

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 11, 2018



# Contents

<b>Abstract</b>	<b>4</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Multi-grid Method . . . . .	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: Unified Approach To Multithreading Languages . . . . .	4
1.2.1 History . . . . .	4
1.2.2 Overview . . . . .	5
1.2.3 Device . . . . .	5
1.2.4 Setup device . . . . .	6
1.2.5 Memory management . . . . .	6
1.2.6 kernel . . . . .	7
<b>2 Dense Matrix</b>	<b>9</b>
2.1 Dense matrix - vector multiplication . . . . .	9
2.1.1 Introduction . . . . .	9
2.1.2 C++ implementation . . . . .	10
2.1.3 OCCA implementation . . . . .	10
2.1.4 OCCA vs CPU . . . . .	11
2.2 Dense matrix-matrix multiplication . . . . .	11
2.2.1 Introduction . . . . .	11
2.2.2 C++ implementation . . . . .	12
2.2.3 OCCA implementation . . . . .	13
2.2.4 OCCA vs CPU . . . . .	14
2.3 Dense matrix - matrix addition & subtraction . . . . .	14
2.3.1 Introduction . . . . .	14
2.3.2 C++ implementation . . . . .	15
2.3.3 OCCA implementation . . . . .	15
2.3.4 OCCA vs CPU . . . . .	16
<b>3 Sparse Matrix</b>	<b>17</b>
3.1 Introduction . . . . .	17
3.1.1 Storing . . . . .	17

3.1.2	Storing formats . . . . .	17
3.2	Compressed Row Storage (CRS) . . . . .	18
3.3	Sparse matrix - vector multiplication . . . . .	19
3.3.1	Introduction . . . . .	19
3.3.2	C++ implementation . . . . .	20
3.3.3	OCCA implementation . . . . .	20
3.3.4	OCCA vs CPU . . . . .	21
3.4	Sparse matrix multiplication . . . . .	22
3.4.1	Introduction . . . . .	22
3.4.2	C++ implementation . . . . .	22
3.4.3	OCCA implementation . . . . .	23
3.4.4	OCCA vs CPU . . . . .	24
3.5	Sparse matrix - matrix addition & subtraction . . . . .	25
3.5.1	Introduction . . . . .	25
3.5.2	CPU implementation . . . . .	25
3.5.3	OCCA implementation . . . . .	26
3.5.4	OCCA vs CPU . . . . .	27
<b>4</b>	<b>Dot product</b>	<b>29</b>
4.1	Introduction . . . . .	29
4.2	CPU implementation . . . . .	29
4.3	OCCA implementation / Reduction . . . . .	30
4.4	GPU vs CPU . . . . .	31
<b>5</b>	<b>Gaussian elimination</b>	<b>32</b>
5.1	Introduction . . . . .	32
5.2	C++ implementation . . . . .	32
5.3	OCCA implementation . . . . .	33
5.4	OCCA vs CPU . . . . .	35
<b>6</b>	<b>Jacobi method</b>	<b>36</b>
6.1	Introduction . . . . .	36
6.2	C++ implementation . . . . .	36
6.3	OCCA implementation . . . . .	37
6.4	OCCA vs CPU . . . . .	38
<b>7</b>	<b>Multigrid Method</b>	<b>39</b>
7.1	Introduction . . . . .	39
7.2	Multigrid pseudo-code . . . . .	39
7.3	Multigrid mehtod Dense matrix . . . . .	40
7.3.1	Introduction . . . . .	40
7.3.2	C++ implementation . . . . .	41
7.3.3	OCCA implemintation . . . . .	42
7.3.4	OCCA vs CPU . . . . .	44
7.3.5	Numerical analysis of results . . . . .	44
7.4	Multigrid mehtod sparse matrix . . . . .	45
7.4.1	C++ implementation . . . . .	45
7.4.2	OCCA implementation . . . . .	47

7.4.3	OCCA vs CPU . . . . .	49
7.4.4	Numerical analysis of results . . . . .	50

<b>Conclusion</b>	<b>51</b>
-------------------	-----------

<b>Bibliography</b>	<b>53</b>
---------------------	-----------

# 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 OCCA 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 OCCA and CPU.

# Chapter 1

## Introduction

In this chapter, We will explain the multi-grid definition and OCCA components. We are using CSR format for sparse matrix. We will explain also, What is CSR format ? and how can we save memory with using CSR format ?

### 1.1 Multi-grid Method

#### 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:

$$-u''(x) + \alpha u(x) = f(x) \quad 0 < c < 1, \alpha > 0 \quad (1.1)$$

$$u(0) = u(1) = 0 \quad (1.2)$$

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  $v = (v_1, \dots, v_{n-1})^T$ , whose components satisfy the  $n-1$  linear equations

Defining  $f = (f(x_1), \dots, f(x_{n-1}))^T = (f_1, \dots, f_{n-1})^T$ , the vector of right-side values, we may also represent this system of linear equations in matrix form as

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

or even more compactly as  $Av = f$ . The matrix  $A$  is  $(n-1) \times (n-1)$ , 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  $Av=f$

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

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):

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

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: Unified Approach To Multithreading Languages

### 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. From [4](#)

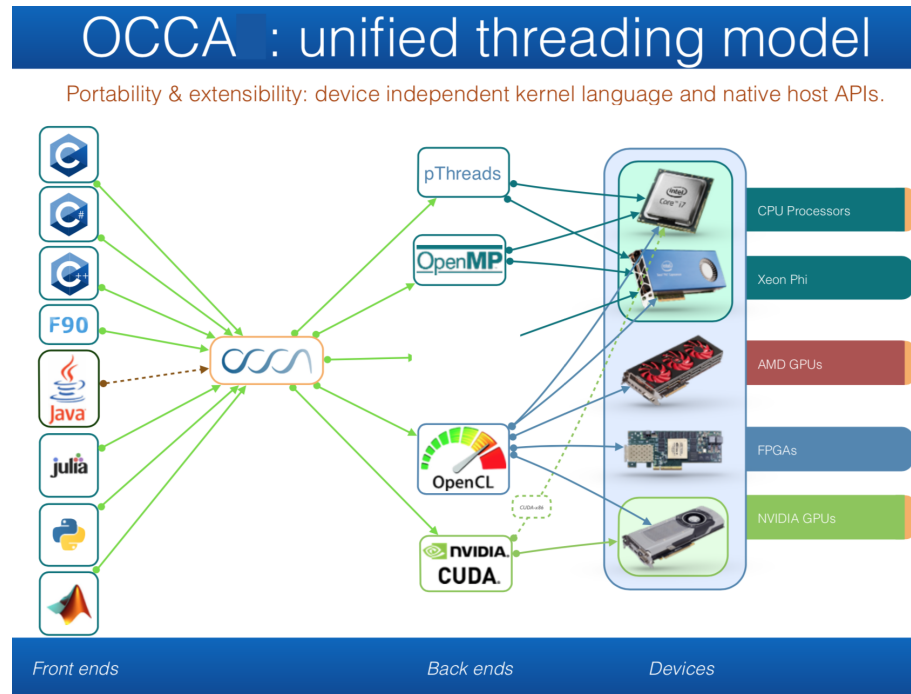


FIGURE 1.1: Current relationship between supported frontends, the OCCA languages, and supported backends

### 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. From [4](#)

### 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.

### 1.2.4 Setup device

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. From [4](#)

```
"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"
```

### 1.2.5 Memory management

We can not allocate runtime 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 OCCA we can allocate memory, with the following simples lines of codes.

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

LISTING 1.1: allocating memory to the device

In code Listing 1.1 line 2 and 3, We are allocating memory of size entries with type float and copy a and b to device. But in line 5, We are just allocate memory. Because we allocate this memory for our result and we are not copying anything to it.

### 1.2.6 kernel

Okl (Programming language use by OCCA) 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

```

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

```

LISTING 1.2: call to kernel function

Above Listing 1.2, We are using function buildKernel for building kernel at runtime. In listing 1.2, 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.

#### Parallelism

OpenCL and CUDA are always separate from the kernel source, but maintain a connection through the working dimension used in a kernel execution. OCCA extends the C for-loops by adding a fourth statement (@outer/ @inner). Aside from showing the explicit loops, OCCA now has the feature to automatically assign working dimensions to the offload model through the outer and inner loops.

```

1  for (int i = 0; i < 3; ++i; @outer) {
2      work(i);
3  }
4  //should be expected to work regardless of the iteration ordering, such as:
5  work(0);
6  work(1);
7  work(2);
8  // and also
9  work(2);
10 work(0);
11 work(1);

```

---

LISTING 1.3: outer loop example

Since the execution order can be non-deterministic and in parallel, there shouldn't be any loop dependencies.

Distributing work between outer and inner loops is heavily dependent on the device architecture. Try aiming for a power of 2 size for inner-loop sizes to make use of vectorization. From [3](#)

## Chapter 2

# Dense Matrix

In this chapter, We will discuss implementation about dense matrix. Dense matrix is a matrix, Which have mostly element are non zero. We will implement matrix- vector, matrix multiplication and matrix addition and subtraction in CPU and OCCA.

### 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  $m \times n$  dimensional matrix can be multiplied only  $n$  dimensional vector. The theoretical operation is shown below, matrix-vector multiplication output is  $m \times 1$  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:

$$y_i = \sum_{j=1}^n a_{ij}x_j \quad i \in [1...m] \quad (2.1)$$

The basic idea with visualisation as below:

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

FIGURE 2.1: matrix vector multiplication

### 2.1.2 C++ implementation

The C++ implementation is the most basic implementation. The idea is below

```

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

LISTING 2.1: matrix vector multiplication in CPU/C++

Listing 2.1 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 OCCA implementation

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

```

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

LISTING 2.2: matrix vector multiplication in OCCA

In Listing 2.2, 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 OCCA vs CPU

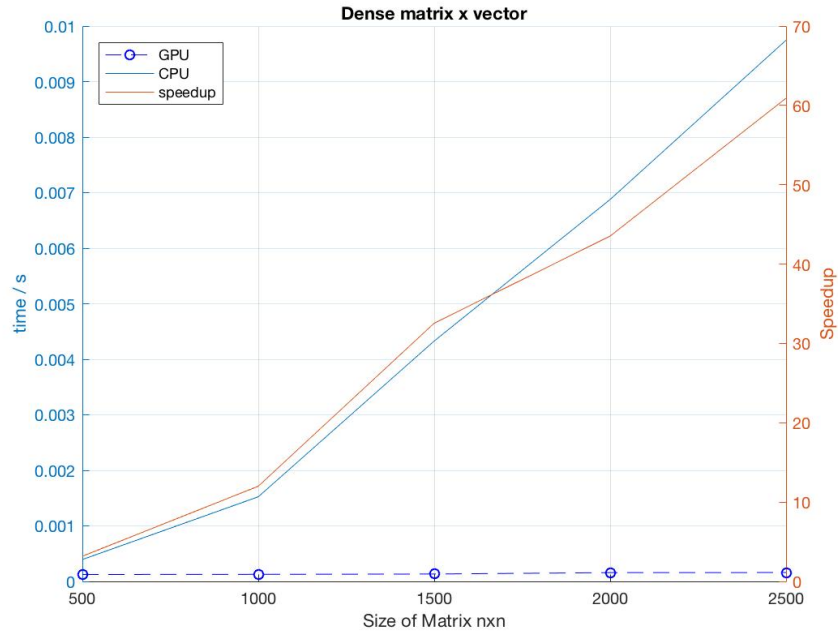


FIGURE 2.2: compare OCCA vs CPU performance

As we can see in figure 2.2 OCCA always faster than CPU. If matrix and vector size is bigger than OCCA perform better than CPU.

## 2.2 Dense matrix-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.



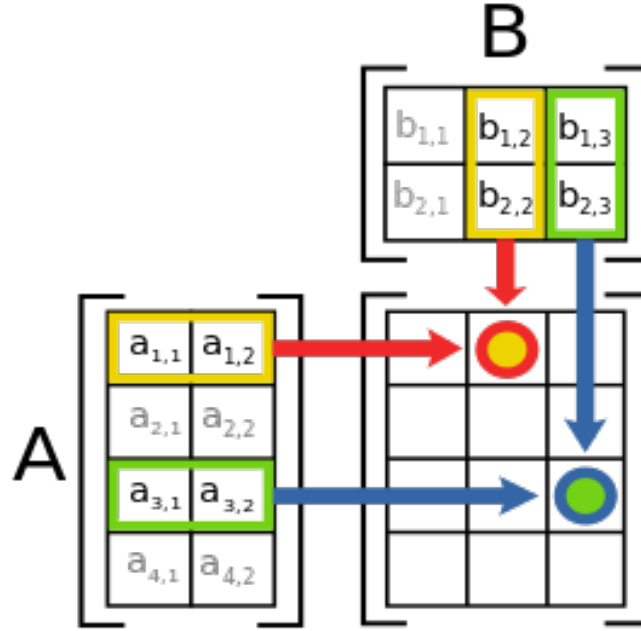


FIGURE 2.3: matrix multiplication visualisation

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

$$C_{ij} = a_{i1}b_{1j} + \cdots + a_{im}b_{mj} = \sum_{k=1}^m a_{ik}b_{kj} \quad (2.2)$$

for  $i = 1, \dots, n$  and  $j = 1, \dots, p$ .

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 C++ implementation

The C++ implementation is the most basic implementation. The pseudo code is below

```

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

LISTING 2.3: matrix multiplication in CPU/C++

In Listing 2.3, 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 OCCA implementation

The OCCA 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

```

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

LISTING 2.4: matrix multiplication in OCCA

Listing 2.4, 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. We explain further about inner loop in chapter 1 [1.2.6](#).

### 2.2.4 OCCA vs CPU

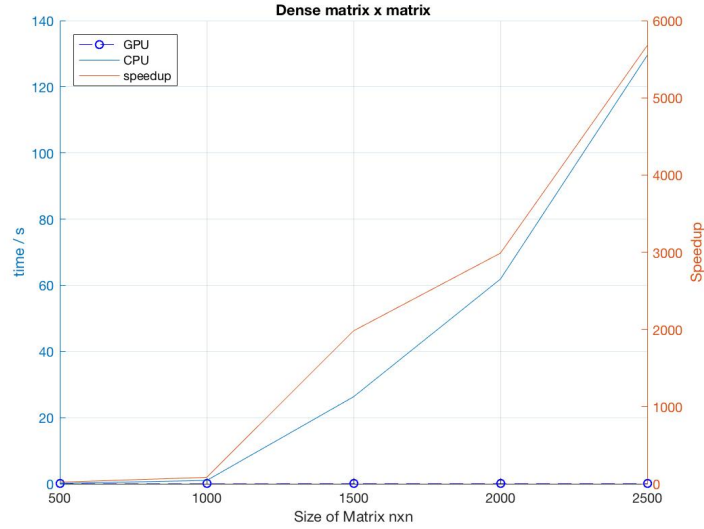


FIGURE 2.4: comparison OCCA vs CPU performance

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 2.4 . 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

$$C_{ij} = a_{ij} \pm b_{ij} \quad (2.3)$$

for  $i = 1, \dots, n$  and  $j = 1, \dots, p$ .

That is the entry  $C_{ij}$  is the sum or subtraction of the  $i^{th}$  row and  $j^{th}$  column of A and the  $i^{th}$  row and  $j^{th}$  column of B.

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

FIGURE 2.5: matrix addition

### 2.3.2 C++ implementation

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

```

1  we have m*n dimension both matrices
2  if addition
3      operation = 1
4  else
5      operation = -1
6  for i < m; i++
7      for j < n; j++
8          result [i][j] = A[i][j] + (operation * B[i][j])

```

LISTING 2.5: matrix addition or subtraction in C++

Listing 2.5, 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 OCCA implementation

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

```

1  we have m*n dimension both matrices
2  for o1 < m; o1+=16;@outer
3      for o0 < m; o0+=16; @outer
4          for y= o1; y <o1+16;y++;@inner
5              for x = o0; x <o0+16;x++;@inner

```

6

```
result[x][y] = A[x][y] + (operation * B[x][y])
```

LISTING 2.6: matrix addition or subtraction in OCCA

Listing 2.6, 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 OCCA vs CPU

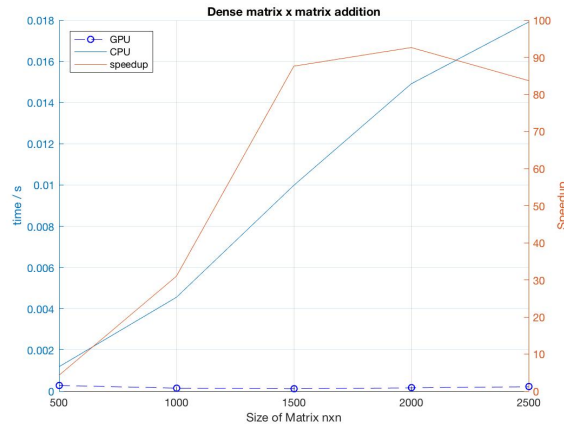


FIGURE 2.6: comparison OCCA vs CPU performance

To test the performance of CPU and OCCA. 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 OCCA as shown above. We use the semiology for plotting in Matlab. If size of matrices is small than the CPU perform better than OCCA. 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  $m \times n$  matrix, we require  $(m * n * \text{sizeof(float)})$  to store the matrix.

But in case, Sparse matrix we need 3 vector which is equal to  $(3 * (\text{size of non-zero element in matrix}) * \text{sizeof(float)})$ .

#### 3.1.2 Storing formats

- Dictionary of keys (DOK)

DOK consists of a dictionary that maps (row, column)-pairs to the value of the elements. Elements that are missing from the dictionary are taken to be zero. The format is good for incrementally constructing a sparse matrix in random order, but poor for iterating over non-zero values in lexicographical order. One typically constructs a matrix in this format and then converts to another more efficient format for processing. From [8](#)

- List of lists (LIL)

LIL stores one list per row, with each entry containing the column index and the value. Typically, these entries are kept sorted by column index for faster lookup. This is another format good for incremental matrix construction. From 8

- Coordinate list (COO)

COO stores a list of (row, column, value) tuples. Ideally, the entries are sorted first by row index and then by column index, to improve random access times. This is another format that is good for incremental matrix construction. From 8

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

We explained in 3.2 . From 6

- Compressed sparse column (CSC or CCS)

CSC is similar to CSR except that values are read first by column, a row index is stored for each value, and column pointers are stored. For example, CSC is (val, rowInd, colPtr), where val is an array of the (top-to-bottom, then left-to-right) non-zero values of the matrix; rowInd is the row indices corresponding to the values; and, colPtr is the list of val indexes where each column starts. From 8

## 3.2 Compressed Row Storage (CRS)

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.

### 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. From 6

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

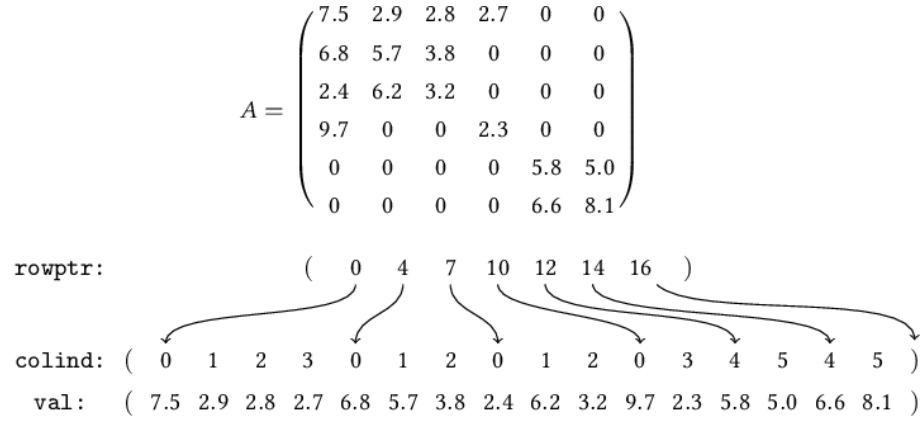


FIGURE 3.1: convert sparse matrix to CSR format

The **val** (3.1) represents the non-zero values of the matrix, read first by row left to right, then by column top to bottom. The **colind** represents the column index corresponding to the values. The **rowptr** 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.

In this project, We use Compressed sparse row (CSR, CRS or Yale format) for our implementation.

### 3.3 Sparse matrix - vector multiplication

#### 3.3.1 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:

$$y_i = \sum_{j=1}^n a_{ij}x_j \quad i \in [1...m] \quad (3.1)$$



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

FIGURE 3.2: sparse matrix

which would be represented by the CSR format vectors:

val =[1 3 2 4 6 5]

val is the non zero element in sparse matrix Figure 3.1.

col =[1 4 2 3 4 2]

col is the column number of non zero element in sparse matrix Figure 3.1.

row =[1 3 6 6 7]

row is the row number of non zero element in sparse matrix Figure 3.1.

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

### 3.3.2 C++ implementation

The C++ implementation is the most basic implementation. The idea is below

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

LISTING 3.1: matrix vector multiplication in C++

Listing 3.1, 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  $O(\text{size of non-zero elements})$ .

### 3.3.3 OCCA implementation

We implemented this algorithm in OKL as follow

```
1  for i=0; i < #row of matrix; i++; @tile(16, @outer, @inner)
```

```

2   for j = row[i]; j < row[i+1]; j++
3       result[i] += non_zero[j] * vector[col[j]]

```

LISTING 3.2: matrix vector multiplication in OCCA

Listing 3.2, 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 restriction in accessing the OCCA 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.

### 3.3.4 OCCA vs CPU

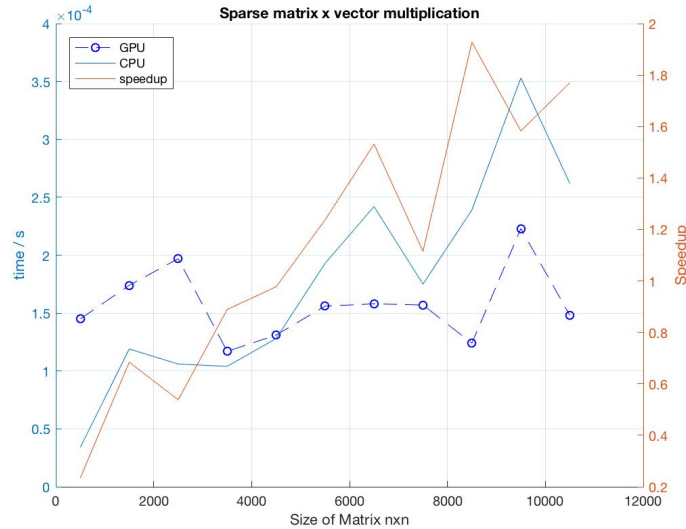


FIGURE 3.3: comparing OCCA vs CPU performance

To test the performance of CPU and OCCA, we took a square sparse matrix and a vector. We save a sparse matrix in CSR format. Now we have 3 vectors, 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. The throughputs 3.3 of the CPU and OCCA as shown above. As we can see that the performance of CPU is better than OCCA, if dimension of the matrix is less than 4000.

But dimension bigger than 4000, CPU performance is start fall down. And OCCA start perform better than CPU.

## 3.4 Sparse matrix multiplication

### 3.4.1 Introduction

Matrix multiplication has been implemented in the API for all kinds of matrices. We will focus on both CSR format 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 described in (3.2)

$$C_{ij} = a_{i1}b_{1j} + \cdots + a_{im}b_{mj} = \sum_{k=1}^m a_{ik}b_{kj} \quad (3.2)$$

for  $i = 1, \dots, n$  and  $j = 1, \dots, p$ .

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

FIGURE 3.4: A and B sparse matrix

which would be represented by the vectors:

$$\begin{aligned} A\_val &= [1 \ 3 \ 2 \ 4 \ 6 \ 5] & B\_val &= [1 \ 3 \ 2 \ 4 \ 6 \ 5] \\ A\_col &= [1 \ 4 \ 2 \ 3 \ 4 \ 2] & B\_col &= [1 \ 4 \ 2 \ 3 \ 4 \ 2] \\ A\_row &= [0 \ 0 \ 1 \ 1 \ 1 \ 3] & B\_row &= [0 \ 0 \ 1 \ 1 \ 1 \ 3] \end{aligned}$$

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

### 3.4.2 C++ implementation

The C++ implementation pseudo code is below.

```
1 | we use 2 vector for save the result of multiplication
```

```

2 int m =0;
3 for j < # of A non-zero elements; j++
4   for k < # of B non-zero elements; k++
5     check condition a_col[j] == b_row[k]
6     check if the position of matrix already in array return position s
7     result[s] += A_val[j]*B_val[k]
8   otherwise
9     result[m] = A_val[j]*B_val[k]
10    position array[m] = A_row[j]* #number of column in matrix B
11    + B_col[k]
12    m++

```

LISTING 3.3: matrix multiplication in C++

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.

### 3.4.3 OCCA implementation

The OCCA implementation in OKL:

```

1 for j < # of A non-zero elements; j++
2   for o0 < # of B non-zero elements; o0++; outer
3     for k = o0 k < o0+16; k++
4       check condition a_col[j] == b_row[k]
5       result[A_row[j]* # column in
6       matrix B + B_col[k]] = A_val[j]*B_val[k]

```

LISTING 3.4: matrix multiplication in OCCA

The OCCA uses 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 (OCCA). We use inner tag in for-loops, which able to vectorized (CPU) or work concurrently (OCCA). 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. It is not implemented till now.

#### 3.4.4 OCCA vs CPU

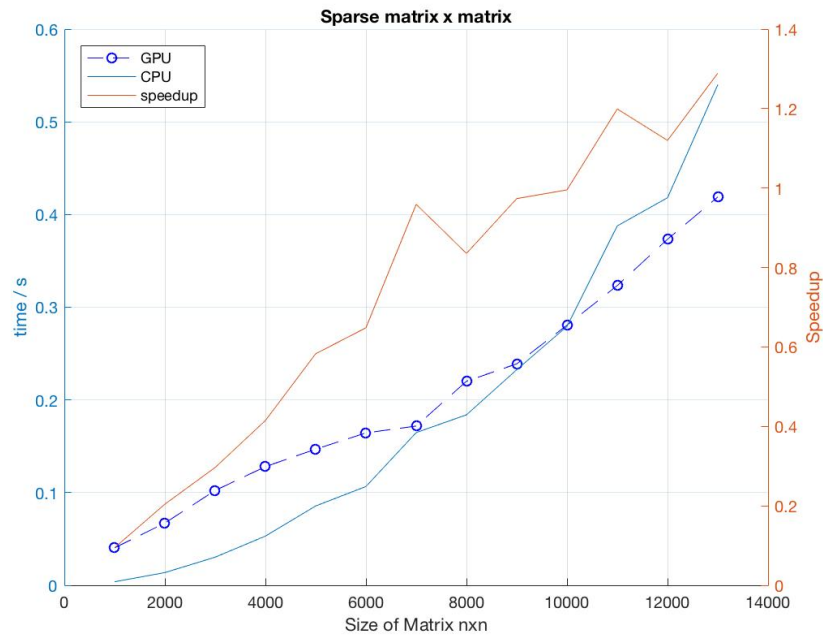


FIGURE 3.5: comparing OCCA vs CPU performance

To test the performance of CPU and OCCA, we took two square sparse matrix. We save each sparse matrix in 3 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 OCCA as shown above. As we can see that the performance of CPU is better than OCCA, if dimension of the matrix is less than 10000. But dimension bigger than 10000, CPU performance is starting fall down. And OCCA 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.

## 3.5 Sparse matrix - matrix addition & subtraction

### 3.5.1 Introduction

Sparse matrix addition or subtraction is not difficult. Firstly, we have to save both matrix in CSR formats.

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

$$C_{ij} = a_{ij} \pm b_{ij} \quad (3.3)$$

for  $i = 1, \dots, n$  and  $j = 1, \dots, p$ .

That is, the entry  $C_{ij}$  is the sum or subtraction of the  $i^{th}$  row and  $j^{th}$  of A and the  $i^{th}$  row and  $j^{th}$  of B.

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

FIGURE 3.6: matrix addition

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.

### 3.5.2 CPU implementation

CPU implementation addition (or subtraction) operation for sparse matrix with CSR format is described in the pseudocode:

```

1 we have two vector of #non zero elements in A + #non zero elements in B
2
3 # copy A_non_zero elements to result
4 for i < #non zero elements in A; i++
5     result[i] = A_non Zero [i]
6     position[i] = A_row[i] * #number of column matrix A + A_col_number[i]
7 # copy B_non_zero elements to result
8 for i < #non zero elements in B; i++

```

```

9   result[i+#non zero elements in A] = B_non Zero [i]
10  position[i+#non zero elements in A] = B_row[i] * #number of column matrix
    B + B_col_number[i]
11 # checking if matrix a and matrix b have non zero element on same position
12 for i < #non zero elements in A + #non zero elements in B-1;i++
13   for j < #non zero elements in A + #non zero elements in B;j++
14     check condition i !=j && i<j
15     check condition position[i] == position[j]
16     result[i] += result[j]
17     result[j] = 0;
18     position[j] = 0;

```

LISTING 3.5: matrix addition (or subtraction) in C++

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.

### 3.5.3 OCCA implementation

The OCCA implementation for sparse matrix addition or subtraction pseudocode is below:

```

1 we have two vector of #non zero elements in A + #non zero elements in B
2 for i < #non zero elements in A;i++;@tile(16, @outer, @inner)
3   result[i] = A_non Zero [i]
4   position[i] = A_row[i] * #number of column matrix A + A_col_number[i]
5 for i < #non zero elements in B;i++;@tile(16, @outer, @inner)
6   result[i+#non zero elements in A] = B_non Zero [i]
7   position[i+#non zero elements in A] = B_row[i] * #number of column matrix
    B + B_col_number[i]
8 for o1 < #non zero elements in A + #non zero elements in B-1;o1++;@outer
9   for o0 < #non zero elements in A + #non zero elements in B;o0++;@outer
10     for i = o1; i < o1+16;i++;@inner
11     for j = o0; j < o0+16;j++;@inner

```

```

12     check condition i !=j && i<j
13     check condition position[i] == position[j]
14         result[i] += result[j]
15     result[j] = 0;
16     position[j] = 0;

```

LISTING 3.6: matrix addition (or subtraction) in OCCA

Both plus and minus share the same code. 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 OCCA 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 (OCCA). Afterward we have for-loops with the inner tag, which able to be vectorized (CPU) or or work concurrently (OCCA). We have to check the condition  $i \neq j$  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.

### 3.5.4 OCCA vs CPU

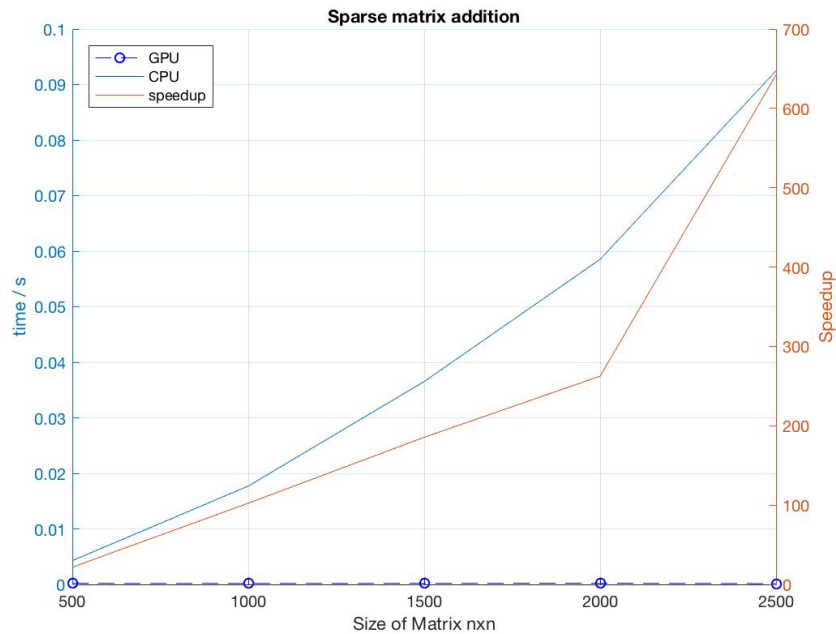


FIGURE 3.7: compare OCCA vs CPU performance



As we can see, the performance of CPU is not very good 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 semilogy for plotting in Matlab. The throughputs of the CPU and OCCA as shown above. As we can see, the performance of OCCA is better than CPU, if we choose matrices size bigger.

## Chapter 4

# Dot product

### 4.1 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  $\mathbf{a} = [a_1, a_2, \dots, a_n]$  and  $\mathbf{b} = [b_1, b_2, \dots, b_n]$  is defined as:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^n a_i b_i = a_1 b_1 + a_2 b_2 + \dots + a_n b_n \quad (4.1)$$

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

$$\begin{bmatrix} 1 & 3 & -5 \end{bmatrix} \begin{bmatrix} 4 \\ -2 \\ -1 \end{bmatrix} = 1 * 4 + 3 * (-2) + (-5) * (-1) = 3. \quad (4.2)$$

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

### 4.2 CPU implementation

The CPU implementation is below:

```
1   for i < size of vector; i++  
2       result += vector1[i] * vector2[i]  
3
```

LISTING 4.1: vector dot product in C++

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.

### 4.3 OCCA implementation / Reduction

Kernel implementation of dot product is below.

```

1  for (int group = 0; group < ((entries + block - 1) / block); ++group;
    @outer) {
2      shared float s_vec[256];
3      for (int item = 0; item < block; ++item; @inner) {
4          if ((group * block + item) < entries) {
5              s_vec[item] = vec[group * block + item] * vec2[group *
block + item];
6          } else {
7              s_vec[item] = 0;
8          }
9      }
10     for (int alive = ((block + 1) / 2); 0 < alive; alive /= 2) {
11         for (int item = 0; item < block; ++item; @inner) {
12             if (item < alive) {
13                 s_vec[item] += s_vec[item + alive];
14             }
15         }
16         for (int item = 0; item < block; ++item; @inner) {
17             if (item == 0) {
18                 blockSum[group] = s_vec[0];
19             }
20         }
21     }
22 }
23

```

LISTING 4.2: vector dot product in OCCA

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.

## 4.4 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.

## Chapter 5

# Gaussian elimination

### 5.1 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

$$1x + 3y = 3$$

$$5y = 1$$

$$x = 12/5$$

$$y = 1/5$$

### 5.2 C++ implementation

The C++ implementation as below:

```
1 | void Gauss_elimination_cpu(float a[], float d[], int n) {
```

```

2  int i, j, k, temp;
3  //***** Forward elimination process*****//
4  for (int i = 0; i < n - 1; i++) {
5      for (int k = i + 1; k < n; k++) {
6          float c = a[k * (n + 1) + i] / a[i * (n + 1) + i];
7          for (int j = i; j <= n; j++) {
8              a[k * (n + 1) + j] = a[k * (n + 1) + j] - (c * a[i * (n + 1) + j]);
9          }
10     }
11 }
12 //***** Backward Substitution method*****//
13 for (i = n - 1; i >= 0; i--){
14     d[i] = a[i * (n + 1) + n];
15     for (int j = i + 1; j < n; j++) {
16         if (j != i) {
17             d[i] = d[i] - a[i * (n + 1) + j] * d[j];
18         }
19     }
20     d[i] = d[i] / a[i * (n + 1) + i];
21 }
22 }

```

LISTING 5.1: Gauss elimination in C++

**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.

### 5.3 OCCA implementation

Kernel implementation of Gauss Elimination is below.

```

1  //***** Forward elimination process*****//
2  for (int k = i+1; k<n; k++; @tile(16, @outer, @inner)) {

```

```

3      float c = a[k*(n+1)+i]/a[i*(n+1)+i];
4      for (int j=0; j<= n; j++) {
5          a[k*(n+1)+j]=a[k*(n+1)+j]-(c*a[i*(n+1)+j]);
6      }
7  }
8 }
9 //***** Backward Substitution method*****//
10 for (int k =0; k < 1; k++; @tile(16, @outer, @inner)) {
11     d[i] = a[i*(n+1)+n];
12     for (int j =0; j <n; j++) {
13         if (j > i) {
14             d[i] = d[i] - (a[i*(n+1)+j] * d[j]);
15         }
16     }
17     d[i] = d[i]/a[i*(n+1)+i];
18 }
19 }

```

LISTING 5.2: Gauss elimination in OCCA

In OCCA, 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.

## 5.4 OCCA vs CPU

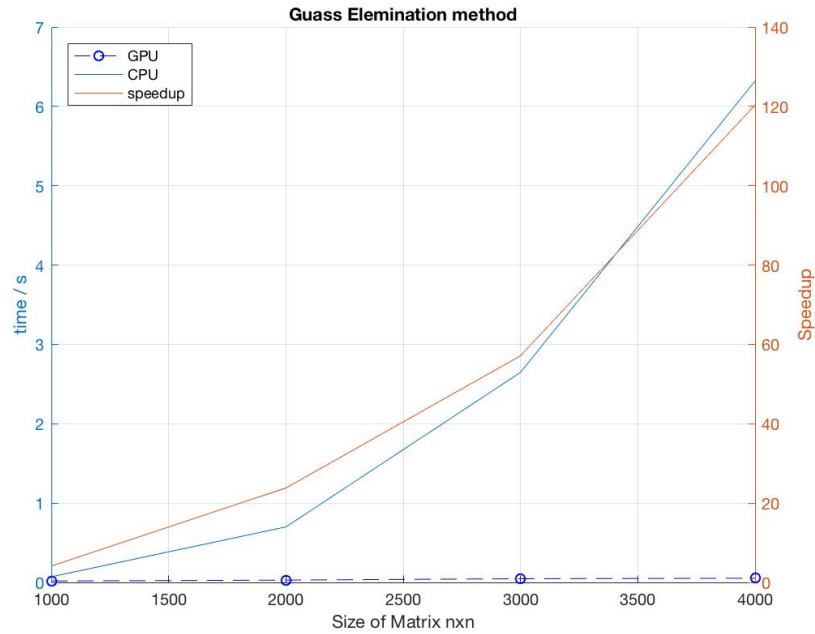


FIGURE 5.1: compare OCCA vs CPU performance

As we can see CPU will more expensive than OCCA. 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 OCCA is better than CPU.



## Chapter 6

# Jacobi method

### 6.1 Introduction

Jacobi method is an iterative solver used for approximating the solution of diagonally dominant system of linear equations. 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

$$\sum_{j=1}^n a_{i,j}x_j = b_i \quad (6.1)$$

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

$$x_i = (b_i - \sum_{j \neq i} a_{i,j}x_j)/a_{i,i} \quad (6.2)$$

This suggests an iterative method defined by

$$x_i^{(k)} = (b_i - \sum_{j \neq i} a_{i,j}x_j^{(k-1)})/a_{i,i} \quad (6.3)$$

Which is the Jacobi method.

### 6.2 C++ implementation

The C++ implementation as below:

```
1  for (int k = 0; k < num_iter; k++) {  
2      for (int i = 0; i < n; i++) {  
3          float sum = 0;  
4          for (int j = 0; j < n; j++) {
```

```

5         if ( j != i ) {
6             sum += (a[i * (n + 1) + j] * x[j]);
7         }
8     }
9     x_new[i] = ((a[i * (n + 1) + n]) - sum) / a[i + i * (n + 1)];
10 }
11 for (int i = 0; i < n; i++) {
12     x[i] = x_new[i];
13 }
14 }

```

LISTING 6.1: Jacobi method in C++

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.

### 6.3 OCCA implementation

The OCCA implementation as below:

```

1  for (int i = 0; i < n; i++; @tile(16, @outer, @inner) ) {
2      x_new[i] = a[i * (n + 1) + n];
3      float sum = 0;
4      for (int j = 0; j < n; j++) {
5          if ( j != i ) {
6              sum += a[i + j * (n + 1)] * x[j];
7          }
8      }
9      x_new[i] = ((a[i * (n + 1) + n]) - sum) / a[i + i * (n + 1)];
10 }
11 for (int i = 0; i < n; i++; @tile(16, @outer, @inner){
12     x[i] = x_new[i];
13 }

```

LISTING 6.2: Jacobi method in OCCA

This is OCCA 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.

## 6.4 OCCA vs CPU

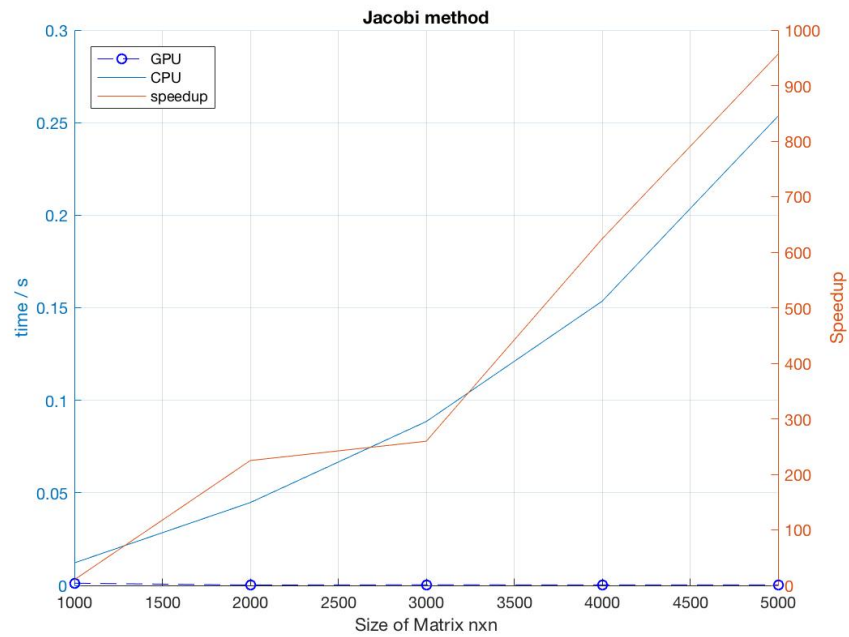


FIGURE 6.1: compare OCCA vs CPU performance

For Jacobi method, OCCA is better than CPU. We can compare according to our results. We compare the matrix size from 1000X1000 to 5000X5000. In this size matrix, OCCA 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.

## Chapter 7

# Multigrid Method

### 7.1 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. From [2](#)

### 7.2 Multigrid pseudo-code

The main structure of a MultiGrid algorithm should be describe by following sequence of operations [7.1](#)

$$v^h = MultiGrid(A^h, v^h, f^h, \alpha_1, \alpha_2) \quad (7.1)$$

$$Relax \quad \alpha_1 \quad times \quad on \quad A^h u^h = f^h \quad on \quad \Omega^h \quad with \quad arbitrary \quad initial \quad guess \quad v^h \quad (7.2)$$

$$compute \quad residual \quad on \quad fine \quad grid \quad r^h = f^h - A^h v^h \quad (7.3)$$

$$reduce \quad residual \quad on \quad coarse \quad grid \quad r^{2h} = I_h^{2h} r^h \quad (7.4)$$

$$\text{reduce matrix on coarse grid } A^{2h} = I_{2h}^h A^h I_h^{2h} \quad (7.5)$$

$$\text{Recursive call to MultiGrid to solve } A^{2h} e^{2h} = r^{2h} \text{ on } \Omega^{2h} \quad (7.6)$$

$$\text{correct fine grid solution } v^h = v^h + I_{2h}^h e^{2h} \quad (7.7)$$

$$\text{Relax } \alpha_2 \text{ times on } A^h u^h = f^h \text{ on } \Omega^h \text{ with } v^h \quad (7.8)$$

In this procedure, we relax  $\alpha_1$  times the system of equation  $A^h u^h = f^h$  with the given initial guess  $v^h$ . In this project we use the Jacobi method. We compute the residuals  $r^h$  with the new  $v^h$ . After that, we use the reduction operator on  $r^h$ . We reduce the matrix  $A^h$  by applying the reduction and interpolation operators. Which return  $A^{2h}$ . Now, We use the recursion with our new system  $A^{2h} e^{2h} = r^{2h}$  to solve system on coarse grid  $\Omega^{2h}$ . And after that we correct the solution on fine grid  $v^h$  by using the correction  $e^{2h}$ . And finally, we relax with  $\alpha_2$  times our  $A^h u^h = f^h$  with the initial guess  $v^h$ .

## 7.3 Multigrid mehtod Dense matrix

### 7.3.1 Introduction

In the multi-grid method, We use the Gauss elimination method as exact solver, Jacobi method for relaxation. Some specific functions for matrix interpolation, matrix reduction, vector interpolation. The main idea is

```

1 multi-grid method(a, b, x, recursion)
2     if (recursion == 0)
3         gauss-elimination method(a, b, x)
4         return
5     jacobi method(a, b, x, 10)
6
7     \\reduction of vector b
8     b2h = Reduction (b - a*x)
9
10    \\ initialize
11    x2h = 0
12
13    \\ compute a2h, R = reduction matrix, I = interpolation matrix
14    a2h = R * a * I
15
16    multi-grid method(a2h, b2h, x2h, recursion-1)
```

```

17
18     xh = interpolation x2h
19
20     x = x + xh
21
22     jacobi method(a, b, x, 10)

```

LISTING 7.1: multigrid method idea

With the reduction, We decrease the size of the vector. For example if we have a vector of size  $n$ , after reduction we obtain a new vector of size  $n/2$ . We call the multi-grid method again with the reduced  $A$ ,  $b$  and  $x$ . And we repeat this process until the recursion counter is equal to 0; and when it is 0, It calls the Gauss elimination method to solve the given system by stopping the recursive calls.

### 7.3.2 C++ implementation

The C++ implementation as below:

```

1  if (recursion == 0) {
2
3      Gauss_elimination_cpu(a, b, x, row);
4      return ;
5  }
6  float *x_new2 = new float [row];
7
8  init_zero(x_new2, row);
9  jacobi_method_cpu(a, x, b, x_new2, row, alpha);
10
11  float *b2h = new float [row / 2];
12  float *res1 = new float [row];
13  float *x_new2h = new float [row];
14  float *a2h = new float [(row / 2) * (row / 2)];
15
16  init_zero(b2h, row / 2);
17  init_zero(res1, row);
18  init_zero(x_new2h, row);
19  init_zero(a2h, (row / 2) * (row / 2));
20
21  matrix_x_vector(row, row, x, a, x_new2h);
22

```

```

23     add_sub_vector(b, x_new2h, res1, row, -1);
24
25     reduction_vector(res1, (row / 2), b2h);
26
27     init_zero(x_new2h, row);
28
29     interpolation_reduction_matrix(a, row, a2h);
30
31     if (row / 2 <= 0) {
32         cout << "error" << endl;
33         return;
34     }
35     multigrid_method(a2h, x_new2h, b2h, recursion - 1, row / 2, alpha);
36
37     float * res_int = new float[(row * 2) + 1];
38     init_zero(res_int, (row * 2) + 1);
39
40     reduction_interpolation_vector(x_new2h, row, res_int);
41     add_sub_vector(x, res_int, x, row, 1);
42
43     jacobi_method_cpu(a, x, b, x_new2, row, alpha);

```

LISTING 7.2: Multigrid method in C++

In CPU, It is the same implementation as I describe above. Firstly, We check if the 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.

### 7.3.3 OCCA implemintation

The OCCA implementation as below:

```

1     if (recursion == 0) {
2         gauss_elimination_call_gpu(row, o_a, o_b, o_x, device);
3         return;
4     }
5     occa::memory o_d, o_b2h, o_x2h, o_a2h, o_res, o_res_result2h;

```

```

6      // Allocate memory on the device
7
8      o_d = device.malloc(row * sizeof(float));
9      o_b2h = device.malloc((row / 2) * sizeof(float));
10     o_x2h = device.malloc(row * sizeof(float));
11     o_res = device.malloc(row * sizeof(float));
12     o_a2h = device.malloc((row / 2) * (row / 2) * sizeof(float));
13     o_res_result2h = device.malloc(((row * 2) + 1) * sizeof(float));
14
15     jacobi_method_call_gpu(row, o_a, o_b, o_x, o_d, device, alpha);
16
17     dense_Matrix_Vector_Multiplication_call_gpu(row, o_a, o_x, o_res,
18     device);
19
20     add_sub_call_gpu(row, o_b, o_res, o_res, device, -1);
21
22     relaxation_reduction_vector(row / 2, o_res, o_b2h, device);
23
24     reduction_interpolation_reduction_matrix_call_gpu(row, o_a, o_a2h,
25     device);
26
27     if (row / 2 <= 0) {
28         cout << "error" << endl;
29         return;
30     }
31
32     multigrid_method_gpu(row / 2, o_a2h, o_b2h, o_x2h, device, recursion -
33     1, alpha);
34
35     relaxation_interpolation_vector_call_gpu(row, o_x2h, o_res_result2h,
36     device);
37
38     add_sub_call_gpu(row, o_x, o_res_result2h, o_x, device, 1);
39
40     jacobi_method_call_gpu(row, o_a, o_b, o_x, o_d, device, alpha);

```

LISTING 7.3: multigrid method in OCCA

It has same idea like CPU implementation, We just make all calculation on OCCA. And we described before Gauss elimination and Jacobi method in OCCA.



### 7.3.4 OCCA vs CPU

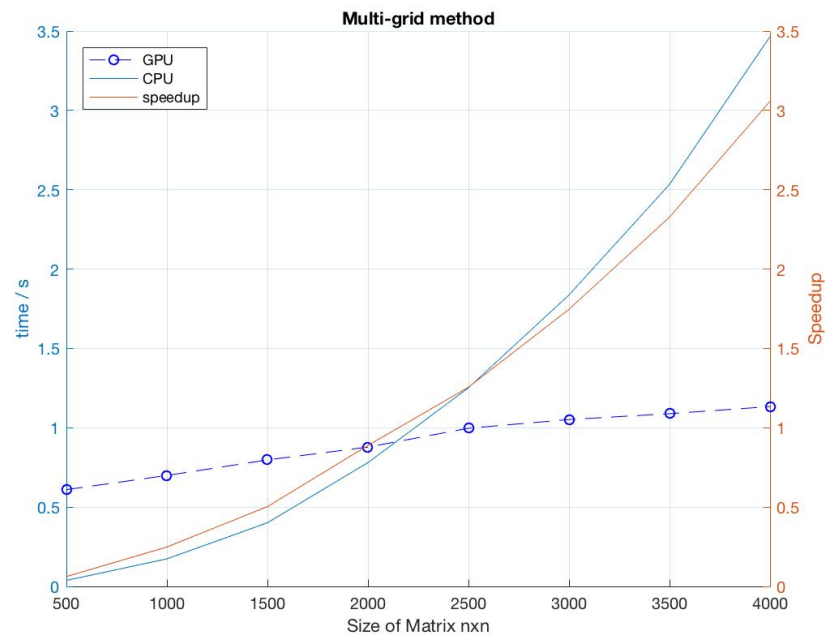


FIGURE 7.1: compare OCCA vs CPU performance

In this graph 7.1 we observe that if the matrix size is small CPU is better than OCCA with GPU. And according to size of matrix, the CPU time increase faster also. But OCCA is increase very slightly. In my opinion, OCCA is much better if the matrix size is large and OCCA with GPU is faster than CPU.

### 7.3.5 Numerical analysis of results

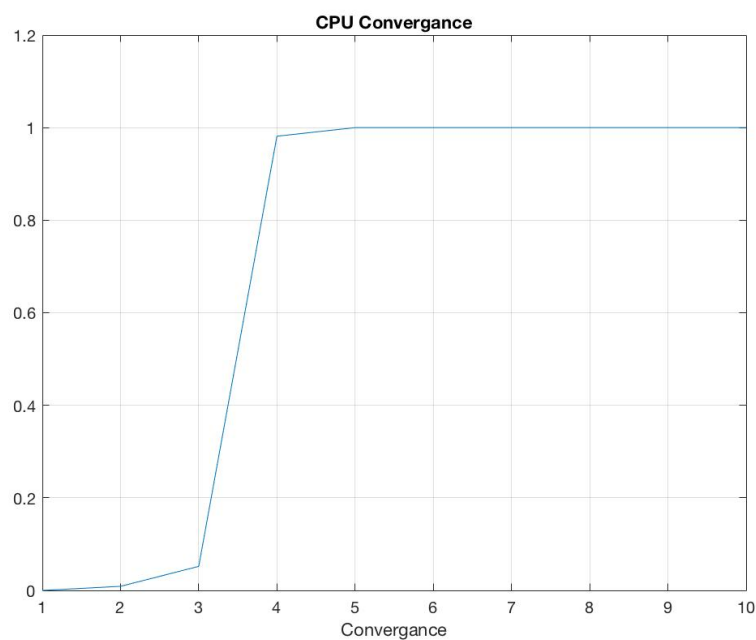


FIGURE 7.2: Convergence rate

This convergence rate 7.2, which refer to SPD matrix 2000 by 2000. We have used 32 bit plotting point. In 4s Multigrid step, it converge to solution.

## 7.4 Multigrid mehtod sparse matrix

### 7.4.1 C++ implementation

The C++ implementation as below:

```

1  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) {
2
3      if (recursion == 0 || row / 2 < 3 ) {
4          int *point = new int[size_a];
5          float *aa = new float [row * row];
6          for (int i = 0; i < size_a; i++) {
7              point[i] = a_col_number[i] * row + a_row[i];
8          }
9
10         vectorToMatrix(row, row, size_a, a_non_zero, aa, point);
11
12         Gauss_elimination_cpu(aa, b, x, row);
13
14         delete [] point;
15         delete [] aa;
16
17         return ;
18     }
19
20
21     float *x_new2 = new float [row];
22     init_zero(x_new2, row);
23
24     jacobi_method_cpu_sparse_matrix(a_non_zero, a_col_number, a_row, b, x,
    x_new2, row, alpha, size_a);
25
26
27     float *b2h = new float [row / 2];

```

---

```

28     float *res1 = new float [row];
29     float *x_new2h = new float [row];
30
31     init_zero(b2h, row / 2);
32     init_zero(res1, row);
33     init_zero(x_new2h, row);
34
35     sparse_matrix_x_vector(row, size_a, x, a_row, a_col_number, a_non_zero,
36                             x_new2h);
37     add_sub_vector(b, x_new2h, res1, row, -1);
38     reduction_vector_sparse(res1, row, b2h);
39
40
41     int size_non = (size_a + row) * 3;
42
43     float *a2h = new float [size_non];
44     int *a2h_row = new int [size_non];
45     int *a2h_col = new int [size_non];
46
47     init_zero(a2h, size_non);
48     init_zero(a2h_row, size_non);
49     init_zero(a2h_col, size_non);
50
51
52     size_non = interpolation_reduction_matrix_sparse_matrix(a_non_zero,
53                                                            a_col_number, a_row, size_a, row, a2h, a2h_row, a2h_col);
54
55
56     multigrid_method_sparse_matrix(a2h, a2h_col, a2h_row, x_new2h, b2h,
57                                     recursion - 1, row / 2, alpha, size_non);
58
59
60     float * res_int = new float [(row * 2) + 1];
61
62     init_zero(res_int, (row * 2) + 1);
63
64     reduction_interpolation_vector(x_new2h, row, res_int);
65
66     add_sub_vector(x, res_int, x, row, 1);
67
68     init_zero(x_new2, row);
69

```

```

66     jacobi_method_cpu_sparse_matrix(a_non_zero , a_col_number , a_row , b, x,
        x_new2, row, alpha , size_a);
67
68
69     delete [] x_new2h;
70     delete [] b2h;
71     delete [] res1;
72     delete [] res_int;
73     delete [] a2h;
74     delete [] a2h_col;
75     delete [] a2h_row;
76 }

```

LISTING 7.4: multigrid method in C++

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 recursively 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 jacobi method again alpha time.

#### 7.4.2 OCCA implementation

The OCCA implementation as below:

```

1  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) {
2      if (recursion == 0 || row / 2 <= 3 || size_a < row) {
3          occa::memory o_aa;
4
5          o_aa = device.malloc((row * row) * sizeof(float));
6
7          sparse_vector_to_matrix_gpu_call(row, o_a, o_a_row, o_a_col, device
            , size_a, o_aa);
8
9          gauss_elimination_call_gpu(row, o_aa, o_b, o_x, device);
10

```

---

```

11         return;
12     }
13
14     occa::memory o_b2h, o_x2h, o_a2h, o_a2h_row, o_a2h_col, o_res, o_res2,
o_res_result2h, o_row_number ;
15     // Allocate memory on the device
16
17     o_b2h = device.malloc((row / 2) * sizeof(float));
18     o_x2h = device.malloc((row / 2) * sizeof(float));
19     o_res = device.malloc(row * sizeof(float));
20     o_res2 = device.malloc(row * sizeof(float));
21
22     int size_non = (size_a + row);
23     o_a2h = device.malloc(size_non * sizeof(float));
24     o_a2h_row = device.malloc(size_non * sizeof(int));
25     o_a2h_col = device.malloc(size_non * sizeof(int));
26
27
28     jacobi_method_call_gpu_sparse_matrix(row, o_a, o_a_col, o_a_row, o_b,
o_x, device, alpha, size_a);
29
30     sparse_Matrix_Vector_Multiplication_call_gpu(row, size_a, o_a, o_a_row,
o_a_col, o_x, o_res, device);
31
32     add_sub_call_gpu(row, o_b, o_res, o_res2, device, -1);
33
34     relaxation_reduction_vector(row / 2, o_res2, o_b2h, device);
35
36     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);
37
38
39     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);
40
41     o_res_result2h = device.malloc(((row * 2) + 1) * sizeof(float));
42
43     relaxation_interpolation_vector_call_gpu(row, o_x2h, o_res_result2h,
device);
44

```

```

45     add_sub_call_gpu(row, o_x, o_res_result2h, o_x, device, 1);
46
47     jacobi_method_call_gpu_sparse_matrix(row, o_a, o_a_col, o_a_row, o_b,
48     o_x, device, alpha, size_a);

```

LISTING 7.5: multigrid method in OCCA

It has same idea like CPU implementation, We just make all calculation on GPU. And we described before Gauss 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 matrix size. Thats why, We change CSR format matrix to sparse matrix with 0's and than call to Gauss elimination method.

### 7.4.3 OCCA vs CPU

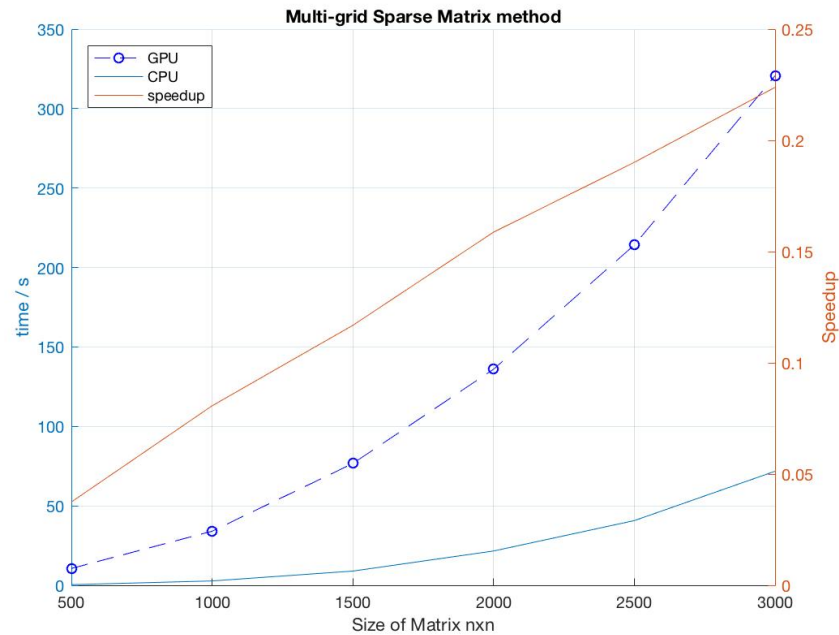


FIGURE 7.3: compare OCCA vs CPU performance

We can see in this graph 7.3 CPU is always faster than OCCA. OCCA, We are call to kernel for each function and copy data also take too much time. We can not implement all the function in kernel, because we do not have any atomic operation in OCCA. We have to return to CPU and call again to kernel. That took too much time. And result, We can see CPU is better than OCCA, but if we consider dense matrix OCCA is better.

#### 7.4.4 Numerical analysis of results

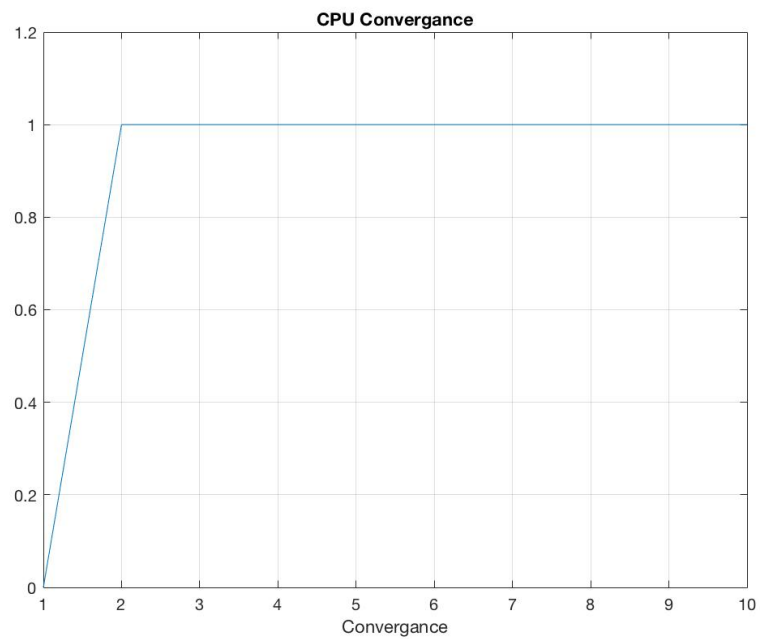


FIGURE 7.4: Convergence rate

This convergence rate 7.4, which refer to SPD sparse matrix 2000 by 2000 and sparsity ratio is 0.05. We have used 32 bit plotting point. In one Multigrid step, it converge to solution.

# Conclusion

This thesis, we develop some algorithms with OCCA and C++ by comparing the performances. The time performance of the implementations shows that OCCA with GPU is normally faster than C++ and CPU. In all cases, the timing results recommend the usage of OCCA instead of the CPU to get a better performance when the input size of the matrices is large. In our case, We got better performance in all algorithms. But with sparse matrices the performance are not always good. OCCA with GPU is slower than C++ and CPU; because at the moment it not support atomic operation, and other advanced parallel programming features. In some methods, due to OCCA limitation, we can not do parallel computation and we have to return to CPU and call back to kernel that is time consuming.

OCCA is easy to use and understand. OCCA uses a syntax similar to C. It has similar for loops (with additional field) and conditional statements. It is good solution, If somebody want to develop parallel applications for different architectures like openCL and CUDA (it works with same code for both architectures). We do not need to write separate code for OpenCL and Cuda. We have shared or exclusive memory operations in OCCA.

The problem of OCCA is that at the moment, the main structure of the project is not completely defined. The biggest problem is the lack of documentation. It has just 2 or 3 slide of documentation and 7 or 8, very short examples. So It is hard to understand, how to implement an optimal kernel.

For example in this project, I need atomic operations. But at the moment are not yet supported by OCCA. And the syntax of the kernel is not yet completely defined. For example during the development of this project, the OCCA developers has decided to change a little the syntax of the kernel. And I have spent some time to adapt all the methods.

Currently, OCCA developers are working on loop-carried dependency analysis for kernel



generation + testing and support offset in kernel calls. They are also developing tiling loop labels for example `tile(x,y)` and possibly loop-collapsing. And working also on atomic operations.

I think OCCA is good solution for parallel programming with GPU. Because an application can work on all different devices like Nvidia, Intel and Radeon GPUs. So the developers are not constrained to a specific family of products. But at the moment, as described, the big problem of OCCA is that it is a work in progress. So, It is very interesting language, but now, it is not recommendable for long term or big projects.

**Bibliography:**

1. [L<sup>A</sup>T<sub>E</sub>X 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](#)