Kannan Srinathan and Kishore Kothapalli,* A Novel Heterogeneous Algorithm for Multiplying Scale-Free Sparse Matrices, **IEEE - IPDPS, ASHES 2015**.
- *[Submitted] Hardhik Mallipeddi, Kiran Raj Ramamoorthy and Kishore Kothapalli,* Nearly Balanced Work Partitioning via Sampling for Heterogeneous Algorithms, **IEEE - 2016** .

Bibliography

[1] J. Alstott, E. Bullmore, and D. Plenz. Powerlaw: a Python package for analysis of heavy-tailed distributions. PLoS ONE 9(1): e85777, 2014.

[2] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report No. UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

[3] D. A. Bader, V. Agarwal, K. Madduri. On the Design and Analysis of Irregular Algorithms on the Cell Processor: A Case Study of List Ranking. IPDPS, 2007.

[4] D. A. Bader, and K. Madduri. GTgraph: A suite of synthetic graph generators. Available at https://sdm.lbl.gov/kamesh/software/GTgraph/

[5] D. S. Banerjee, K. Kothapalli. Hybrid algorithms for list ranking and graph connected components. HiPC, 2011.

[6] D. S. Banerjee, P. Sakurikar, and K. Kothapalli. Fast, scalable parallel comparison sort on hybrid multi-core architectures. In Proceedings of IPDPS Workshops, 2013.

[7] D. S. Banerjee, S. Sharma, and K. Kothapalli. Work efficient parallel algorithms for large graph exploration. HiPC 2013.

[8] M. Baskaran, and R. Bordawekar. Optimizing sparse matrix-vector multiplication on GPUs using compile-time and run-time strategies. Technical report, IBM Technical Report, 2008.

[9] N. Bell, and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report, 2008

[10] N. Bell, and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In Proceedings of SC, 2009.

[11] G. E. Blelloch, J. C. Hardwick, S. Chatterjee, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In Proceedings of PPOPP, 1993.

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

[13] M. Boyer, K. Skadron, S. Che, and N. Jayasena. Load Balancing in a Changing World: Dealing with Heterogeneity and Performance Variability. In Proceedings of ACM Computing Frontiers, 2013.

[14] Aydin Buluc and John R. Gilbert. Challenges and advances in parallel sparse matrix-matrix multiplication. In The 37th International Conference on Parallel Processing (ICPP), 2008.

[15] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In Proceedings of PPoPP, 2010.

[16] J. Chhugani, N. Satish, Changkyu Kim, J. Sewall, P. Dubey. Fast and Efficient Graph Traversal Algorithm for CPUs: Maximizing Single-Node Efficiency. IPDPS, 2012.

[17] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. Introduction to algorithms. MIT Press, 2001

[18] J.K. Cullum, and R.A. Willoughby. Lanczos Algorithms for Large Symmetric Eigenvalue Computations. Birkhauser, 1985.

[19] T. A. Davis. Direct Methods for Sparse Linear Systems. Society for Industrial and Applied Mathematics, 2006.

[20] M. deLorimier, and A. DeHon. Floating-point Sparse Matrix-Vector Multiply for FPGAs. FPGA, 2005.

[21] J. W. Demmel. Applied Numerical Linear Algebra. Society for Industrial and Applied Mathematics, 1997.

[22] M. Garland. Sparse matrix computations on many-core GPUs. In Proceedings of DAC, 2008.

[23] R. A. V. D. Geijn, and J. Watts. SUMMA: Scalable universal matrix multiplication algorithm. Concurrency: Practice and Experience, 1997.

[24] A. Gharaibeh, L. B. Costa, E. Santos-Neto, and M. Ripeanu. On graphs, GPUs, and blind dating: A workload to processor matchmaking quest. In Proceedings of IEEE IPDPS, 2013.

[25] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in Matlab: Design and implementation. SIAM Journal of Matrix Analysis and Applications, 1992.

[26] G.H. Golub, and C.F. Van Loan. Matrix Computations. 2nd edition. Johns Hopkins University Press, Baltimore, 1989.

[27] C. Gregg, M. Boyer, K. Hazelwood, and K. Skadron. Dynamic heterogeneous scheduling decisions using historical runtime data. in Workshop on Applications for Multi- and Many-Core Processors, 2011.

[28] J. Greiner. A comparison of parallel algorithms for connected components. In Proceedings of SPAA, 1994.

[29] D. Grewe and Michael F. P. OBoyle. A Static Task Partitioning Approach for Heterogeneous Systems Using openCL. In International Conference on Compiler Construction, 2011.

[30] F. Gustavson. Two fast algorithms for sparse matrices: multiplication and permuted transposition. ACM Transactions on Mathematical Software (TOMS), 1978.

[31] D. S. Hirschberg, A. K. Chandra, D. V .Sarwate. Computing connected components in parallel computers. Communications of the ACM, 1979.

[32] Sungpack Hong, T. Oguntebi, K. Olukotun. Efficient Parallel Graph Exploration on Multi-Core CPU and GPU. PACT, 2011

[33] S. Hong, N. C. Rodia, and K. Olukotun. On Fast Parallel Detection of Strongly Connected Components (SCC) in Small-World Graphs. In Proceedings of SC, 2013.

[34] Q. Hou, K. Zhou, and B. Guo. SPAP: A programming language for heterogeneous many-core systems. Technical report, Zhejiang University, Graphics and Parallel Systems Lab, 2010.

[35] S. Indarapu, M. Maramreddy, and K. Kothapalli. Architecture- and Workload-aware algorithms for Spare Matrix-Vector Multiplication. ACM Compute, 2014.

[36] K. Koer, I. Grasso, B. Cosenza, and T. Fahringer. An Automatic Input-Sensitive Approach for Heterogeneous Task Partitioning. In ACM ICS, 2013.

[37] K. Kothapalli, S. Indarapu, S. Sharma, D. S. Banerjee, and R. Nigam. Workload Aware Algorithms for Heterogeneous Platforms, Under Submission, Available at http://cstar.iiit.ac.in/kkishore/workqueue.pdf, 2013

[38] Zhiling Lan, Valerie E. Taylor, and Greg Bryan. A novel dynamic load balancing scheme for parallel systems. JPDC, 2002.

[39] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In Proceedings of the 37th annual international symposium on Computer architecture. ISCA 2010.

[40] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In Proceedings of ACM SIGKDD, 2005.

[41] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. Internet Mathematics, 2009.

[42] H. Ltaief, S. Tomov, R. Nath, and J. Dongarra. Hybrid Multicore Cholesky Factorization with Multiple GPU Accelerators. IEEE Transactions on Parallel and Distributed Computing, 2010.

[43] C. K. Luk, S. Hong, and H. Kim. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In Proceedings of International Symposium on Microarchitecture (MICRO), 2009.

[44] K. K. Matam, K. Kothapalli. Accelerating Sparse Matrix Vector Multiplication in Iterative Methods Using GPU. ICPP, 2011.

[45] K. K. Matam, S. B. Indarapu, and K. Kothapalli. Sparse matrix-matrix multiplication on modern architectures. HiPC, 2012.

[46] A. Monakov and A. Arutyun. Implementing Blocked Sparse Matrix-Vector Multiplication on NVIDIA GPUs. In Proceedings of SAMOS, 2009.

[47] A. Monakov, A. Avetisyan, and A. Lokhmotov. Automatically tuning sparse matrix-vector multiplication for GPU Architectures. In Proceedings of HiPEAC, 2010.

[48] R. Motwani, and P. Raghavan. Randomized Algorithms. Cambridge University Press, 2000.

[49] S. C. Park, J. P. Draayer, and S. Q. Zheng. Fast sparse matrix multiplication. Computer Physics Communications, 1992.

[50] Pawan Harish, and P. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In Proceedings of HiPC, 2007.

[51] G. Penn. Efficient transitive closure of sparse matrices over closed semi-rings. Theoretical Computer Science, 2006.

[52] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. Numerical Recipes, The Art of Scientific Computing. 2nd edition. Cambridge University Press, 1992.

[53] M. O. Rabin and V. V. Vazirani. Maximum matchings in general graphs through randomization. Journal of Algorithms, 1989.

[54] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for many-core GPUs. In Proceedings of IPDPS, 2009.

[55] D. P. Scarpazza, O. Villa, and F. Petrini. Efficient Breadth-First Search on the Cell/BE Processor. IEEE Transactions on Parallel Distributed Systems, 2008.

[56] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. 2007. Scan primitives for GPU computing. In Proceedings of GH, 2007.

[57] Y. Shan, W. Tianji, Y. Wang, B. Wang, Z. Wang, N. Xu, and H. Yang. FPGA and GPU implementation of large scale SpMV. SASP, 2010.

[58] J. Shen, A. L. Varbanescu, P. Zou, Y. Lu, and H. Sips. Improving Performance by Matching Imbalanced Workloads with Heterogeneous Platforms. In Proceedings of ACM ICS, 2014.

[59] Y. Shiloach, and U. Vishkin. An O(log n) parallel connectivity algorithm. Journal of Algorithms, 1982.

[60] J. Siegel, O. Villa, S. Krishnamoorthy, A. Tumeo, Xiaoming Li. Efficient sparse matrix-matrix multiplication on heterogeneous high performance systems. Cluster Computing Workshops and Posters, IEEE 2010.

[61] J. Soman, K. Kothapalli, and P. J. Narayanan. Some GPU Algorithms for Graph Connected Components and Spanning Tree, Parallel Processing Letters, 2010

[62] Fengguang Song, Asim YarKhan, and Jack Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multi-core systems. In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, 2009.

[63] P. D. Sulatycke, K. Ghose. Caching-efficient multi-threaded fast multiplication of sparse matrices. In Proceedings of IPDPS, 1998.

[64] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense liner algebra for hybrid GPU accelerated many-core systems. Parallel Computing, 2009.

[65] V. Volkov, and J.W. Demmel. Benchmarking GPUs to tune dense linear algebra. In Proceedings of SC, 2008.

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

[67] G. Wang and X. Ren. Power-efficient work distribution method for CPU-GPU heterogeneous system. In Proceedings of ISPA, 2010

[68] Zheng Wei, and Joseph JaJa. Optimization of linked list prefix computations on multi-threaded GPUs using CUDA. In Proceedings of IPDPS 2010.

[69] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multi-core platforms. In Proceedings of ACM SC, 2007.

[70] I. Yamazaki, Tingxing Dong, S. Tomov, J. Dongarra. Tri-diagonalization of a Symmetric Dense Matrix on a GPU Cluster. IPDPSW, 2013

[71] X. Yang, S. Parthasarathy, and P. Sadayappan, Fast Sparse Matrix-Vector Multiplication on GPUs: Implications for Graph Mining. In Proceedings of VLDB, 2011.

[72] R. Yuster, and U. Zwick. Fast sparse matrix multiplication. ACM Transactions on Algorithms, 2005

[73] L. Zhuo, V. K. Prasanna. Scalable and modular algorithms for floating-point matrix multiplication on FPGAs. In Proceedings of IPDPS, 2004.

[74] U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. Journal of the ACM, 2002.

[75] The Top 500 Super-computing sites. Available at http://www.top500.org

[76] Timothy A. Davis, The University of Florida Sparse Matrix Collection, In Proceedings of NA DIGEST 1994.

[77] Stanford Network Analysis Platform Dataset. http://www.cise.ufl.edu/research/sparse/matrices/SNAP/

[78] openMP Application Programming Interface. https://computing.llnl.gov/tutorials/openMP/

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

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

[81] nVidia cusp-library. http://cusplibrary.github.io/

[82] nVidia cusparse Library. http://developer.nvidia.com/cusparse/