A Project Report

On

Parallel programming using GPU

BY

Barkha Saxena

2014ABPS0735H

Under the supervision of

Anil Nemili

SUBMITTED IN PARTIAL FULLFILLMENT OF THE REQUIREMENTS OF

MATH F376: DESIGN ORIENTED PROJECT

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI (RAJASTHAN)

HYDERABAD CAMPUS

(OCTOBER 2017)

Acknowledgement

I would like to take this opportunity to thank my mentor Dr. Anil Nemili for his guidance and constant support. I am grateful to my senior Ridam Jain for helping me to complete this report.

I feel contented as I could apply the concept of parallel programming using GPU to the subproblem(sparse matrix multiplication) of my projects of pagerank retrieval and recommender system implementation.

Birla Institute of Technology and Science-Pilani,

Hyderabad Campus

Certificate

This is to certify that the project report entitled "**Parallel programming using GPU**" submitted by Barkha Saxena (ID No.2014ABPS0735H) in partial fulfillment of the requirements of the course MATH F376, Design Oriented Project Course, embodies the work done by her under my supervision and guidance.

Date: **(Anil Nemili)**

BITS- Pilani, Hyderabad Campus

Abstract

This report attempts to explain the need of GPU in complex computations with three example : matrix multiplication, Sparse matrix-vector multiplication (SpMV), and solving for laplace equation. Introduction includes the disadvantages of CPU computations over GPU computations.Followed by demonstrating the same with two different implementation of matrix multiplication on CPU and GPU.

For example, the implementation of recommender systems in computer science: a user-rating matrix (a 2D sparse matrix) is made,each cell containing rating of the user(corresponding row) for the movie(corresponding column).Preprocessing is done on this matrix according to the technique(mainly collaborative filtering, single value decomposition (SVD) and CUR). For instance in user-user based collaborative filtering, each row is centralized and converted to unit vector.Then to find the most similar users to the given user(for whom ratings are to be predicted for the given movie) ,we take cosine product of this matrix with user's rating vector.This cosine product step is the most time consuming step .

This report contains 3 implementations to optimize this step. As SpMV is bounded by memory constraints, the main focus is memory efficiency along with compact storage formats.After which the 3 different methods are efficiently performed on CUDA utilizing the parallel grained architecture of GPU. Sparse matrix-vector multiplication (SpMV) is peformed in each of the following formats : Diagonal Format(DIA), Coordinate Format(COO) and Compressed Sparse Row Format(CSR) .The formats along with their implementations are explained.

It was found that while the diagonal format is limited to the matrices of specified format although it takes into account the bandwidth efficiency but is computationally expensive.

The kernel for matrix multiplication using COO was found to be robust as the storage was linked to the number of non-zero elements rather than the dimensions of the sparse matrix.Thereby it offers consistent performances although it has the worst computational intensity.

CSR format was useful when we ensure contiguous memory access .Instead of using one thread per matrix , I made blocks of 16 threads , taking care of the memory colocaesion,the bandwidth was minimized.Although this depends on the warp size and the non-zeros per rows, hence yields poor performance in case of highly unstructured matrices.

CONTENTS

1. INTRODUCTION

1.1 GPGPU and its need

Graphics Processing Unit (GPU) manufacturers have introduced low level hardware to perform highly parallel computation tasks .This special hardware is known as General Purpose GPU cores (GPGPU).They are traditionally used for graphic rendering .This shift from CPU to GPGPU processing was motivated by the failure of performance scaling in general purpose CPUs. There has been a major impact on high performance computing in wide ranges of application domains. Areas of major influences include weather forecasting, molecular dynamics and fluid flow .With increasing acceptance of GPGPU technology in different sections, a significant development in engineering and scientific research is being observed.CUDA2 and OpenCL3 are the two recent frameworks highly adopted by third-party software developers.

This GPGPU hardware can be used for accelerating the characteristics of graphics algorithms which in turn will enable the development of special purpose graphics processors having extremely high performance. Most suited applications for GPU processing include physical simulation applications, which rely on use of linear algebra, requires multiplication of large matrices in real time and numerical solutions to PDE(partial differential equations). It has been observed that GPGPU technology is very well suited for applications based on pixels like computer vision, image and video processing. To help real time tracing, ray-object intersections are computed using GPGPU technology in ray tracing (realistic image synthesis algorithm requiring intensive computations).

The two properties of the algorithms used in GPGPU implementation are:

1. Intensive throughput: It implies processing of huge amount data in algorithm such that it can run simultaneously.
2. Data parallelism: It implies execution of operations on different data elements simultaneously. These two properties serve as an advantage in achieving high performance. Numerous simple processing units operate simultaneously on various data sets.

1.2 Basic difference between CPU and GPU

CPUs allot major resources (primarily chip area) for faster processing of single streams of instructions. To hide memory latency and to process streams of complex instructions it uses the concept of caching.

Whereas a chip of GPU contains several individual processing elements which further simultaneously execute single streams of instruction on various data elements. Here very fast context switching helps hide memory latency that is while one subset of data elements is being processed and a memory fetch is issued, then that subset is set aside. A huge advantage of GPUs over CPU is execution of certain algorithms is about 10 to 100 times faster.

1.3 Disadvantages

With major advantages ,this technology also has some disadvantages.

1) Additional Expense of incorporating GPU hardware into systems .Some applications might be served better by increasing the number of CPUs rather than bearing this cost . It also increases the power consumption and heat production in the system.

2) To gain advantages in various fields , their algorithms should be coded according to GPU architecture and there is major difference between programming for a GPU and traditional CPUs. Particularly, incorporation in pre-existing codes is more difficult than switching from one CPU family to another In order to make algorithm run according to GPGPUs significant changes to critical components are needed in the code.

2.Matrix Multiplication

Matrix multiplication is an example for data-parallel concurrency optimizations, since input data is unmodified and the output data range consists of a set of independent computation tasks. Besides being a basic operation for scientific computing, its optimization techniques are similar to many algorithms. Hence, matrix multiplication is one of the most fundamental examples in learning parallel programming.

Problem Statement : A matrix A of M rows and K columns,is to be multiplied with matrix B of K rows and N columns.And the result is to be stored in matrix C of M rows and N columns.

2.1 Naive Implementation On CPUs

Naïve matrix multiply algorithms are $O(n^3)$,and consist of a simple triple for-loop; the two outer loops iterate over the row and column index and the inner for loop performs a dot-product of the row & column vectors.

PSUEDO CODE

```
void main(){

    define  matrices A, B, C

    for i = 0 to M do

    //loop for traversing rows of matrix A
        for j = 0 to N do

        //loop for traversing rows of matrix A
          for k = 0 to K do
                        C(i,j) <= C(i,j) + A(i,k) * B(k,j)

                        /* compute element C(i,j) */
            end
        end
    end
}
```

To further make it simpler, square matrices (all M,N,K are equal) are used.

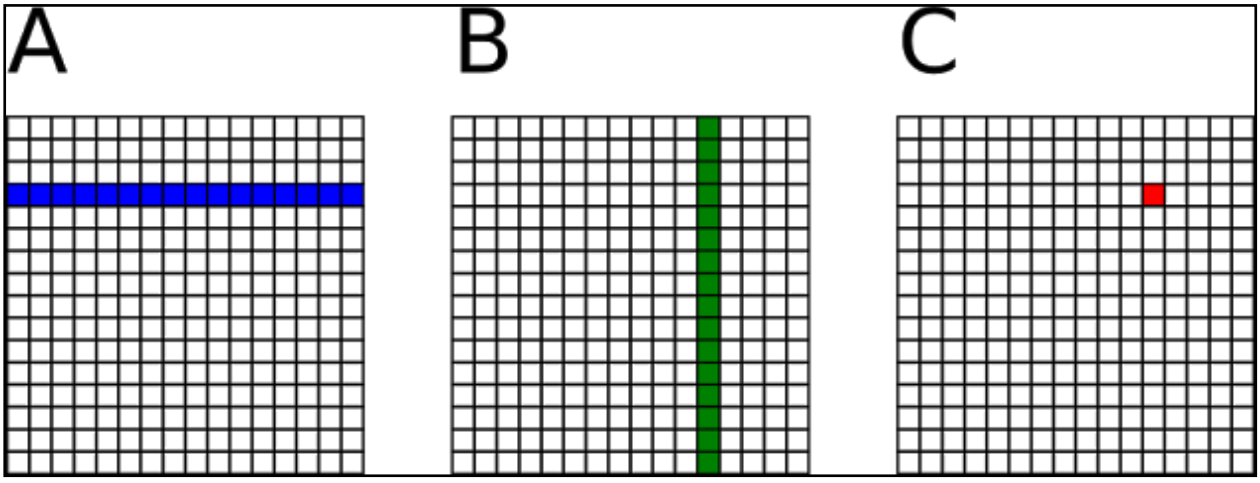


Illustration 1. The element of C (in red) can be calculated by multiplying one row of A (in blue) and one column of B (in green).

2.2 Naive Implementation On GPUs

In this implementation, each row of matrix A and each column of matrix B are loaded from global memory to GPU.Then a thread is assigned to compute corresponding element of matrix C.Hence the inner product is done and the results are stored back in the global memory.

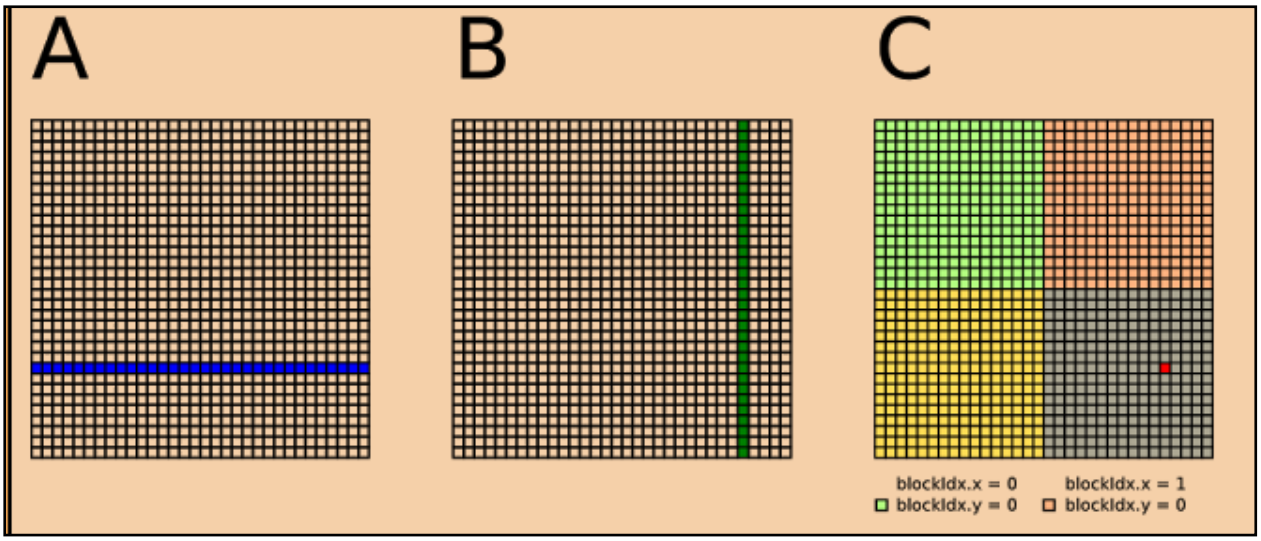


Illustration 2. This figure highlights the corresponding rows and columns which were multiplied to obtain the red highlighted cell in C.

PSUEDO CODE

```
/* Codes running on CPU */

void main(){

    define CPU_variable_A, CPU_variable_B, CPU_variable_Cin the CPU memory
    define GPU_variable_A, GPU_variable_B, GPU_variable_C in the GPU memory
        //copy from CPU to GPU
    memcopy CPU_variable_A to GPU_variable_A
    memcopy CPU_variable_B to GPU_variable_B
        //declaring matrices
    dim3 dimBlock(16, 16)
    dim3 dimGrid(No/dimBlock.x, Mo/dimBlock.y)

    matrixMul<<<dimGrid, dimBlock>>>(GPU_variable_A,
GPU_variable_B,GPU_variable_C,K)
    //calling function matrixMul

 memcopy GPU_variable_Cto CPU_variable_C
    //Copying the resuts back to CPU
}
```

```
/* pseudo Code as it is running on GPU */


__global__ void
matrixMul(GPU_variable_A,GPU_variable_B,GPU_variable_C,K){

    tempo <= 0

    i <= blockIdx.y * blockDim.y + threadIdx.y     // Row i of matrix C
    j <= blockIdx.x * blockDim.x + threadIdx.x    // Column j of matrix C

    for k = 0 to Ko-1 do
        var <= var + GPU_variable_A(i,k) * GPU_variable_B(k,j)
    end
   //calculating element of matrix C
    GPU_variable_C (i,j) <= var

}
```

This implementation is bandwidth bounded as the "computation-to-memory ratio" is approximately 1/4 (flop/byte) where computation amounts to (2 x Mo x No x Ko) flop, and the global memory access is (2 x Mo x No x Ko) word.

2.3 Increasing "Computation-to-Memory Ratio" by Tiling

The tiled matrix multiplication can be used to increase the "computation-to-memory ratio". Here matrices are divided into tiles. Each of this tiles is computed by a thread block. Each thread in the block calculates one element in the tile.

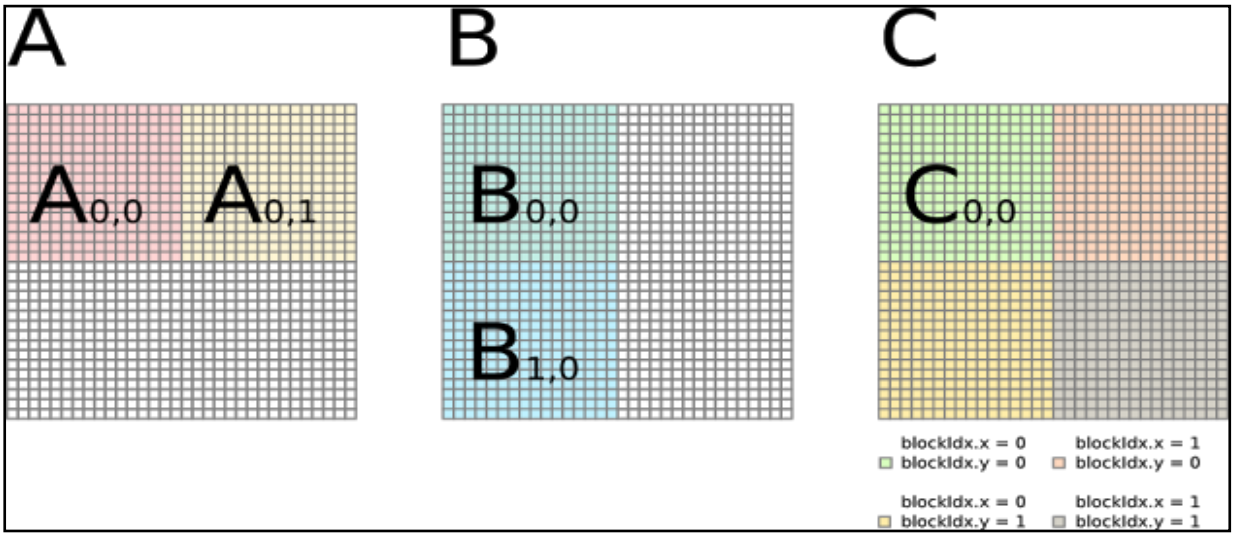
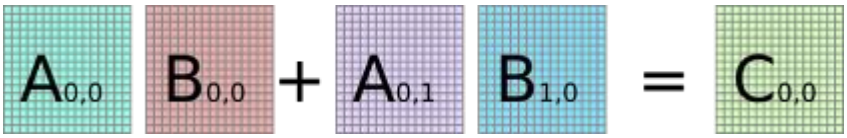


Illustration 3. The figure shows a 32 x 32 matrix divided into four 16 x 16 tiles.

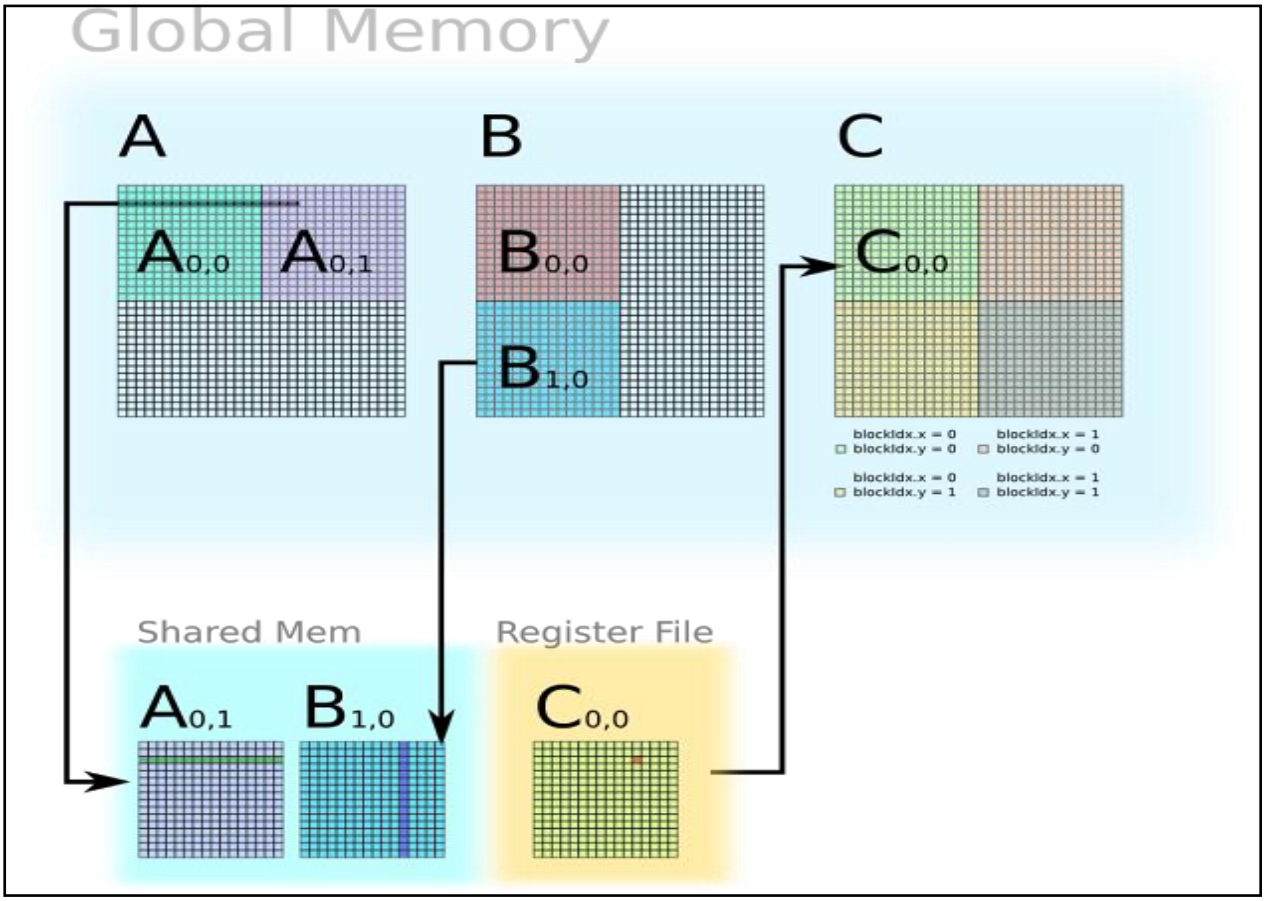
The matrix C is computed in multiple iterations in the GPU kernel. Every iteration consists of one thread block loading a tile of A and a tile of B from global to shared memory, on which computations are performed .The results of C are stored in register. After completion of all iterations, it stores that tile of C into global memory.

For example, a thread block can computer $C_{0,0}$ in two iterations: $C_{0,0} = A_{0,0} B_{0,0} + A_{0,1} B_{1,0}$.



In the first iteration,the tile $A_{0,0}$ and tile $B_{0,0}$ are loaded by the thread block from global memory into shared memory. Then inner product is performed by each thread resulting in an element of C,which is then stored in the register.

In the second iteration, the tile $A_{0,1}$ and tile $B_{1,0}$ are loaded by the thread block from global memory into shared memory. Then inner product is performed by each thread resulting in an element of C, which is added to the previous value stored in registers. After the final iteration this element of C is stored back to global memory .



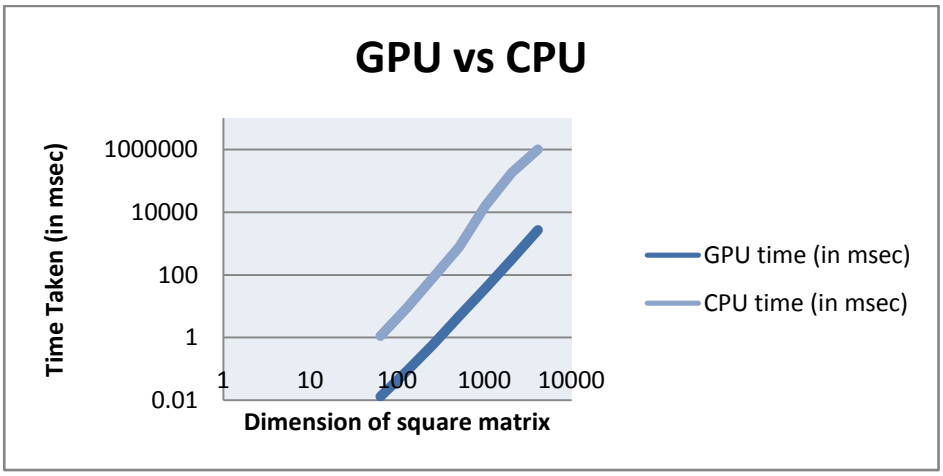
In this, the number of computation is still the same(2 x M x N x K) flop. However,the amount of global memory access is (2 x M x N x K / B) word by using the tile size B.So, the "computation-to-memory ratio" is approximately B/4 (flop/byte), which can be controlled by changing the value of B.

2.4 RESULT AND CONCLUSION

The matrix multiplication code was implemented on Intel(R) Core(TM) i5-3337U CPU @ 1.80GHz and GeForce GT 740M (384 CUDA cores) which yielded the following result.

Dimension of square matrix	GPU time (in msec)	CPU time (in msec)
64	0.013	1.113
128	0.083	8.667
256	0.585	79.589
512	4.747	760.879
1024	36.983	15773.227
2048	307.17	180919.692
4096	2650.335	1000879.994

Table 1. Showing the time taken by GPU and CPU for respective computation of matrix multiplication.

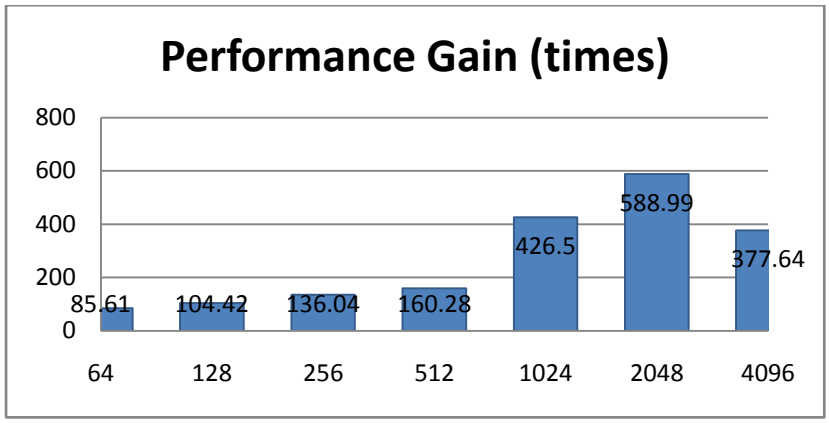


Graph 1. Values of Table 1 are plotted and hence comparision between CPU AND GPU processing is made.

From Table 1 and Graph 1 the clear performance gain by the GPU over the corresponding single-threaded CPU implementation can be observed clearly. Also from Table 2 and Graph 2 illustrates the increase in performance gain with the increase in the size of the matrix . Although it is difficult to discern from the scale of Graph, it was observed that the CPU is always faster than GPU for smaller datasets because to overcome the latency issues , GPU always requires a larger dataset.

size of matrix	Performance gain (times)
64	85.61
128	104.42
256	136.04
512	160.28
1024	426.5
2048	588.99
4096	377.64

Table 2 Showing the performance gain achieved by GPU over CPU corresponding to increasing matrix size.



Graph 2 Plot of Table 2 illustrating the increase in gain with increasing size of matrix.

Hence in domains where large datasets are needed to be used ,it is advantageous to use parallelized GPU computing in place of CPU computing. One of the applications can be in Ranked Retrieval model of a search engine where in order to calculate the relevance(cosine product), matrices of large sizes need to be multiplied .

3. Compression and Sparse matrix multiplication

3.1 Introduction and its need

Sparse matrix-vector multiplication (SpMV) holds utmost importance in solving iterative procedure for eigenvalues problems in sparse linear algebra. The immense parallelization and excellent performance of graphic processing units(GPUs) in high performance computing applications can be easily mapped to dense linear algebra, whereas additional problems of optimization are faced while utilizing this potential for sparse matrix computation.

For example, the implementation of recommender systems in computer science: a user-rating matrix (a 2D sparse matrix) is made,each cell containing rating of the user(corresponding row) for the movie(corresponding column).Preprocessing is done on this matrix according to the technique(mainly collaborative filtering, single value decomposition (SVD) and CUR). For instance in user-user based collaborative filtering, each row is centralized and converted to unit vector.Then to find the most similar users to the given user(for whom ratings are to be predicted for the given movie) ,we take cosine product of this matrix with user's rating vector.This cosine product step is the most time consuming step .

This report contains 3 implementations to optimize this step. As SpMV is bounded by memory constraints, the main focus is memory efficiency along with compact storage formats.After which the 3 different methods are efficiently performed on CUDA utilizing the parallel grained architecture of GPU. Sparse matrix-vector multiplication (SpMV) is peformed in each of the following formats : Diagonal Format(DIA), Coordinate Format(COO) and Compressed Sparse Row Format(CSR) .The formats along with their implementations are explained.

Problem Statement: Y=AX

Where :

A is a 2-dimensional sparse square matrix having N rows and N columns,

X is a column matrix, and

Y is the resultant matrix.

3.2 Concept of parallelization using CUDA

This programming model consist of user defined grid containg threadblocks each having 16 threads.The host program sequentially execute these threadblocks each at a time.In one threadblock all 16 threads run the same kernel concurrently. While execution of a block,it's threads can communicate amongst themselves and share the common private memory space of this block. A kernel is piece of computation code which is also called Single Program Multiple Data(SPMD).The hardware takes care of the creation and management of threads.

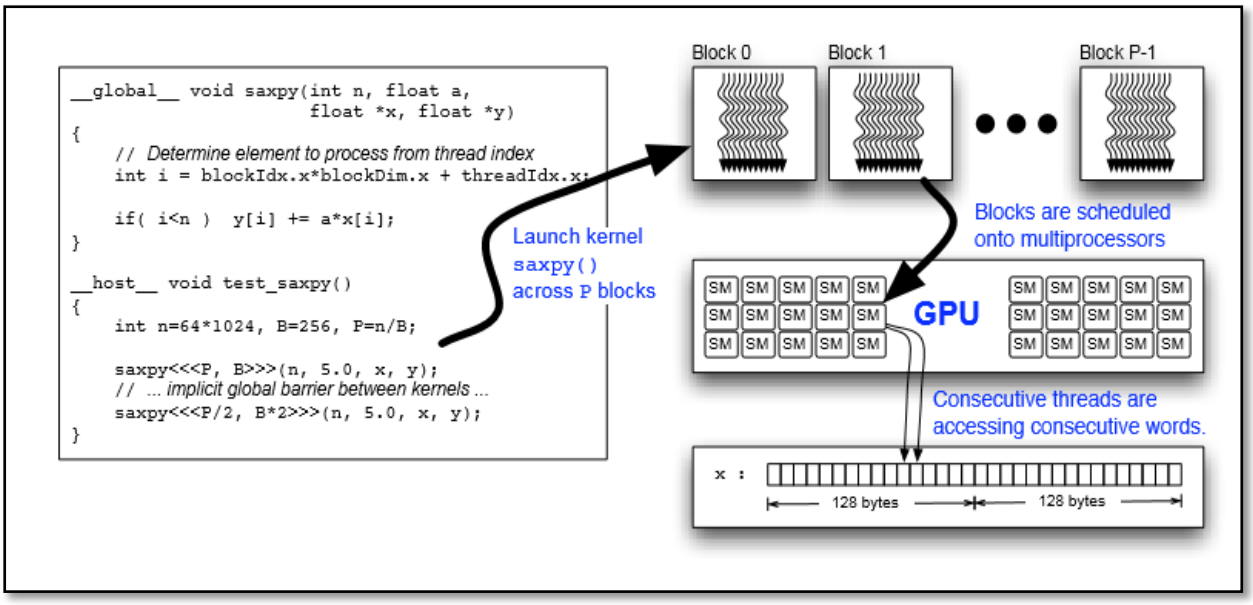


Figure 1. Explaining the parallelization of code.

Memory divergence : when the threads of a block access (load/store) oof-chip memory at scattered locations.

If we restructure in such a way that memory being accessed is coalesced to few segments of global memory, then we can greatly optimize the performance.

3.3 Diagonal Format(DIA) and its kernel

This format of representation is used when all the non-zeroes in the soarse matrix are present only in diagonals.This format can't be for general purpose unstructured matrices but it proves to be quite efficient in minimizing memory bandwidth in these particular cases.

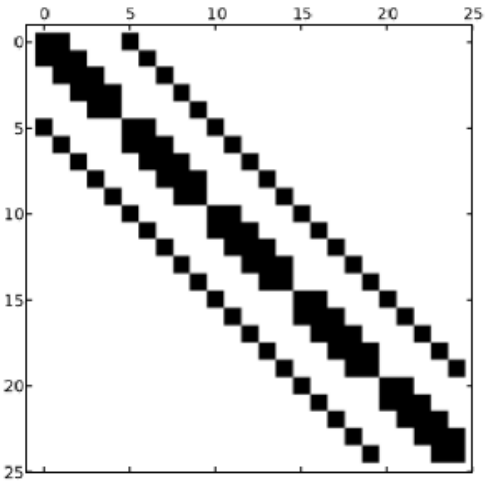


Figure 2: A typical pictorial representation of a 25 by 25 sparse matrix containing 5 non-zero diagonals

Numerical Representation of diagonal format :

It is formed by two arrays

$$\mathtt{A} = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

$$\mathtt{data} = \begin{bmatrix} * & 1 & 7 \\ * & 2 & 8 \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix} \qquad \mathtt{offsets} = \begin{bmatrix} -2 & 0 & 1 \end{bmatrix}$$

Figure 3: Example showing DIA format representation of sparse matrix A :

1. data array: consecutively storing all the non-zero elements of matrix A
2. offset array: storing the distance from the main diagonal .Conventionally the sub diagonals are stored with a negative sign whereas a positive sign is used for super diagonals.

Advantage : The rows and column indices are implicitly stored so this reduces the memory used significantly.As it is sotred incolumn major form the memory access is contiguous, threby improving memory transactions.
Disadvantage : It can't be used for general unstructured sparse matrix as it is limited to particular type of matrix format.

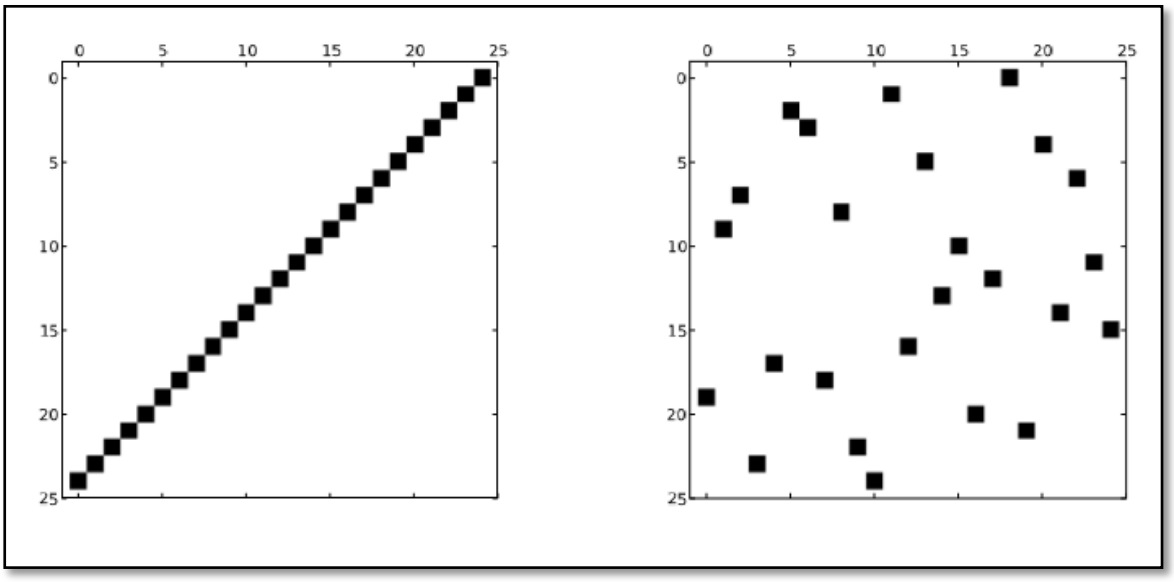


Figure3: Showing ill-suited matrix structures for DIA

Implementation of Matrix multiplication

A thread is assigned to each row of the matrix .The matrix is divided into threadblocks each containing 16 threads all of which parallely execute the same kernel.

Inside the kernel first the sequential row is calculated.Followed by fetching corresponding values of matrix A and matrix X and hence calculating dot product and storing in matrix Y.

```
__global__ void spmv_dia_kernel(const int total_rows , const int total_cols , const int
total_diagonal , const int * dia_offset, const float * data , const float * x, float * y)

{

        int r,i,c;

        float v=0.0;

        r = blockDim.x * blockIdx.x + threadIdx.x;

        if(r < total_rows)

        {

                float dot_product = 0;

                for(i = 0; i < total_diagonal; i++)

                {

                        c =  dia_offset[i]+r;

                        v = data[r+total_rows * i];

                        if(c >= 0 && c < total_cols) dot_product += v * x[c];

                }

    y[r] += dot_product;

    }

}
```

3.4 Coordinate Format(COO) and its kernel

This format checks for the non-zero elements and stores their corresponding row and column indices and data.It is used to represent any general arbitary sparse matrix representation

$$\mathtt{A} = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

$$\begin{aligned} \mathtt{row} &= \begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 2 & 3 & 3 \end{bmatrix} \\ \mathtt{col} &= \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix} \\ \mathtt{data} &= \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix} \end{aligned}$$

Where the arrays

data : stores all the non-zero elements consecutively

row: stores the indices of corresponding rows

col: stores the indices of corresponding rows

Advantage : It is storage invariant as the memory used for the arrays is proportional to the number of non-zero elements and not to the dimensions of the sparse matrix.This representation can be used for any general sparse matrix.

Disadvantage:The rows and column indices have to be stored explicitly.

3.5 Compressed Sparse Row Format(CSR) and its kernel

This is another representation for any unstructured sparse matrix.The row indices are explicitly stored whereas the column indices are implicitly stored.

$$\mathtt{A} = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

$$\begin{aligned} \mathtt{ptr} &= \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix} \\ \mathtt{indices} &= \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix} \\ \mathtt{data} &= \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix} \end{aligned}$$

Where the array

Ptr[N+1] : stores the total non-zero elements encountered before entering that row.

indices: stores the indices of corresponding rows

data : stores all the non-zero elements consecutively

Advantage : This is used for any unstructured sparse matrix.The column indices are implicitly stored.

Disadvantage: In general the performance doesn't achieve the required optimization.

Implementation of matrix multiplication

Each thread correspond to each row. Threadblocks of 16 thread run concurrently running the same kernel.In the kernel,the incides for the data array which are to be used are calculated for this thread.They are calculated with the help of ptr array.

After which corresponding elements of matrix X are retrieved and the cosine product is thus calculated.

```
__global__ void spmv_csr_kernel(const int total_rows , const int * ptr, const int * indices , const float * data , const float * x, float * y)

{
        int r,i, row_front, row_back;
        r = blockDim.x * blockIdx.x + threadIdx.x;
        //calculated row no in row
        if(r < total_rows)
        {
            float dot_product = 0.0;
          row_front = ptr[r];
          row_back = ptr[r+1];
         //for this row non-zero values are stored in data from row_start to row_end
          for (i= row_front; i < row_back; i++)
          dot += data[i] * x[indices[i]];
          //solution matrix
```

```
            y[r] += dot_product;
        }
}
```

3.4 Conclusion

We saw 3 implementations of sparse matrix multiplication(SpMV). As SpMV is bounded by memory constraints, the main focus is memory efficiency along with compact storage formats. Diagonal Format(DIA), Coordinate Format(COO) and Compressed Sparse Row Format(CSR) along with their implementations were studied.

It was found that while the diagonal format is limited to the matrices of specified format although it takes into account the bandwidth efficiency but is computationally expensive.

The kernel for matrix multiplication using COO was found to be robust as the storage was linked to the number of non-zero elements rather than the dimensions of the sparse matrix.Thereby it offers consistent performances although it has the worst computational intensity.

CSR format was useful when we ensure contiguous memory access .Instead of using one thread per matrix , I made blocks of 16 threads , taking care of the memory colocaesion,and hence the bandwidth was minimized.Although this depends on the warp size and the non-zeros per rows, hence yields poor performance in case of highly unstructured matrices.

4. Laplace Equation

4.1 Introduction

Laplace equation is a partial differential equation of second order :

$$\nabla^2 u=0$$

$$\text{or } (\frac{\delta^2}{\delta x^2}+\frac{\delta^2}{\delta y^2}+\frac{\delta^2}{\delta z^2})u = 0$$

Where

u(r) is a function of spatial co-ordinates,
r = (x,y,z), and
∇^2 is the laplacian operator.

When the the boundary conditions on a domain D are known and we have to find the solution Ø inside it, then this type of problem is known as Dirichlet problem for Laplace's equation.For example Laplace eqution is valid on heat equations,so we can fix the boundary temperature of

the domain and solve for the temperature distributuion inside.The temperatures at each point is calculated iteratively until a stationary state is achieved i.e the temperatures doesn't vary in further iterations.Hence the temperature distribution for the interior points is calculated by using this formula

In two dimension laplace equation is used for solving

1. Analytical function
2. Fluid flow
3. Electrostatics

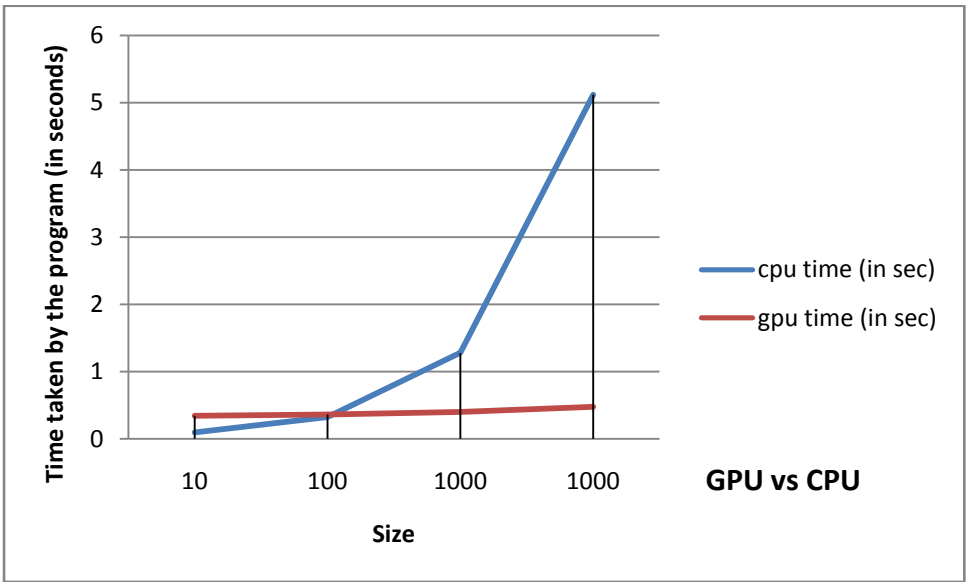
In three dimension, it is used to solve green's equation.
Within a domain,t he solutions of Laplace's equation are analytic in nature.That implies a linear combination of solutions of laplace's equation is also a solution to it.They are all harmonic functions.This property of superposition helps in solving many complex problems easily.

4.2 Problem Statement

A thin 2-dimensional rectangular solid plate with specified boundary temperatures is studied.The solution of laplace equation is used to determine the temperature distribution inside the plate.The plate is divided into points forming a computational grid. At each point the laplace equation is solved.

4.3 Result and conclusion

As it can be seen from the graph plotted between time taken by the program and the size of meshgrid, we can conclude at size less than ~10^2 time taken by GPU is more as compared to CPU for solving the laplace equation.But at higher ranges of size GPU outperforms CPU significantly.



Graph 3 Comparing the GPU vs CPU performance.

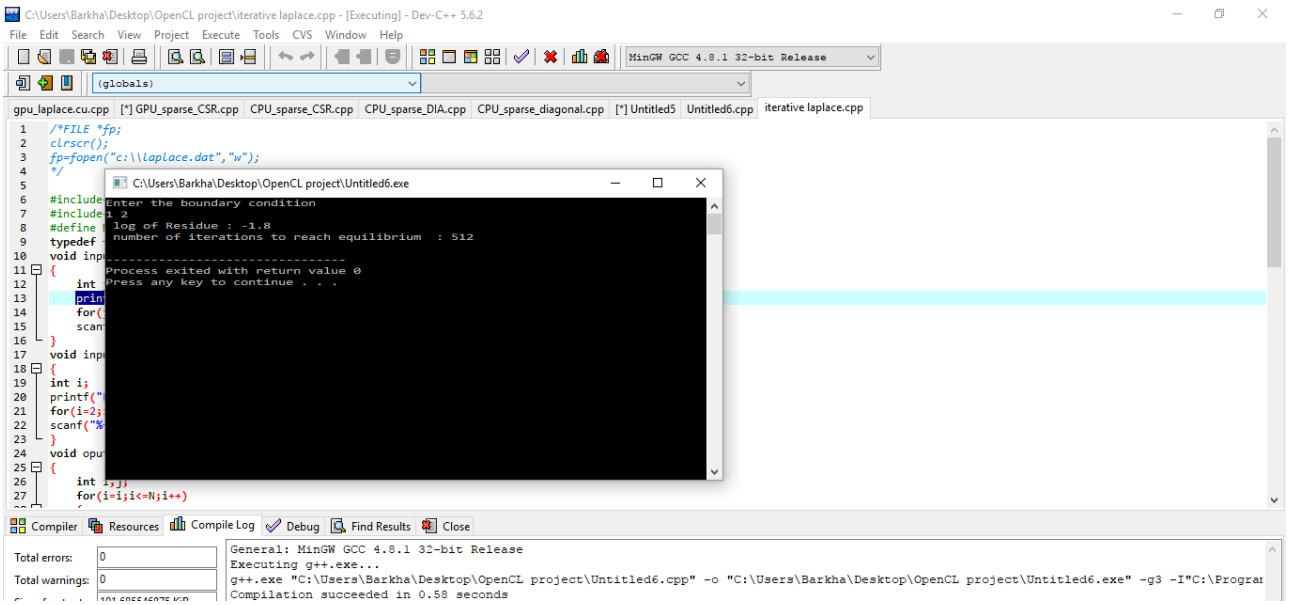


Figure 5: Showing the CPU implementation of laplace code with the residue and iterations done.

The residue is calculated in each iteration by finding the difference between new calculated value at that point and the old value from the previous iteration. Then it plotted on logarithmic scale to conclude as the number of iterations increases it decreases from 0(log new_res/old_res) to negative values.And around 5000 iterations it stabilises to -2.5 approximately.

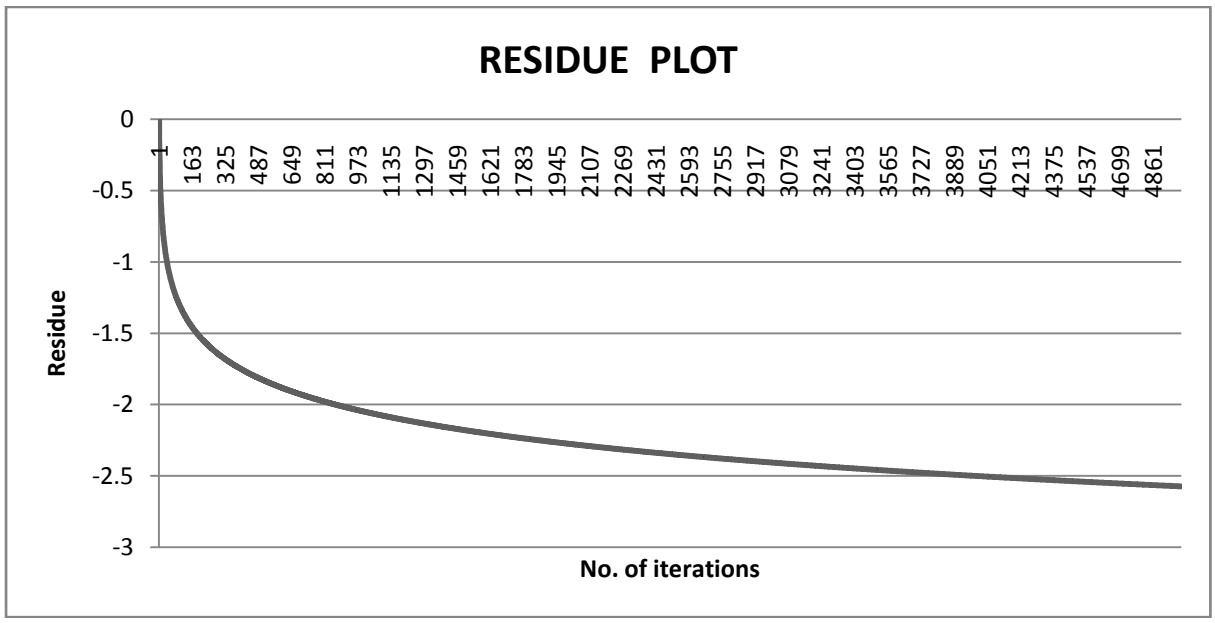


Figure 6: Logarithmic plot of residue function with respect to the number of iterations.

5. REFERENCES

1. http://www.es.ele.tue.nl/~mwijtvliet/5KK73/?page=mmcuda
2. NVIDIA. CUDA v5.0 documentation. http://docs.nvidia.com/cuda/index.html
3. www.wikipedia.org
4. For the sparse matrix multiplication : http://www.nvidia.in/docs/IO/66889/nvr-2008-004.pdf