Università della Svizzera italiana

GPU Implementation of the Lean Algebraic Multi-grid (LAMG) Solver for Large-scale Graph Problems

Bachelor Project

Author:
Satish Kumar

Supervisor:
Prof Rolf Krause
Riva Simone

December 2, 2018

Contents

Abstract 4

1 Introduction 1
- 1.1 Multi-grid Problem 1
  - 1.1.1 Model Problem 1
    - 1.1.1.1 One-dimensional boundary value problem: 1
    - 1.1.1.2 Solution Methods 2
    - 1.1.1.3 A two-dimensional boundary value problem 3
    - 1.1.1.4 Computation Costs 4
- 1.2 OCCA 4
  - 1.2.1 History 4
  - 1.2.2 Overview 5
  - 1.2.3 Device 5
  - 1.2.4 Memory management 6
  - 1.2.5 kernel 6
- 1.3 Compressed Row Storage (CRS) 7
  - 1.3.1 Introduction 7

2 Dense Matrix 9
- 2.1 Dense matrix - vector multiplication 9
  - 2.1.1 introduction 9
  - 2.1.2 CPU implementation 10
  - 2.1.3 GPU implementation 10
  - 2.1.4 GPU vs CPU 11
- 2.2 Dense matrix multiplication 11
  - 2.2.1 introduction 11
  - 2.2.2 CPU implementation 12
  - 2.2.3 GPU implementation 13
  - 2.2.4 GPU vs CPU 13
- 2.3 Dense matrix - matrix addition & subtraction 14
  - 2.3.1 Introduction 14
  - 2.3.2 CPU implementation 14
  - 2.3.3 GPU implementation 15
  - 2.3.4 GPU vs CPU 16

3 Sparse Matrix 17
- 3.1 introduction 17

3.1.1 Storing . . . 17
3.1.2 Storing formats . . . 17
3.2 Sparse matrix - vector multiplication . . . 18
3.2.1 introduction . . . 18
3.2.2 CPU implementation . . . 19
3.2.3 GPU implementation . . . 19
3.2.4 GPU vs CPU . . . 20
3.3 Sparse matrix multiplication . . . 20
3.3.1 introduction . . . 20
3.3.2 CPU implementation . . . 21
3.3.3 GPU implementation . . . 22
3.3.4 GPU vs CPU . . . 23
3.4 Sparse matrix - matrix addition & subtraction . . . 23
3.4.1 introduction . . . 23
3.4.2 CPU implementation . . . 24
3.4.3 GPU implementation . . . 25
3.4.4 GPU vs CPU . . . 26

Dot product **27**
Introduction . . . 27
CPU implementation . . . 27
GPU implementation . . . 28
GPU vs CPU . . . 29

Multi-grid Method **30**
Introduction . . . 30
Solution Methods . . . 30
Gaussian elimination . . . 31
Introduction . . . 31
CPU implementation . . . 31
GPU implementation . . . 32
GPU vs CPU . . . 33
Jacobi Method . . . 34
Introduction . . . 34
CPU implementation . . . 34
GPU implementation . . . 35
GPU vs CPU . . . 35
Multi-grid mehtod Dense matrix . . . 36
Introduction . . . 36
CPU implementation . . . 37
GPU implemintation . . . 38
GPU vs CPU . . . 40
Multi-grid mehtod sparse matrix . . . 40
CPU implementation . . . 40
GPU implementation . . . 43
GPU vs CPU . . . 45

Conclusion **46**

Bibliography **48**

Bibliography **48**

Abstract

Graphic Processing Unit (GPU) has become one of the most important components in modern computer systems. GPUs have evolved from a single-purpose graphic rendering hardware to a powerful processor that is capable of handling many different kinds of computing task.
In this report we have created linear algebra API using OCCA and C++.
We demonstrated that the individual linear algebra components can be faster when using the GPU as compared to the CPU.
We have worked in OCCA because OCCA is a open-source library.
It provide a kernel language, a minor extension to C. It is easy to understand.
OCCA supports device kernel expansion for the OpenCL, OpenMP and CUDA.
We implemented the matrix multiplication. Multiplication between matrix and vector, Sparse matrix in CSR format, sparse matrix in CSR format with vector multiplication, matrix - matrix addition, matrix - matrix subtraction , dot product between vectors, with reduction and Multi-grid method for dense matrix and sparse matrix. Thus we have analyzed and compared the performance between GPU and CPU.

Chapter 1

Introduction

1.1 Multi-grid Problem

1.1.1 Model Problem

Multigrid methods were originally applied to simple boundary value problems that arise in many physical applications. For simplicity and for historical reasons, these problems provide a natural introduction to multi-grid methods.

1.1.1.1 One-dimensional boundary value problem:

$-u''(x) + \alpha u(x) = f(x) \quad 0 < c < 1, \alpha > 0$

$u(0) = u(1) = 0$

While this problem can be handled analytically, our present aim is to consider numerical methods. Many such approaches are possible, the simplest of which is a finite difference method. The domain of the problem $x : 0 \leq x \leq 1$ is partitioned into n subintervals by introducing the grid points $X_j = j_h$, where h = 1/n is the constant width of the subintervals. which we denote h. At each of the n-1 interior grid points, the original differential equation (1.1) is replaced by a second-order finite difference approximation. In making this replacement, we also introduce as an approximation to the exact solution $U(X_j)$. This approximate solution may now be represented by a vector $v = (v_i, ..., v_{n-i})^T$, whose components satisfy the n —l linear equations

Defining $f = (f(x_1), ..., f(x_{n_1}))^T = (f_1, ..., f_{n-i})^T$, the vector of right-side values, we may also represent this system of linear equations in matrix form as

$$1/h^2 \begin{bmatrix} 2+\alpha h^2 & -1 & & & \\ -1 & 2+\alpha h^2 & -1 & & \\ & . & . & . & \\ & & . & . \quad . & -1 \\ & & -1 & -1 & 2+\alpha h^2 \end{bmatrix} \begin{bmatrix} v_1 \\ . \\ . \\ . \\ v_{n-1} \end{bmatrix} = \begin{bmatrix} f_1 \\ . \\ . \\ . \\ f_{n-1} \end{bmatrix}$$

or even more compactly as Av = f. The matrix A is (n —1) x (n —1), tridiagonal, symmetric and positive definite.

1.1.1.2 Solution Methods

To solve the systems of linear equations there are several methods of solution:

• Direct

– Gaussian elimination

– Factorisation

• Iterative

– Jacobi

– Gauss-Seidel

– Conjugate Gradient, etc.

When it comes to choosing between direct or iterative solution methods, there are several factors to consider.

The first consideration is the application and the computer that is used. Since direct methods are expensive in terms of memory and time intensive for CPUs, they are preferable for small to medium-sized 2D and 3D applications. Conversely, iterative methods have a lower memory consumption and for large 3D applications, they outperform direct methods. Further, it is important to note that iterative methods are more difficult to tune and more challenging to get working for matrices arising from multi-physics problems.

1.1.1.3 A two-dimensional boundary value problem

$-u''(xx) - u(yy) + \alpha u(x) = f(x,y) \quad 0 < x < 1, \quad 0 < y < 1,$

$u = 0, x = 0, x = 1, y = 0, y = 1 \quad \alpha > 0$

We obtain a block-tridiagonal system Av=f

$$\begin{bmatrix} A_1 & -I_y & & & \\ -I_y & A_2 & -I_y & & \\ & . & . & . & \\ & & . & . & . \\ & & & -I_y & A_{N-1} \end{bmatrix} \begin{bmatrix} v_1 \\ . \\ . \\ . \\ v_{N-1} \end{bmatrix} = \begin{bmatrix} f_1 \\ . \\ . \\ . \\ f_{N-1} \end{bmatrix}$$

where I_y is a diagonal matrix with $1/h_y^2$ on the diagonal

The main idea of multi-grid is to accelerate the convergence of a basic iterative method (known as relaxation, which generally reduces short-wavelength error) by a global correction of the fine grid solution approximation from time to time accomplished by solving a coarse problem. The coarse problem, while cheaper to solve is similar to the fine grid problem in that it also has short and long-wave length errors. It can also be solved by a combination of relaxation and appeal to still coarser grids. This recursive process is repeated until a grid is reached where the cost of direct solution there is negligible compared to the cost of one relaxation sweep on the fine grid. This multi-grid cycle typically reduces all error components by a fixed amount bounded well below one independent of the fine grid mesh size. The typical application for multi-grid is in the numerical solution of elliptic partial differential equations in two or more dimensions.

There are many variations of multi-grid algorithms, but the common features are that a hierarchy of discretisations (grids) is considered. The important steps are:

Smoothing – reducing high frequency errors, for example using a few iterations of the Gauss–Seidel method.

Restriction – downsampling the residual error to a coarser grid.

Interpolation or prolongation – interpolating a correction computed on a coarser grid into a finer grid.

1.1.1.4 Computation Costs

Let 1 Work Unit(WU) be the cost of one relaxation sweep on the fine-grid.

• Ignore the cost of restriction and interpolation (typically about 20 percentage of the total cost).

• Consider a V-cycle with 1 pre-Coarse-Grid correction relaxation sweep and 1 post-Coarse- Grid correction relaxation sweep.

• Cost of V-cycle(in WU):

$2(1 + 2^{-d} + 2^{-2d} + 2^{-3d} + ... + 2^{-Md} < \frac{2}{1-2^{-d}}$

Cost is about 4,8/3,16/7 WU per V-cycle in 1,2 and 3 dimensions.

• Multi grid has been proven on a wide variety of problems, especially elliptic PDEs, but has also found application among parabolic & hyperbolic PDEs, integral equations, evolution problems, geodesic problems etc.

• With the right setup, multi grid is frequently an optimal (i.e., O(N)) solver.

• Multi grid is of great interest because it is one of the very few scalable algorithms, and can be parallelised readily and efficiently!

1.2 OCCA

OCCA

1.2.1 History

OCCA (like oca-rina) started off a project in Tim Warburton's group. The group mainly worked high-order numerical methods, specifically on the algorithms to make them performant. During that time, they mainly focused on GPU programming using OpenCL and CUDA.

They had wrappers for OpenCL and CUDA to test implementations, which we almost always had 2 almost identical codes to run on NVIDIA and AMD GPUs.

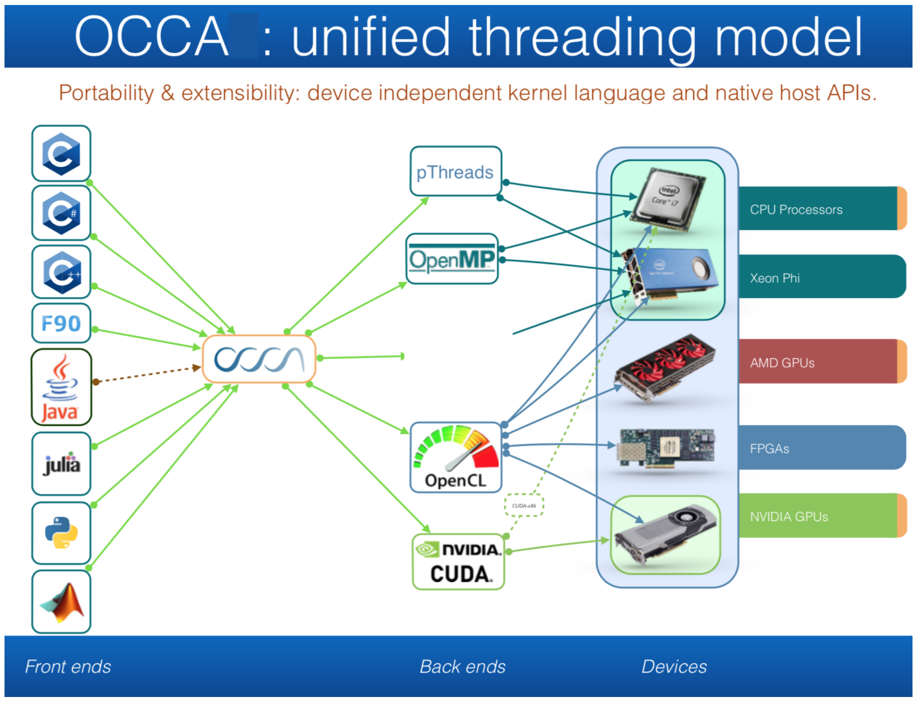


FIGURE 1.1: Current relationship between supported frontends, the OCCA languages, and supported backends

1.2.2 Overview

The different projects mentioned have focused on creating some mapping between two or more programming languages for these assertors or switching between multiple platform for computation purpose. OCCA, a library including an API that abstracts back-ends and kernel languages from OpenMP, OpenCL and CUDA.

1.2.3 Device

Graphics cards were developed due to the increasing demand for improved graphics in video games. The similarities between CUDA and OpenCL become evident in their programming model but their popularity in use differ. NVIDIA releases their own compiler wrapper, nvcc to allow CUDA kernels to be embedded in the application code. While OpenCL separates host code (application code) with the device code (kernels), which steepens the learning curve.
An implementation of this concept was developed, producing the OCCA intermediate representation (IR) which generalises current parallel architectures to unify the different

languages and standards for heterogeneous computing, including serial code, Pthreads, OpenMP, CUDA, OpenCL

In OCCA we can define the target device runtime, with the following simples lines of codes.

"mode: 'Serial'"

We can used the device manually also

CUDA	"mode: 'CUDA', deviceID: 0"
OpenCL	"mode: 'OpenCL', deviceID: 0, platformID: 0"
THREADS	"mode: 'Threads', threads: 4, pinnedCores: [0, 1, 2, 3]"
OPENMP	"mode: 'OpenMP', threads: 4"

1.2.4 Memory management

The memory in the device. We can not allocate directly memory in the device. Hence in the host, data is usually initialised either copied to the device, modified in the device by running a kernel in the device.

In OCCA we can define the target device runtime, with the following simples lines of codes.

```
// Copy a and b to the device
occa::memory o_a  = device.malloc(entries * sizeof(float), a);
occa::memory o_b  = device.malloc(entries * sizeof(float), b);
// Don't initialise o_ab
occa::memory o_ab = device.malloc(entries * sizeof(float))
```

1.2.5 kernel

Okl now has the feature to automatically assign working dimensions to the off-load model through the outer and inner loops. A kernel launch in OpenCL and CUDA are always separate from the kernel source, but maintain a connection through the working dimensions used in a kernel execution.

Kernel are build at runtime so we require 2 things

1. file name with the kernel source code
2. Name of the kernel in source code

```
occa::kernel addVectors = device.buildKernel("addVectors.okl",
                                              "addVectors")
    // we call to kernel function
    addVectors(n, o_a, o_b, o_ab);
```

Above, We are using function buildKernel for building kernel at runtime. Where addVectors.okl is a file name with extension okl and addVectors is the kernel function name. And next-step we call to kernel with its variables.

1.3 Compressed Row Storage (CRS)

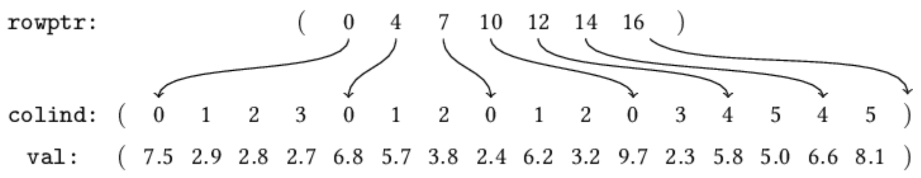
Compressed Row Storage (CRS)

1.3.1 Introduction

The Compressed Row and Column Storage formats are the most general. They make absolutely no assumptions about the sparsity structure of the matrix and they don't store any unnecessary elements. On the other hand, they are not very efficient, needing an indirect addressing step for every single scalar operation in a matrix-vector product or preconditioned solve. For CRS, we create 3 vectors and size of the vectors are length of non-zero elements of matrix. They are

- Non-zero elements of the matrix
- column number for non-zero elements of the matrix
- row number for non-zero elements of the matrix

$$A = \begin{pmatrix} 7.5 & 2.9 & 2.8 & 2.7 & 0 & 0 \\ 6.8 & 5.7 & 3.8 & 0 & 0 & 0 \\ 2.4 & 6.2 & 3.2 & 0 & 0 & 0 \\ 9.7 & 0 & 0 & 2.3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5.8 & 5.0 \\ 0 & 0 & 0 & 0 & 6.6 & 8.1 \end{pmatrix}$$



```
rowptr:            ( 0   4   7   10  12  14  16 )
colind: ( 0   1   2   3   0   1   2   0   1   2   0   3   4   5   4   5 )
val:    ( 7.5 2.9 2.8 2.7 6.8 5.7 3.8 2.4 6.2 3.2 9.7 2.3 5.8 5.0 6.6 8.1 )
```

The first vector represents the non-zero values of the matrix, read first by row left to right, then by column top to bottom. The second vector represents the column index corresponding to the values. The third vector represents the indexes belonging to a row, it contains an index per row corresponding to an index in the two other vectors. It is clear that all indexes from this starting index and to the starting index of the next row will belong to the given row. The last index is the number of rows plus one, so the algorithm doesn't have to check if we're at the last row.

Chapter 2

Dense Matrix

2.1 Dense matrix - vector multiplication

2.1.1 introduction

Matrix vector multiplication computes the product of matrix and vector. We can perform multiplications between a matrix and vector when number of columns of matrix equal number of rows of vector. In mathematical term m*n dimensional matrix can be multiplied only n dimensional vector.The theoretical operation is shown below, matrix-vector multiplication output is m*1 dimensional vector.
Input data: dense matrix A of size m-by-n (with entries a_{ij}), its vector cofactor x of dimension n (with components x_i).

Output data: vector y of dimension m (with components y_i).

Formulas of the method:

$y_i = \sum_{j=1}^{n} a_{ij} x_j$ i $\in [1...m]$

There also exists a block version of the method. However, this description treats only the pointwise version.

$$\begin{pmatrix} 2 & 3 \\ 4 & 5 \\ -1 & 6 \end{pmatrix} \times \begin{pmatrix} 4 \\ 7 \end{pmatrix} = \begin{pmatrix} (2 * 4) + (3 * 7) \\ (4 * 4) + (5 * 7) \\ (-1 * 4) + (6 * 7) \end{pmatrix} = \begin{pmatrix} 29 \\ 51 \\ 38 \end{pmatrix}$$

2.1.2 CPU implementation

The CPU implementation is the most basic implementation. The idea is below

```
we have m*n dimensional matrix
for i < m; i++
  for j < n; j++
    result_vector[i] += matrix[i*n+j]*vector[j]
```

In each iteration of the for loops, the code computes the product of two elements of matrix and vector and add the result. So the time complexity of the entire multiplication is $O(n^2)$, if we have matrix size n-by-n.

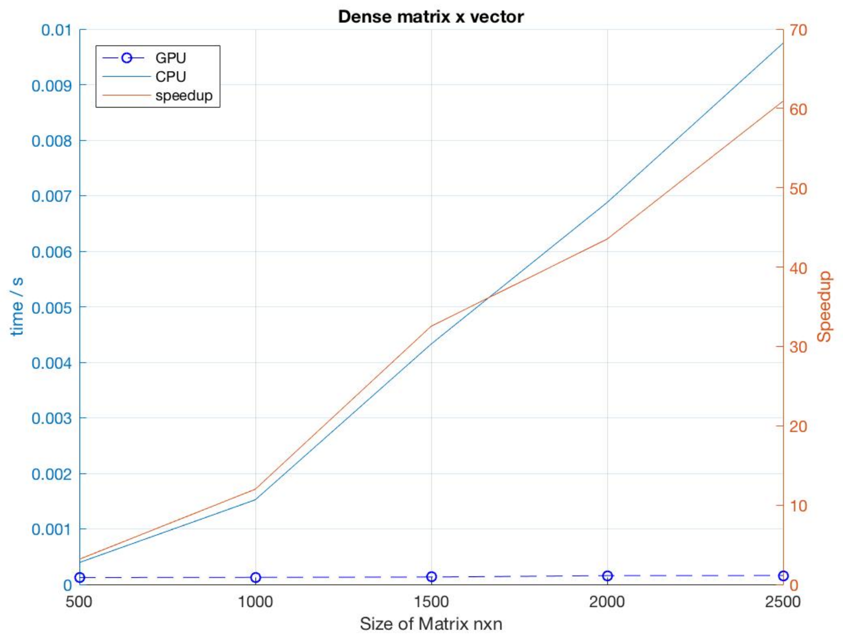
2.1.3 GPU implementation

The GPU utilises the data parallelism in matrix - vector multiplication.

```
we have m*n dimensional matrix
for i < m; i++;@tile(16, @outer, @inner)) // Work-group implicit loops
  for j < n; j++
    result_vector[i] += matrix[i*n+j]*vector[j]
```

An additional tile tag was introduced to facilitate kernel development due to many kernels only requiring the use of simple bounds and iteration strides. The tile tag, tiling for-loops as one and two dimensional sets of inner/outer loops. The tile(16) assign the working dimension. In this example it assign the working dimension is 16.

2.1.4 GPU vs CPU

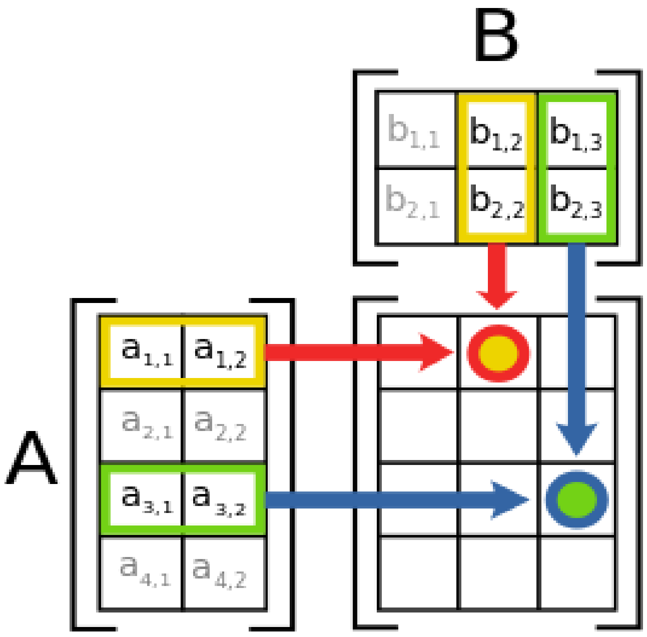


As we can see in figure GPU always faster than CPU. If matrix and vector size is bigger than 500.

2.2 Dense matrix multiplication

2.2.1 introduction

Matrix multiplication computes the product of two matrices. The theoretical operation show below, where matrix A and matrix B are the input matrices and produce the output matrix.



Let C is output matrix then mathematical formulation of the equation is

$C_{ij} = a_{i1}b_{1j} + \cdots + a_{im}b_{mj} = \sum_{k=1}^{m} a_{ik}b_{kj}$

for i = 1, ..., n and j = 1, ..., p.

That is, the entry C_{ij} of the product is obtained by multiplying term-by-term the entries of the i^{th} row of A and the j^{th} column of B and summing these m products. In other words, C_{ij} is the dot product of the i^{th} row of A and the j^{th} column of B.

2.2.2 CPU implementation

The CPU implementation is the most basic implementation. The pseudo code is below

```
we have  m*n and n*m dimensional matrices
for i < m; i++
    for j <m; j++
        for k < n;k++
            result[i][j] += A[i][k]*B[k][j]
```

In each iteration of the for loops, the code computes the product of two elements from matrix A and matrix B and add the product to the result from previous iteration of the loops. For example m by m square matrix to compute the element in result matrix, the CPU performs m multiply operations and n-1 sum operations. So the time complexity of entire multiplication there are $O(n^3)$ multiply operations and $O(n^2)$ sum operations.

2.2.3 GPU implementation

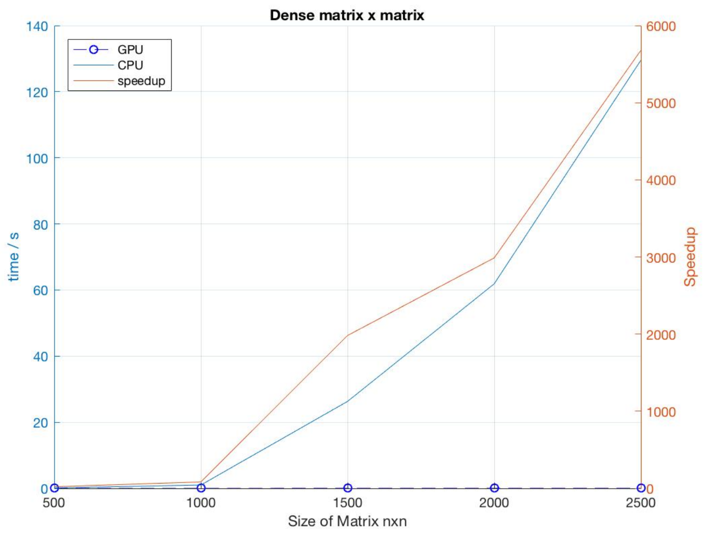
The GPU utilises the parallelism in matrix multiplication. The kernel launches n^2 thread blocks for an n by n square matrix multiplication. Each block does the multiplication of the row of matrix A and column of matrix B. Each thread block computes the product of one element of A and one element of matrix B.

We have pseudo code for okl as below

```
we have  m*n and n*m dimensional  matrices
for k < n; k++
  for o1 < m; o1+=16;@outer
    for o0 < m; o0+=16; @outer
      for y= o1; y <o1+16;y++;@inner
        for x = o0; x <o0+16;x++;@inner
          result[i][j] += A[y][k]*B[k][x]
```

The start, end and stride used in the outer and inner loops to support argument based variables and the working dimensions are resolved at run-time. Currently working dimensions must constant across on all the inner-loops defined in outer loop. In this example k will increment linearly but x and y will work in working dimensions. In this case our working dimensions is 16

2.2.4 GPU vs CPU



To test the performance of CPU and GPU. We take two array of size m-by-n and compute the matrices multiplication .We use the square matrices, the dimensions of the

matrices start from 500 by 500 to 2500 by 2500. The throughputs of the CPU and GPU as shown above. We use the semiology for plotting in Matlab. If size of matrices is small than the CPU perform better than GPU. But the large the size, the CPU gets poor performance.
The GPU memory transfer overhead has negative impact on the overall GPU performance. GPU memory transfer overhead cause the GPU overall throughput to be 10x lower than the GPU kernel overhead.

2.3 Dense matrix - matrix addition & subtraction

2.3.1 Introduction

We can perform addition or subtraction operations over 2 matrices. In this, we add element at position i*j in matrix A with element at the same position in matrix B. We can perform addition or subtraction operations only on same dimension matrices.
Let C is output matrix then mathematical formulation of the equation is
$C_{ij} = a_{ij} \pm b_{ij}$
for i = 1, ..., n and j = 1, ..., p.
That is the entry C_{ij} is the sum or subtracion of the i^{th} row and j^{th} column of A and the i^{th} row and j^{th} column of B.

$$\begin{pmatrix} 2 & 9 \\ 3 & 4 \\ -1 & 0 \end{pmatrix} + \begin{pmatrix} 3 & 2 \\ 4 & 5 \\ 2 & 6 \end{pmatrix} = \begin{pmatrix} 5 & 11 \\ 7 & 9 \\ 1 & 6 \end{pmatrix}$$

2.3.2 CPU implementation

The CPU implementation is easy for the addition or subtraction. We can use the same function for the both operations. The pseudo code is below

```
we have m*n dimension both matrices
if addition
  operation = 1
else
  operation = -1
for i < m; i++
    for j < n; j++
        result [i][j] = A[i][j]+(operations * B[i][j])
```

In each iteration the code will add one element of ij position of A matrix with one element of ij position of B matrix. For the subtraction we just multiply -1 with element of B matrix. And we save the result on same position in result matrix. The time complexity of this code is O($m * n$). If m is equal to n than we can say $O(n^2)$.

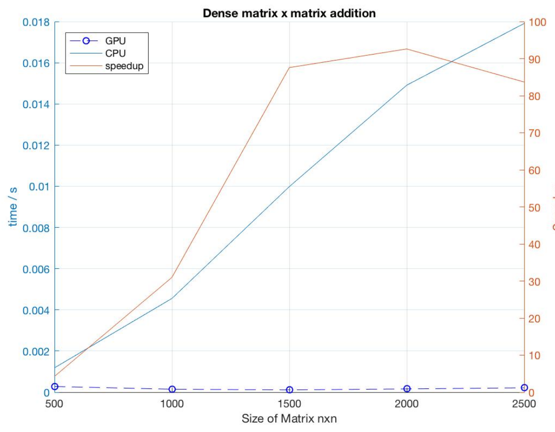
2.3.3 GPU implementation

A simple approach to compute addition or subtraction of two matrices on a GPU. The OKL pseudo code is below

```
we have m*n dimension both matrices
for o1 < m; o1+=16;@outer
    for o0 < m; o0+=16; @outer
        for y= o1; y <o1+16;y++;@inner
            for x = o0; x <o0+16;x++;@inner
                result[x][y] = A[x][y]+(operation * B[x][y])
```

The start, end and stride used in the outer and inner loops to support argument based variables and the working dimensions are resolved at run-time. Currently, working dimensions must constant across on all the inner-loops defined in outer loop. In this example x and y will work in working dimensions. In this case our working dimensions is 16. As we can see we are increment o0 and o1 by 16 and run x and y from o0 and o1 to o0+16 and o1+16. Which divide the work in group items.

2.3.4 GPU vs CPU



To test the performance of CPU and GPU. We take two array of size m-by-n and compute the addition of the matrices. We use the square matrices, the dimensions of the matrices start from 500 by 500 to 2500 by 2500. The throughputs of the CPU and GPU as shown above. We use the semiology for plotting in Matlab. If size of matrices is small than the CPU perform better than GPU. But the large the size, the CPU gets poor performance.

Chapter 3

Sparse Matrix

3.1 introduction

A sparse matrix or sparse array is a matrix in which most of the elements are zero. The number of zero-valued elements divided by total number of elements is called sparsity of the matrix. When storing and computing sparse matrix on device it is necessary to use the efficient algorithm and data structure. Because we can store sparse matrix in significantly in less storage.

3.1.1 Storing

A matrix is typically stored as a two dimensional or one dimensional but the length of one dimensional is equal to length of row of matrix * length of column of matrix. For m x n matrix, we require m * n * sizeof(float) to store the matrix.
But in case, Sparse matrix we need 3 vector which is equal to 3*(size of non-zero element in matrix)* sizeof(float).

3.1.2 Storing formats

- Dictionary of keys (DOK)
- List of lists (LIL)
- Coordinate list (COO)

- Compressed sparse row (CSR, CRS or Yale format)
- Compressed sparse column (CSC or CCS)

We are using Compressed sparse row (CSR, CRS or Yale format) for our implementation.

3.2 Sparse matrix - vector multiplication

3.2.1 introduction

Matrix vector multiplication has been implemented in the API for all kinds of matrices. We will focus on CSR formatted sparse matrix vector multiplication. An example CSR formatting with indexes starting from 1 would be the matrix

The matrix vector product using CRS format can be expressed in the usual way:
$y_i = \sum_{j=1}^{n} a_{ij} x_j$ i $\in [1...m]$

$$\begin{bmatrix} 1 & 0 & 0 & 3 \\ 0 & 2 & 4 & 6 \\ 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \end{bmatrix}$$

which would be represented by the vectors:
val =[1 3 2 4 6 5]
col =[1 4 2 3 4 2]
row =[1 3 6 6 7]

And we have vector
vector = [1 2 3 4]

We can perform multiplications between a matrix and vector when number of columns of matrix equal number of rows of vector

3.2.2 CPU implementation

The CPU implementation is the most basic implementation. The idea is below

```
for i < size of non_zero elements; i++
  result[row[i]] += non_zero[i] * vector[col[i]]
```

In each iteration of the for loop, the code computes the product of non-zero element of sparse matrix and position of column number of sparse matrix with same row number of vector element. And store the result on row number of non-zero element in sparse matrix. It is better for time and space complexity. The time complexity of this algorithm is O(size of non-zero elements).

3.2.3 GPU implementation

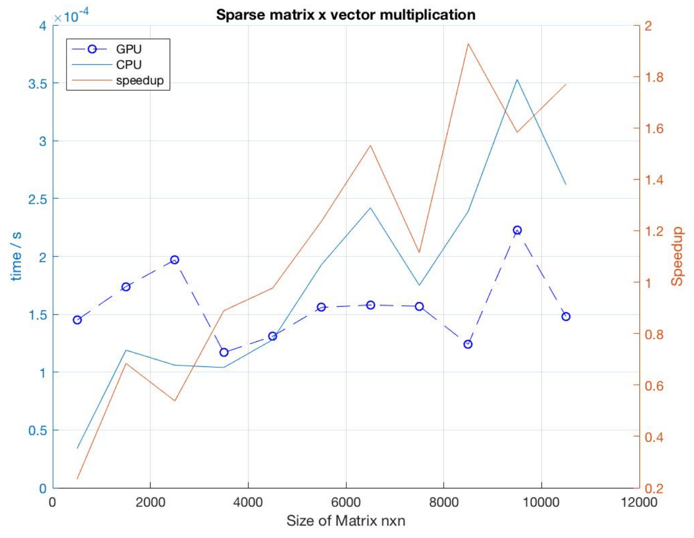
We implemented this algorithm in OKL as follow

```
for i=0; i < #row of matrix; i++; @tile(16, @outer, @inner)
  for j = row[i]; j <row[i+1];j++
    result[i] += non_zero[j] * vector[col[j]]
```

In this algorithm, (#row of matrix) is the number of row in matrix and row is a vector which save the number or non-zero elements in each row. For example, we have 2 non zero elements in row 0 then its row[0] = 0 and row[1] = 2. The most challenging is the sparse matrix-vector multiplication v = Au. Due to the coalescing restriction in accessing the GPU memory it is not efficient to use the standard compressed storage data format.

This algorithm runs number of rows multiply number of non zero element in the row. It store the element in result vector and in every iteration, it compute the product of non zero element and column number of non zero element position of vector element.

3.2.4 GPU vs CPU



To test the performance of CPU and GPU, we took a square sparse matrix and a vector. We save sparse matrix in 3 vectors in CSR format. Now we have 3 vector for sparse matrix, which stores non-zero elements of matrix, column number of non-zero elements in sparse matrix and row number of non-zero elements of the sparse matrix. For test cases, we use square sparse matrix. The dimension of the matrix starts from 500 by 500 to 10500 by 10500. We use the semiology for plotting in Matlab. The throughputs of the CPU and GPU as shown above. As we can see that the performance of CPU is better than GPU, if dimension of the matrix is less than 4000. But dimension bigger than 4000, CPU performance is start fall down. And GPU start perform better than CPU.

3.3 Sparse matrix multiplication

3.3.1 introduction

Matrix multiplication has been implemented in the API for all kinds of matrices. We will focus on both CSR formatted sparse matrix multiplication. An example of CSR formatting the sparse matrix as follow

Let C is output matrix then mathematical formulation of the equation is

$C_{ij} = a_{i1}b_{1j} + \cdots + a_{im}b_{mj} = \sum_{k=1}^{m} a_{ik}b_{kj}$

for i = 1, ..., n and j = 1, ..., p.

$$A=\begin{bmatrix}1&0&0&3\\0&2&4&6\\0&0&0&0\\0&5&0&0\end{bmatrix}\quad B=\begin{bmatrix}1&0&0&3\\0&2&4&6\\0&0&0&0\\0&5&0&0\end{bmatrix}$$

which would be represented by the vectors:

A_val =[1 3 2 4 6 5] B_val =[1 3 2 4 6 5]

A_col =[1 4 2 3 4 2] B_col =[1 4 2 3 4 2]

A_row =[0 0 1 1 1 3] B_row =[0 0 1 1 1 3]

We can perform multiplications between a matrix and matrix when number of columns of matrix equal number of rows of another matrix.

3.3.2 CPU implementation

The CPU implementation pseudo code is below.

```
we use 2 vector for save the result of multiplication
int m =0;
for j < # of A non-zero elements; j++
  for k < # of B non-zero elements; k++
    check condition a_col[j] == b_row[k]
      check if the position of matrix already in array return position s
        result[s] += A_val[j]*B_val[k]
      otherwise
        result[m] = A_val[j]*B_val[k]
        position array[m] = A_row[j]* #number of column in matrix B
          + B_col[k]
       m++
```

In CPU implementation, the j run from 0 to number of non-zero elements in A. Second loop k, start from 0 and end to number of the non-zero elements in B. It took j position of A_col and compare with all the B_row elements. Where A_col is the column number of non-zero elements and B_row is the row number of non-zero elements in matrix B. If we have same value on j position and k position than we have to multiply that elements and save it. After that we have to check if we already multiply and save the same row of matrix A and column of matrix B. If yes, then we have to add this in same element. If not, then we have to save result in new position of the result vector and position vector and increment the m. According to this procedure, we can save the memory because we do not need the matrix for saving the result.

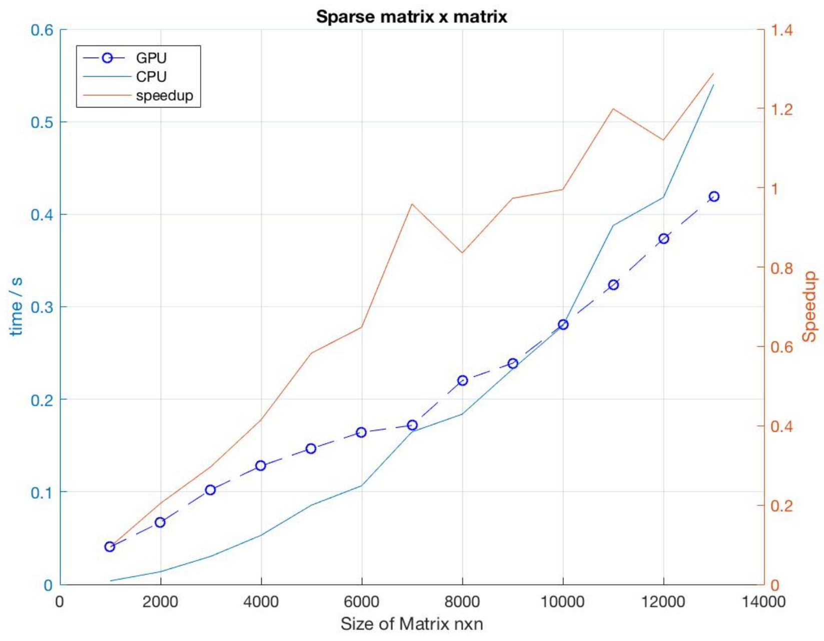
3.3.3 GPU implementation

The GPU implementation in OKL as follow

```
for j < # of A non−zero elements; j++
  for o0 < # of B non−zero elements; o0++; outer
    for k = o0 k < o0+16; k++
      check condition a_col[j] == b_row[k]
          result[A_row[j]* # column in
          matrix B + B_col[k]] = A_val[j]*B_val[k]
```

The GPU utilises the data parallelism in matrix multiplication. Except for the modified data structure for the sparse matrix storage the standard CRS matrix multiplication algorithm can be used with only minor modifications. In OKL, we just use for loops with an outer tag, which parallelised by threads (CPU) or work-groups (GPU). Another, we use inner tag in for-loops, which able to vectorized (CPU) or work concurrently (GPU). The biggest problem of sparse matrix is result storage. As you can see that we are using the full length matrix for storing the result of two sparse matrix. The problem is OCCA do not define atomic operations clearly. All the working group work individually.

3.3.4 GPU vs CPU



To test the performance of CPU and GPU, we took two square sparse matrix. We save sparse matrices in 6 vectors in CSR format. Now we have 6 vector for sparse matrices, which stores non-zero elements of matrix, column number of non-zero elements in sparse matrix and row number of non-zero elements of the sparse matrix. For test cases, we use square sparse matrices. The dimensions of the matrices start from 2000 by 2000 to 14000 by 14000. We use the semiology for plotting in Matlab. The throughputs of the CPU and GPU as shown above. As we can see that the performance of CPU is better than GPU, if dimension of the matrix is less than 10000. But dimension bigger than 10000, CPU performance is starting fall down. And GPU start perform better than CPU. CPU rise too fast after 12000. Because when we increase the matrix size it also increase the non-zero elements in matrix.

3.4 Sparse matrix - matrix addition & subtraction

3.4.1 introduction

Sparse matrix addition or subtraction is not too difficult. Firstly, we have to save both matrix in CSR formats. And we have matrices in this forms.
Let C is output matrix then mathematical formulation of the equation is

$C_{ij} = a_{ij} \pm b_{ij}$
for i = 1, ..., n and j = 1, ..., p.
That is, the entry C_{ij} is the sum or subtracion of the i^{th} row and j^{th} of A and the i^{th} row and j^{th} of B.

$$\begin{pmatrix} 2 & 9 \\ 3 & 4 \\ -1 & 0 \end{pmatrix} + \begin{pmatrix} 3 & 2 \\ 4 & 5 \\ 2 & 6 \end{pmatrix} = \begin{pmatrix} 5 & 11 \\ 7 & 9 \\ 1 & 6 \end{pmatrix}$$

A_val =[1 3 2 4 6 5]   B_val =[1 3 2 4 6 5]
A_col =[1 4 2 3 4 2]   B_col =[1 4 2 3 4 2]
A_row =[0 0 1 1 1 3]   B_row =[0 0 1 1 1 3]
Normally, we have to add j^{th} position of element of matrix A in j^{th} position of element of matrix B and save it on j^{th} position in result matrix.
We have to be same dimensional matrices for addition or subtraction of matrices.

3.4.2 CPU implementation

CPU implementation addition or subtraction operation for sparse matrix with CSR format as follow:

```
we have two vector of #non zero elements in A + #non zero elements in B
for i < #non zero elements in A; i++
  result[i] = A_non Zero [i]
  position[i] = A_row[i] * #number of column matrix A + A_col_number[i]
for i < #non zero elements in B; i++
  result[i+#non zero elements in A] = B_non Zero [i]
  position[i+#non zero elements in A] = B_row[i] * #number of column matrix
     B
     + B_col_number[i]
for i < #non zero elements in A + #non zero elements in B-1; i++
  for j < #non zero elements in A + #non zero elements in B; j++
    check condition i != j && i<j
    check condition position[i] == position[j]
      result[i] += result[j]
      result[j] = 0;
      position[j] = 0;
```

As we can see, we have two vectors for save the output result and position. In result vector, we save the addition or subtraction of matrix A elements with matrix B elements. Position vector, we save the position of the result element. In this algorithm we are firstly save all the non zero element of matrix A and after we save all the non zero element of matrix B. As same as, we save position of all the non zero elements A and B in vector position. And in the end we just walk through the result vector and position vector. When we have duplicate element in vector position, we add that position element of result vector to the duplicate position element and make it zero and also remove element from the position vector make it zero.

3.4.3 GPU implementation

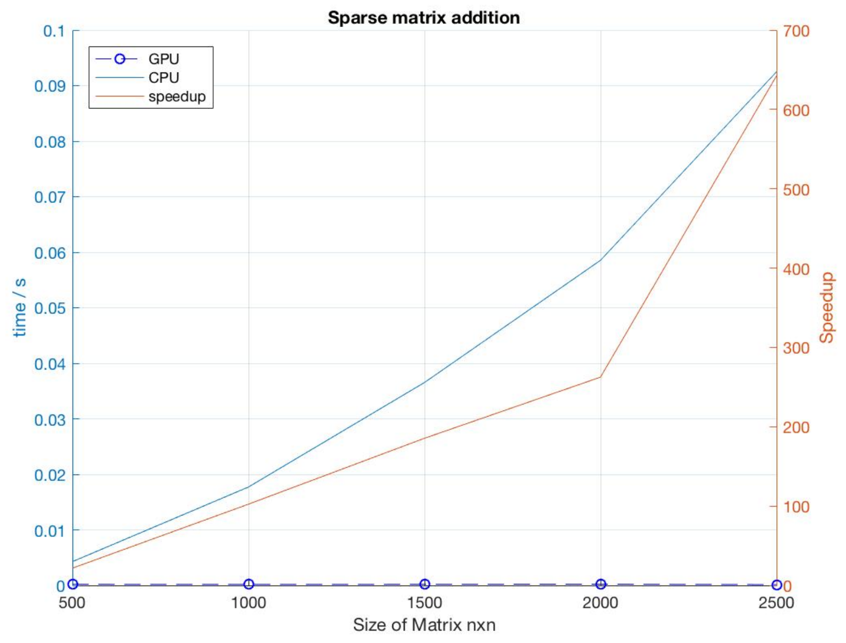
The GPU implementation for sparse matrix addition or subtraction pseudo is below:

```
we have two vector of #non zero elements in A + #non zero elements in B
for i < #non zero elements in A; i++;@tile(16, @outer, @inner)
  result[i] = A_non Zero [i]
  position[i] = A_row[i] * #number of column matrix A + A_col_number[i]
for i < #non zero elements in B; i++;@tile(16, @outer, @inner)
  result[i+#non zero elements in A] = B_non Zero [i]
  position[i+#non zero elements in A] = B_row[i] * #number of column matrix
    B
     + B_col_number[i]
for o1 < #non zero elements in A + #non zero elements in B−17;o1++;@outer
  for o0 < #non zero elements in A + #non zero elements in B;o0++;@outer
    for i = o1; i < o1+16;i++;@inner
  for j = o0; j < o1+16;j++;@inner
    check condition i !=j && i<j
    check condition position[i] == position[j]
      result[i] += result[j]
      result[j] = 0;
      position[j] = 0;
```

Both plus and minus matrices operations have been implemented, but since they share the same performance. I have only presented plus operation. While it’s called matrix addition, it can work for all data types, but it only applies to the values. That is, in the sparse, we only store values for given coordinates, but we can still add a sparse matrix to a sparse matrix.

In GPU implementation is similar to the CPU implementation, but in first two loops we add fourth clause tile in for-loop that indicating the type of parallelism to be take by the for-loop. After that next two loops we use the outer1 and outer0 in for-loops, which parallelised by threads (CPU) or working-groups (GPU). Afterward we have for-loops with the inner tag, which able to be vectorized (CPU) or or work concurrently (GPU). We have to check the condition i != j because it will add the same position value again and again. Now we have to check the duplicate element in position vector and add the same position result element. And make it zero in result vector and position vector.

3.4.4 GPU vs CPU



As we can be see, the performance of CPU is not very good. This is unfortunately as expected. We use the two sparse matrices and convert it in the CSR format. The dimension of the matrix starts from 500 by 500 to 2500 by 2500. We use the semiology for plotting in Matlab. The throughputs of the CPU and GPU as shown above. As we can see, the performance of GPU is better than CPU, if we choose matrices size bigger than 500 by 500.

Dot product

Introduction

The dot product or scalar product is an algebraic operation that takes two equal length sequence of numbers and return a single number.

The dot product of two vectors a = [a1, a2, ..., an] and b = [b1, b2, ..., bn] is defined as:

$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$

where $\sum$ denotes summation and n is the dimension of the vector space.

$$\begin{bmatrix} 1 & 3 & -5 \end{bmatrix} \begin{bmatrix} 4 \\ -2 \\ -1 \end{bmatrix} = 1 * 4 + 3 * (-2) + (-5) * (-1) = 3.$$

For dot product, we have to same length of both vectors.

CPU implementation

The CPU implementation is below:

```
    for i < size of vector;i++
      result += vector1[i] * vector2[i]

```

As we can see, It is simple to implement in CPU. We walk through the vector1 and vector2 and multiply the same position elements of both vectors and add result in previous result.

GPU implementation

Kernel implementation of dot product is below.

```
for (int group = 0; group < ((entries + block - 1) / block); ++group;
   @outer) {
        shared float s_vec[256];
        for (int item = 0; item < block; ++item; @inner) {
            if ((group * block + item) < entries) {
                s_vec[item] = vec[group * block + item] * vec2[group *
   block + item];
            } else {
                s_vec[item] = 0;
            }
        }
    for (int alive = ((block + 1) / 2); 0 < alive; alive /= 2) {
            for (int item = 0; item < block; ++item; @inner) {
                if (item < alive) {
                    s_vec[item] += s_vec[item + alive];
                }
            }
            for (int item = 0; item < block; ++item; @inner) {
                if (item == 0) {
                    blockSum[group] = s_vec[0];
                }
            }
        }
    }

```

We implement partial reduction of of vector using loop tiles of size block. In this case block is 256. As we can see firstly we save elements multiplication in shared memory. Now it will produce a vector of length block. Now in next loop it alive point to middle of the vector and item point 0 of vector. we add the elements of the vector and save it on position 0. In next loop, we just copy result to blocksum.

GPU vs CPU

As we can estimate GPU will more expensive than CPU. Because in GPU firstly we have to copy the multiplication to the shared memory. After that we start the partial reduction . And we know that memory transfer in GPU is more expensive than CPU. Another reason is that we implemented with partial reduction in GPU, but in CPU we are implement without reduction.

Multi-grid Method

Introduction

Multigrid (MG) methods in numerical analysis are algorithms for solving differential equations using a hierarchy of discretizations. They are an example of a class of techniques called multiresolution methods, very useful in problems exhibiting multiple scales of behavior.
The main idea of multigrid is to accelerate the convergence of a basic iterative method (known as relaxation, which generally reduces short-wavelength error) by a global correction of the fine grid solution approximation from time to time, accomplished by solving a coarse problem.

Solution Methods

To solve the systems of linear equations we are using this methods:
• Direct
– Gaussian elimination
• Iterative
– Jacobi

Gaussian elimination

Introduction

Gaussian elimination (also known as row reduction) is an algorithm for solving systems of linear equations. It is usually understood as a sequence of operations performed on the corresponding matrix of coefficients.
The fundamental idea is to add multiples of one equation to the others in order to eliminate a variable and to continue this process until only one variable is left. Once this final variable is determined, its value is substituted back into the other equations in order to evaluate the remaining unknowns. This method, characterized by step−by−step elimination of the variables, is called Gaussian elimination.

$$\begin{bmatrix} 1 & 3 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 3 \\ 5 \end{bmatrix}$$

$1x + 3y = 3$

$2x + 1y = 5$

Multiply row 1 by 2 and subtract from row 2

$1x + 3y = 3$

$5y = 1$

$x = 12/5$

$y = 1/5$

CPU implementation

The CPU implementation as below:

```
void Gauss_elmination_cpu(float a[], float d[], int n) {
int i, j, k, temp;
//********* Forward elimination process**************//
for (int i = 0; i < n - 1; i++) {
  for (int k = i + 1; k < n; k++) {
    float c = a[k * (n + 1) + i] / a[i * (n + 1) + i];
    for (int j = i; j <= n; j++) {
      a[k * (n + 1) + j] = a[k * (n + 1) + j] - (c * a[i * (n + 1) + j] )
  ;
```

```
      }
    }
  }
    //**************** Backward Substitution method****************//
  for (i = n - 1; i >= 0; i--){
    d[i] = a[i * (n + 1) + n];
    for (int j = i + 1; j < n; j++) {
      if (j != i) {
        d[i] = d[i] - a[i * (n + 1) + j] * d[j];
      }
    }
    d[i] = d[i] / a[i * (n + 1) + i];
  }
}
```

Forward elimination: reduction to row echelon form. Using it one can tell whether there are no solutions, or unique solution, or infinitely many solutions.
Back substitution: further reduction to reduced row echelon form.
In this code, We have i is the row number of matrix and k is walk through all the row which are below the i^{th} row. We have variable c which store the difference of i^{th} row, i^{th} column and k^{th} row, i^{th} column. And j walk through the element of i^{th} and k^{th} row. After finishing this method, We have upper triangular matrix.
Which we can solve by backward substitution method. We walk from down to up. We got the value and we use it for solve next value.

GPU implementation

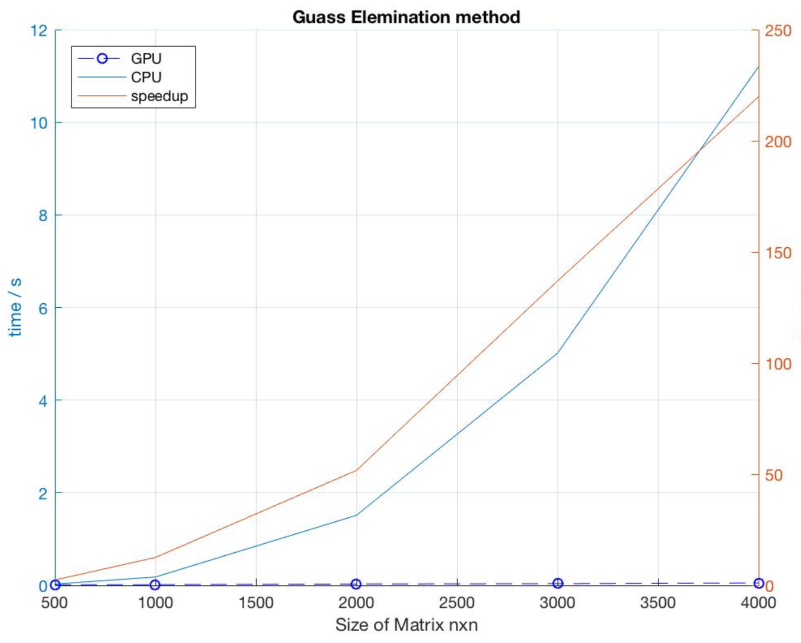
Kernel implementation of Guass Elimination is below.

```
    //********* Forward elimination process**************//
    for (int k =i+1; k<n; k++; @tile(16, @outer, @inner)) {
        float c = a[k*(n+1)+i]/a[i*(n+1)+i];
        for (int j=0; j<= n; j++) {
            a[k*(n+1)+j]=a[k*(n+1)+j]-(c*a[i*(n+1)+j]);
        }
    }
}
//**************** Backward Substitution method****************//
  for (int k =0; k < 1; k++; @tile(16, @outer, @inner)) {
```

```
            d[i] = a[i*(n+1)+n];
          for (int j =0; j <n; j++) {
                if (j > i) {
                    d[i] = d[i] - (a[i*(n+1)+j] * d[j]);
                }
            }
          d[i] = d[i]/a[i*(n+1)+i];
      }
}
```

In GPU, We implement the forward substitution and backward substitution method separately. Currently, working dimensions must constant across on all the inner-loops defined in outer loop (@tile(16, @outer, @inner)). In this example k will work in working dimensions. In this case our working dimensions is 16. But the implementation idea is same as CPU implementation.

GPU vs CPU



As we can see CPU will more expensive than GPU. We measure the length of matrix from 500X500 to 4000X4000. And GPU always take time near 0.xxx and CPU takes more time according to length of matrix. According to this conclusion, We can say that GPU is better than CPU.

Jacobi Method

Introduction

The Jacobi method is easily derived by examining each of the n equations in the linear system Ax = b in isolation. If in the i^{th} equation

$$\sum_{j=1}^{n} a_{i,j}x_j = b_i$$

We solve for the value of x_i while assuming the other entries of x remain fixed, We obtain

$$x_i = (b_i - \sum_{j\neq i} a_{i,j}x_j)/a_{i,i}$$

This suggests an iterative method defined by

$$x_i^{(k)} = (b_i - \sum_{j\neq i} a_{i,j}x_j^{(k-1)})/a_{i,i}$$

Which is the Jacobi method.

CPU implementation

The CPU implementation as below:

```
for (int k = 0; k < num_iter; k++) {
  for (int i = 0; i < n; i++ ) {
    float sum = 0;
    for (int j = 0; j < n; j++ ) {
      if ( j != i ) {
        sum += (a[i * (n + 1)+ j] * x[j]);
      }
    }
          x_new[i] = ((a[i * (n + 1) + n]) - sum ) / a[i + i * (n + 1)];
      }
  for (int i = 0; i < n; i++) {
    x[i] = x_new[i];
      }
}
```

This is Jacobi method, Where first loop is count the number of iteration and another loop i,j are start from 0 to n which is the size of the matrix nXn. x is the initial guess vector and x_new is result vector. And in the last loop, We copy the x_new to x. It run again and again till k is not equal or bigger than number of iteration.

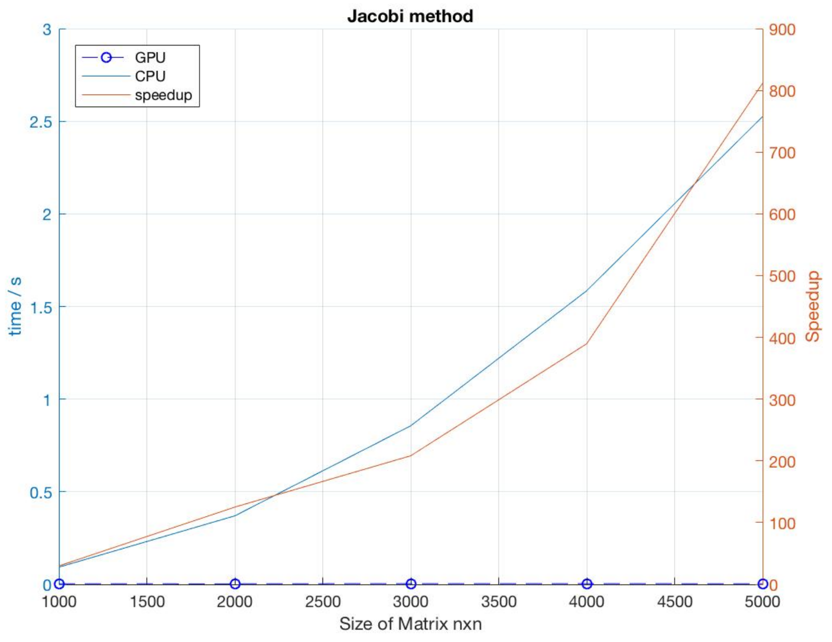
GPU implementation

The GPU implementation as below:

```
for (int i = 0; i < n; i++; @tile(16, @outer, @inner) ) {
  x_new[i] = a[i * (n + 1) + n];
  float sum =0;
  for (int j = 0; j < n; j++ ) {
    if ( j != i ) {
      sum += a[i + j * (n + 1)] * x[j];
          }
      }
       x_new[i] = ((a[i * (n + 1) + n]) - sum ) / a[i + i * (n + 1)];
      }
      for (int i = 0; i < n; i++; @tile(16, @outer, @inner){
      x[i] = x_new[i];
  }
```

This is GPU implementation of Jacobi method. It is same as CPU implementation but here we are using the fourth value for for loop. It is @tile(16, @outer, @inner), the tile tag, tiling for-loops as one and two dimensional sets of inner/outer loops. The tile(16) assign the working dimension. In this example it assign the working dimension is 16.

GPU vs CPU



For Jacobi method, GPU is better than CPU. We can compare according to our results. We compare the matrix size from 1000X1000 to 5000X5000. In this size matrix, GPU finish the program in 0.xxx time and CPU take time more than 0.2xx and it increase strictly according to size of the matrix.

Multi-grid mehtod Dense matrix

Introduction

In the multi-grid method, We use the Gauss elimination method, Jacobi method, matrix interpolation, matrix reduction, vector interpolation and call the multi-grid method recursive. The main idea is

```
multi-grid method(a, b, x, recursion)
  if (recursion == 0)
    gauss-elimination method(a, b, x)
    return
  jacobi method(a, b, x, 10)

  \\reduction of vector b
  b2h = Reduction (b - a*x)

  \\ initialize
  x2h = 0

  \\ compute a2h, R = reduction matrix, I = interpolation matrix
  a2h = R*  a * I

  multi-grid method(a2h, b2h, x2h, recursion -1)

  xh = interpolation x2h

  x = x + xh

  jacobi method(a, b, x, 10)
```

With the reduction, We decrease the size of the vector. For example, We have a vector of size n after reduction, We have a new vector of size n/2. We call the multi-grid method again with the half size of a, b and x. And we repeat this process till the recursion will be 0 and when it will be 0, It calls to Gauss elimination method and stops the recursive loop.

CPU implementation

The CPU implementation as below:

```
 if (recursion == 0) {

      Gauss_elmination_cpu(a, b, x, row);
      return ;
  }
  float *x_new2 = new float[row];

  init_zero(x_new2, row);
  jacobi_method_cpu(a, x, b, x_new2, row, alpha);

  float *b2h = new float[row / 2];
  float *res1 = new float[row];
  float *x_new2h = new float[row];
  float *a2h = new float[(row / 2) * (row / 2)];

  init_zero(b2h, row / 2);
  init_zero(res1, row);
  init_zero(x_new2h, row);
  init_zero(a2h, (row / 2) * (row / 2));

  matrix_x_vector(row, row, x, a, x_new2h);

  add_sub_vector(b, x_new2h, res1, row, -1);

  reduction_vector(res1, (row / 2), b2h);

  init_zero(x_new2h, row);

  interpolation_reduction_matrix(a, row, a2h);
```

```

    if (row / 2 <= 0) {
        cout << "error" << endl;
        return;
    }
    multigrid_method(a2h, x_new2h, b2h, recursion - 1, row / 2, alpha);

    float * res_int = new float[(row * 2) + 1];
    init_zero(res_int, (row * 2) + 1);

    reduction_interpolation_vector(x_new2h, row, res_int);
    add_sub_vector(x, res_int, x, row, 1);

    jacobi_method_cpu(a, x, b, x_new2, row, alpha);
```

In CPU, It is the same implementation as I describe above. Firstly, We check recursion is 0 or not. If yes, we call the Gauss elimination method and stop the recursion. If not, We call to Jacobi method alpha times. We calculate the b_h^{2h},x_h^{2h} and a_h^{2h}. Size of x_h^{2h} is half of size x_{2h}^{h}. And we call multigrid method recusively with the b_h^{2h},x_h^{2h} and a_h^{2h}. And subtract 1 from recursion. After recursion call, We convert x_h^{2h} to x_{2h}^{h} and add in x. In last, We call jacabi method again alpha time.

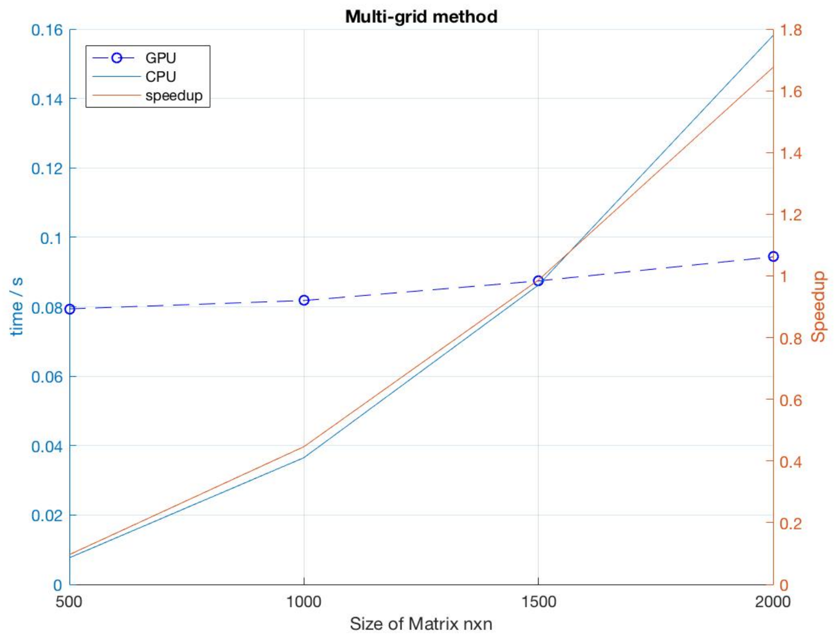
GPU implemintation

The GPU implementation as below:

```
  if (recursion == 0) {
        gauss_elmination_call_gpu(row, o_a, o_b, o_x, device);
        return;
    }
    occa::memory o_d, o_b2h, o_x2h, o_a2h, o_res, o_res_result2h;
    // Allocate memory on the device

    o_d   = device.malloc(row * sizeof(float));
    o_b2h  = device.malloc((row / 2) * sizeof(float));
    o_x2h  = device.malloc(row * sizeof(float));
    o_res  = device.malloc(row * sizeof(float));
    o_a2h  = device.malloc((row / 2) * (row / 2) * sizeof(float));
    o_res_result2h  = device.malloc(((row * 2) + 1) * sizeof(float));

    jacobi_method_call_gpu(row, o_a, o_b, o_x, o_d, device, alpha);

    dense_Matrix_Vector_Multiplication_call_gpu(row, o_a, o_x, o_res,
    device);

    add_sub_call_gpu(row, o_b, o_res, o_res, device, -1);

    relaxation_reduction_vector(row / 2, o_res, o_b2h, device);

    reduction_interpolation_reduction_matrix_call_gpu(row, o_a, o_a2h,
    device);

    if (row / 2 <= 0) {
        cout << "error" << endl;
        return;
    }

    multigrid_method_gpu(row / 2, o_a2h, o_b2h, o_x2h, device, recursion -
    1, alpha);

    relaxation_interpolation_vector_call_gpu(row, o_x2h, o_res_result2h,
    device);

    add_sub_call_gpu(row, o_x, o_res_result2h, o_x, device, 1);

    jacobi_method_call_gpu(row, o_a, o_b, o_x, o_d, device, alpha);
```

It has same idea like CPU implementation, We just make all calculation on GPU. And we described before Guass elimination and jacobi method in GPU.

GPU vs CPU



We can see in this graph if matrix size is smaller than 1500 than CPU is better than GPU. And according to size of matrix, the CPU time increase faster also. But GPU is increase very slightly. In my opinion, GPU is much better if we have matrix size is bigger than 1500 X 1500 than GPU will be much faster than CPU.

Multi-grid mehtod sparse matrix

CPU implementation

The CPU implementation as below:

```
void multigrid_method_sparse_matrix(float a_non_zero[], int a_col_number
  [], int a_row[], float x[], float b[], int recursion, int row, int
  alpha, int size_a) {

   if (recursion == 0 || row / 2 < 3 ) {
       int *point = new int[size_a];
       float *aa  = new float [row * row];
       for (int i = 0; i < size_a; i++) {
           point[i] = a_col_number[i] * row + a_row[i];
       }
```

```

        vectorToMatrix(row, row, size_a, a_non_zero, aa, point);

        Gauss_elmination_cpu(aa, b, x, row);

        delete [] point;
        delete [] aa;

        return ;
    }


    float *x_new2 = new float[row];
    init_zero(x_new2, row);

    jacobi_method_cpu_sparse_matrix(a_non_zero, a_col_number, a_row, b, x,
    x_new2, row, alpha, size_a);


    float *b2h = new float[row / 2];
    float *res1 = new float[row];
    float *x_new2h = new float[row];

    init_zero(b2h, row / 2);
    init_zero(res1, row);
    init_zero(x_new2h, row);

    sparse_matrix_x_vector(row, size_a, x, a_row, a_col_number, a_non_zero,
     x_new2h);
    add_sub_vector(b, x_new2h, res1, row, -1);
    reduction_vector_sparse(res1, row, b2h);

    init_zero(x_new2h, row);

    int size_non = (size_a + row) * 3;

    float *a2h = new float[size_non];
    int *a2h_row = new int[size_non];
    int *a2h_col = new int[size_non];

    init_zero(a2h, size_non);
    init_zero(a2h_row, size_non);
    init_zero(a2h_col, size_non);


    size_non = interpolation_reduction_matrix_sparse_matrix(a_non_zero,
   a_col_number, a_row, size_a, row, a2h, a2h_row, a2h_col);

    multigrid_method_sparse_matrix(a2h, a2h_col, a2h_row, x_new2h, b2h,
   recursion - 1, row / 2, alpha, size_non);

    float * res_int = new float[(row * 2) + 1];

    init_zero(res_int, (row * 2) + 1);

    reduction_interpolation_vector(x_new2h, row, res_int);

    add_sub_vector(x, res_int, x, row,  1);

    init_zero(x_new2, row);

    jacobi_method_cpu_sparse_matrix(a_non_zero, a_col_number, a_row, b, x,
   x_new2, row, alpha, size_a);


    delete [] x_new2h;
    delete [] b2h;
    delete [] res1;
    delete [] res_int;
    delete [] a2h;
    delete [] a2h_col;
    delete [] a2h_row;
}
```

In CPU, It is the same implementation as I describe above. The difference is use of CSR format matrix format rather than use dense matrix with most values 0. Firstly, We check recursion is 0 or not. If yes, we call the Gauss elimination method and stop the recursion. If not, We call to Jacobi method alpha times. We calculate the b_h^{2h},x_h^{2h} and a_h^{2h}. Size of x_h^{2h} is half of size x_{2h}^{h}. And we call multi-grid method recusively with the b_h^{2h},x_h^{2h} and a_h^{2h}. And subtract 1 from recursion. After recursion call, We convert x_h^{2h} to x_{2h}^{h} and add in x. In last, We call jacabi method again alpha time.

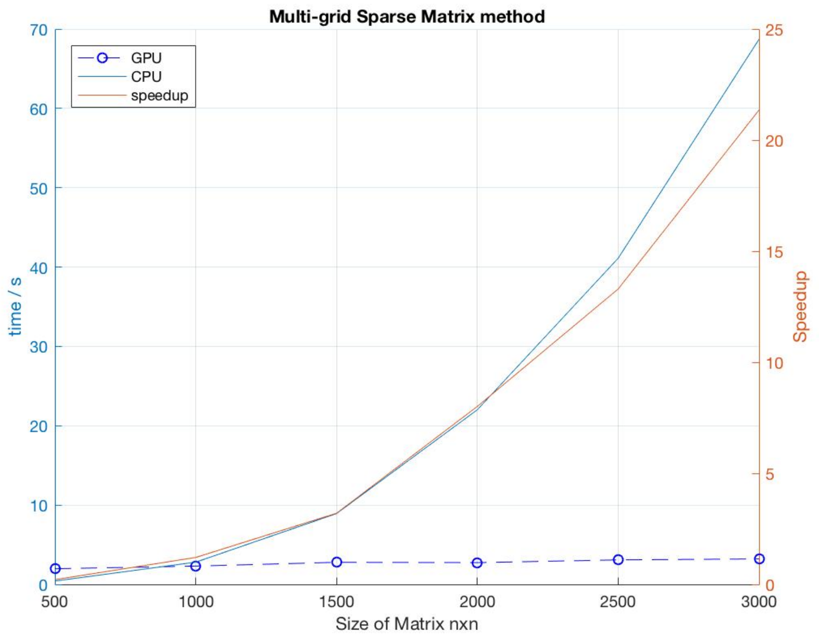
GPU implementation

The GPU implementation as below:

```
void multigrid_method_gpu_sparse_matrix(int row, occa::memory o_a, occa::memory o_a_row, occa::memory o_a_col, occa::memory o_b, occa::memory o_x, occa::device device, int recursion, int alpha, int size_a) {
    if (recursion == 0 || row / 2 <= 3 || size_a < row ) {
        occa::memory o_aa;

        o_aa = device.malloc((row * row) * sizeof(float));

        sparse_vector_to_matrix_gpu_call(row, o_a, o_a_row, o_a_col, device, size_a, o_aa);

        gauss_elmination_call_gpu(row, o_aa, o_b, o_x, device);

        return;
    }

    occa::memory o_b2h, o_x2h, o_a2h, o_a2h_row, o_a2h_col, o_res, o_res2, o_res_result2h, o_row_number ;
    // Allocate memory on the device

    o_b2h  = device.malloc((row / 2) * sizeof(float));
    o_x2h  = device.malloc((row / 2) * sizeof(float));
    o_res  = device.malloc(row * sizeof(float));
    o_res2  = device.malloc(row * sizeof(float));

    int size_non = (size_a + row);
    o_a2h  = device.malloc(size_non * sizeof(float));
    o_a2h_row  = device.malloc(size_non * sizeof(int));
    o_a2h_col  = device.malloc(size_non * sizeof(int));


    jacobi_method_call_gpu_sparse_matrix(row, o_a, o_a_col, o_a_row, o_b, o_x, device,  alpha, size_a);

    sparse_Matrix_Vector_Multiplication_call_gpu(row, size_a, o_a, o_a_row, o_a_col, o_x, o_res, device);

    add_sub_call_gpu(row, o_b, o_res, o_res2, device, -1);

    relaxation_reduction_vector(row / 2, o_res2, o_b2h, device);

    size_non = reduction_interpolation_reduction_sparse_matrix_call_gpu(
row, o_a, o_a_col, o_a_row, o_a2h, o_a2h_row, o_a2h_col, device,
size_a);


    multigrid_method_gpu_sparse_matrix(row / 2, o_a2h, o_a2h_row, o_a2h_col
, o_b2h, o_x2h, device, recursion - 1, alpha, size_non);

    o_res_result2h = device.malloc(((row * 2) + 1) * sizeof(float));

    relaxation_interpolation_vector_call_gpu(row, o_x2h, o_res_result2h,
device);

    add_sub_call_gpu(row, o_x, o_res_result2h, o_x, device, 1);

    jacobi_method_call_gpu_sparse_matrix(row, o_a, o_a_col, o_a_row, o_b,
o_x, device, alpha, size_a);
}
```

It has same idea like CPU implementation, We just make all calculation on GPU. And we described before Guass elimination and jacobi method in GPU. And, We use the CSR matrix format rather than full matrix size and Gauss elimination method is work only with full matix size. Thats why, We change CSR format matrix to sparse matrix with 0's and than call to Gauss elimination method.

GPU vs CPU



We can see in this graph if matrix size is smaller than 800 than CPU is better than GPU. And according to size of matrix, the CPU time increase faster also. But GPU is increase very slightly. In my opinion, GPU is much better if we have matrix size is bigger than 800 X 800 than GPU will be much faster than CPU.

Conclusion

This thesis has shown the performance of the GPU implementation of dense matrix-vector multiplication, dense matrix multiplication, dense matrix addition and subtraction, sparse matrix-vector multiplication, sparse matrix multiplication, sparse matrix addition and subtraction, dot product of two vectors, Gauss-Elimination, Jacobi method, matrix reduction and interpolation, vector interpolation and reduction and Multi-grid method for dense matrix and sparse matrix. The performance of the implementations shows that GPU is faster than CPU. In all the cases, the results recommend using the GPU instead of the CPU to get better performance when input size is large. Experiments in this thesis show that the CPU may have better performance than the GPU when input data size is small though the application has good instruction-level parallelism or data parallelism. GPU as a processor with high inherent parallelism and combining it into clusters opens new opportunities for low budget parallel computing.
OCCA is easy to use and understand. As we see kernel it look like CPU implementation. It include a portable solution for current and future architecture, an ease for the development of parallel codes, and the ability to expose as much as parallelism as possible for achieving optimal performance. A macro-based approach using the preprocessor for source-to-source transformation. An implementation of this model resulted in the OCCA intimidate representation, which supports serial code ,Pthreads, OpenMP, CUDA, OpenCL, and COI. Constructing the OCCA IR used a generalisation of current parallel architectures to unify the different languages and standards for heterogeneous computing.OCCA provides evidence that it is possible to achieve high performance across multiple platforms. With the use of just-in-time compilation and the custom macro-based language, OCCA enables the use of different programming languages on multiple architectures.
In my point of view, OCCA is good to use because it is minimal extensions to C, who

is familiar with regular languages. Because as we seen before it is using for-loops with fourth clause tile or outer or inner. It is easy to understand.

The problem of OCCA, it is not clarify atomic operations. I got difficulty, when i try to save the output of sparse matrix multiplication in CSR format. Because they are not talking anything about atomic operations. We have shared or exclusive memory operations.

Currently, they are working on loop-carried dependency analysis for kernel generation + testing and support offset in kernel calls. They are developing tiling loop labels for example tile(x,y) and possibly loop-collapsing.

Bibliography

1. LaTeX Template
2. A Multigrid Tutorial, 2nd Edition
3. OCCA
4. OKL: A Unified Language for Parallel Architectures
5. OCCA: An Extensible Portability Layer for Many-Core Programming
6. CRS Matrix-Vector Product
7. Compressed Row Storage (CRS)
8. Sparse matrix
9. Dot product
10. Density matrix
11. OCCA: A unified approach to multi-threading languages