Determination of the Usability of FPGA Technology to Accelerate Process of Solving Eigenvalues and Eigenvectors

201373673 Hsieh, Tsung-Ta

A DISSERTATION

Submitted to

The University of Liverpool

in partial fulfilment of the requirements

for the degree of

MASTER OF SCIENCE

September 20 , 2019

Abstract

In recent years, a Field Programmable Gate Array (FPGA) has become a popular technology for many different fields in computer science. FPGA technology enables programmers to develop their own circuit for specific purposes without having to produce an application-specific integrated circuit (ASIC). In some specific applications, FPGA technology provides better performance than CPU. For example, in this project, we designed bitstreams on FPGA to calculate eigenvalues and eigenvectors which provides better performance than CPU. As a result, FPGA can be deployed in HPC environment because programmers are able to change the overlays in FPGA anytime, which makes they can design specific applications in FPGA.

Finding eigenvalues is an important mathematical technique for many fields in computer science. This dissertation hypothesizes that such a process can be efficiently implemented on an FPGA. We consider two popular algorithms to determine eigenvalues, namely the QR algorithm and Jacobi method. We implemented the QR algorithm and Jacobi method on two FPGA platforms that are PYNQ Z2 [11] and Xilinx Alveo U200 [31] to accelerate the process of finding eigenvalues and eigenvectors for real symmetric matrices. Utilizing the techniques of implementing code on FPGA, the original code for CPU can be optimized and ported to FPGA platforms. Moreover, the implementations with specific settings on FPGA can achieve better performance than CPU.

Student Declaration

I confirm that I have read and understood the University's Academic Integrity Policy.

I confirm that I have acted honestly, ethically and professionally in conduct leading to assessment for the programme of study.

I confirm that I have not copied material from another source nor committed plagiarism nor fabricated data when completing the attached piece of work. I confirm that I have not previously presented the work or part thereof for assessment for another University of Liverpool module. I confirm that I have not copied material from another source, nor colluded with any other student in the preparation and production of this work.

I confirm that I have not incorporated into this assignment material that has been submitted by me or any other person in support of a successful application for a degree of this or any other university or degree-awarding body.

SIGNATURE Tsung-Ta Hsieh

DATE September 20, 2019

Table of Content

1. Introduction 1
    1.1 Algorithms for Eigenvalues 1
        1.1.1 Givens Rotation 2
        1.1.2 QR algorithm 3
        1.1.3 Jacobi Eigenvalue Method 6
    1.2 Resources and Techniques on FPGA 10
        1.2.1 Techniques for Implementing Code on FPGA 10
        1.2.2 Fixed Type on FPGA 13
    1.3 Aims 13
    1.4 Data Requirement 13
2. Related Work 14
3. Implementation 15
    3.1 Platforms 15
    3.2 Implementation on PYNQ Z2 15
    3.3 Implementation on Xilinx Alveo 20
4. Result and Evaluation 25
    4.1 Execution Time on PYNQ Z2 25
    4.2 Execution Time on Alveo Card(s) 28
        4.2.1 Execution Time with DDR Memory on FPGA 28
        4.2.2 Execution Time with BRAM 33
        4.2.3 Executing Kernels Simultaneously 35
    4.3 Numpy VS Implementations on FPGA 37
5. Discussion 39
    5.1 Discussion of Data Type on FPGA 39
    5.2 Learning Points 40
    5.3 Professional Issues 41
6. Future Work 43
7. Conclusion 44
8. References 45
Appendix 47

1. Introduction

Finding the eigenvalues and related eigenvectors of a system is important to many fields. For example, Principle Component Analysis [1] is an important technique in data mining and machine learning. PCA projects high dimension data points into lower dimension. By finding the dominant eigenvalue of a covariance matrix, it is the maximum variance of the high dimension data points. The computational cost of finding eigenvalues and eigenvectors can be prohibitively large when the dimension of matrices is large. Eigenvalues and eigenvectors can be used in computer vision as well. For example, in face detection, it needs eigenvectors of a covariance vector in PCA to achieve its goal.

There are some approaches to accelerate this finding process. Many programmers utilize GPUs for this purpose. Furthermore, a Field Programmable Gate Array (FPGA) can be used to accelerate this process as well. FPGA has lower latency of inputting data than GPU. FPGA can receive data from many different interfaces such as PCI-E, UART, USB and so on. Microsoft's paper [2] accelerated Deep Conventional Neural Network [3] (CNN) using FPGA technology has shown that such technology can achieve lower power consumption and higher performance than GPUs. Moreover, programmers can design customized overlay for FPGA boards with software language like C/C++ or OpenCL. As a result, they can make a customized circuit without the prohibitive costs of investing in an Application Specific Integrated Circuit (ASIC). Some cloud platforms like Nimbix or AWS provide Xilinx Alveo U200 [4] instance which is an HPC environment. The specification of Alveo U200 will be mentioned in section 3.1.

The remainder of section 1 explains the algorithms for determining eigenvalues and eigenvectors, resources and developing techniques on FPGA, and aims. Section 2 is related work. In section 3, it describes how to implement the designs of both algorithms. In section 4, it represents the result and evaluation. In section 5, it discusses data type on FPGA and learning points. And section 6 and 7 are future work and conclusion.

1.1 Algorithms for Eigenvalues

For software development, programmers don't solve eigenvalues by utilizing the function $det(A - \lambda I) = 0$, especially, when the dimension of matrices is large. In practical design, programmers utilize approximate algorithms to find eigenvalues and

eigenvectors in matrices, and typically based on a number of iterations. The QR algorithm [5] is the most popular approach for this purpose and is applicable for general matrices. Alternatively, the Jacobi eigenvalue method [6] can be used for real symmetric matrices only. These methods are discussed in more detail below but note that both algorithms apply Givens rotation for transforming matrices. Therefore, we first outline what is meant by a Givens rotation. The eigenvalues of real symmetric matrices are real eigenvalues. For non-symmetric real matrices, their eigenvalues might be real or complex.

1.1.1 Givens Rotation

A Givens rotation [7] matrix $\mathrm{G}(\mathrm{i},\mathrm{j},\theta)$ can be utilized to calculate orthogonal matrices in QR factorization and the eigenvalues in Jacobi method. The typical form for a Givens matrix is:

$$G(i,j,\theta) = \begin{array}{c} \begin{matrix} & & \mathrm{i} & & \mathrm{j} & & \end{matrix} \\ \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & -s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \end{array} \begin{matrix} \\ \\ \mathrm{i} \\ \\ \mathrm{j} \\ \\ \\ \end{matrix}$$

By applying the Givens rotation, it can eliminate element a_{ji}. c in Givens rotation means $\cos\theta$, s means $\sin\theta$. There are many ways to decide θ. However, for both algorithms, they utilized different ways to decide the values of sin and cos without having a specific θ. In order to explain the idea of Givens rotation, we illustrate an approach to decide sin and cos by the following example. Let matrix A_1

$$A_1 = \begin{bmatrix} 6 & 5 & 0 \\ 5 & 1 & 4 \\ 0 & 4 & 3 \end{bmatrix}$$

Let i=1 and j=2, we want to eliminate element a_{21}. We can transform this matrix to A2 which contains $a'_{21} = 0$ by applying a Givens rotation. For i=1 and j = 2, setting $\cos\theta = \frac{a_{11}}{\sqrt{a_{11}^2+a_{21}^2}} = \frac{6}{\sqrt{6^2+5^2}}$ and $\sin\theta = -\frac{a_{21}}{\sqrt{a_{11}^2+a_{21}^2}} = -\frac{5}{\sqrt{6^2+5^2}}$. The Givens matrix G_1 is

$$G_1 = \begin{bmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{bmatrix} \approx \begin{bmatrix} 0.7682 & 0.6402 & 0 \\ -0.6402 & 0.7682 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

We then obtain the new matrix A_2 is

$$G_1 A_1 = A_2 \approx \begin{bmatrix} 7.8102 & 4.4813 & 2.5607 \\ 0 & -2.4327 & 3.0729 \\ 0 & 4 & 3 \end{bmatrix}$$

The example above is one way to calculate the values of sin and cos, there are other approaches to get sin and cos which will be presented in 1.1.2 and 1.1.3 respectively. This project only need to utilize sqrt() instead of sin() and cos() functions to calculate the values of sin and cos. By utilizing elements in matrices, it can get the values of sin and cos.

1.1.2 QR algorithm

QR algorithm [5] utilizes QR factorization to find an orthogonal matrix. Any matrix can be decomposed into two matrices Q and R, where R is an upper triangular matrix, and Q is an orthogonal matrix. The decomposition may not be unique.

By using a series of Givens rotation to eliminate all the lower triangular elements in A, the matrix A can be transformed iteratively to determine the upper triangular matrix R, and the orthogonal matrix Q will be the products of the Givens rotations:

$$\mathrm{A} = \mathrm{QR}$$
$$\mathrm{Q} = \mathrm{G}_1^T \mathrm{G}_2^{\mathrm{T}} \ldots \mathrm{G}_{\mathrm{n}}^{\mathrm{T}}$$
$$\mathrm{R} = \mathrm{Q}^{\mathrm{T}}\mathrm{A}$$

The QR algorithm utilized Q and R to create a new matrix. Let a matrix A_k which is decomposed to $\mathrm{Q_k}$ and $\mathrm{R_k}$. By using $\mathrm{R_k}$ multiplies $\mathrm{Q_k}$, an new matrix $\mathrm{A_{k+1}}$ can be created. $\mathrm{A_{k+1}}$ has the same eigenvalues and eigenvectors with $\mathrm{A_k}$

$$\mathrm{A_{k+1}} = \mathrm{R_k Q_k}$$

Moreover, the QR algorithm can only use orthogonal matrices for calculation.

$$\mathrm{A_{k+1}} = \mathrm{Q_k^T A_k Q_k}$$

In [5], it has mentioned the upper Hessenberg form can decrease the convergence time of the QR algorithm. So, at beginning, matrices should be transformed to upper Hessenberg form in order to decrease time of convergence. By using Givens rotation, a matrix can be transformed to upper Hessenberg form.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & a_{43} & a_{44} \end{bmatrix}$$

Also, by adding shift mechanism to this process, it can decrease time of convergence as well. By subtracting diagonal elements with a shift value, the subtracted matrix has the same eigenvalues with the original one but utilizing the subtracted matrix to do QR factorization can decrease the time of doing QR factorization. However, after doing each QR factorization, the new matrix needs to add shift value back to diagonal elements which is new matrix A_{k+1}. The value of shifting can be set to $\mathrm{a_{nn}}$ which is the element at last column and last row in $\mathrm{A_k}$

$$\mathrm{A_k - u_k I = Q_k R_k, u_k = a_{nn}}$$

$$\mathrm{A_{k+1} = Q_k^T (A_k - u_k I) Q_k + u_k I = Q_k^T A_k Q_k}$$

After doing QR factorization and calculating new matrix A repeatedly, non-diagonal elements are eliminated to 0. Eigenvalues are diagonal elements in the final matrix. Furthermore, eigenvectors can be calculated with Qs during the process.

$$\mathrm{EVs = Q_1 Q_2 \dots Q_n}$$

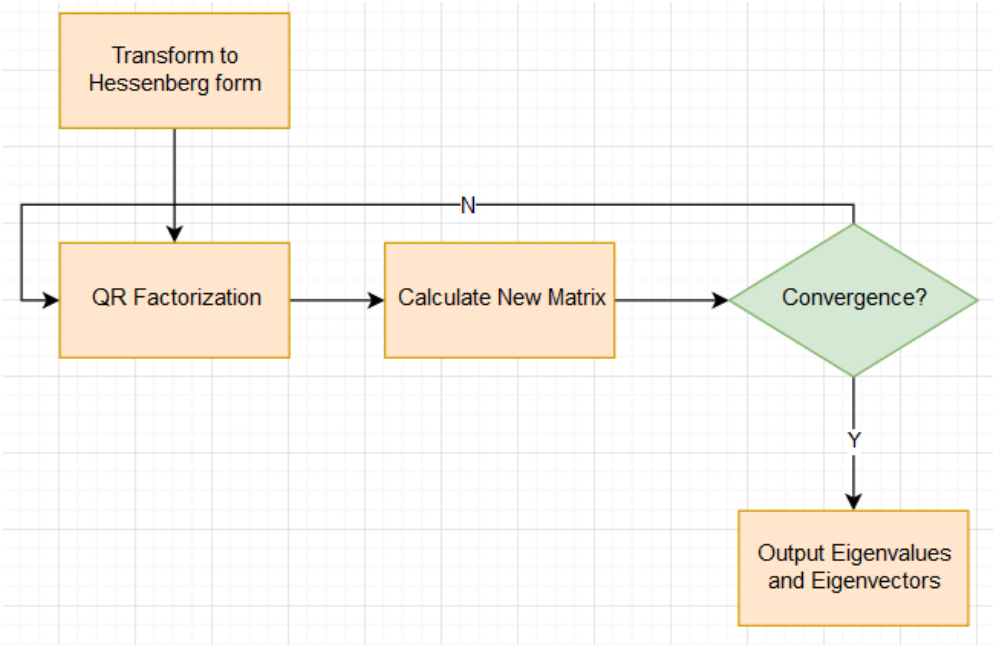

Figure 1.1.1 The Process of QR algorithm

The whole process of the QR algorithm is figure 1.1.1, a matrix is transformed to an upper Hessenberg matrix at beginning. And, the Hessenberg matrix is put into a loop which would be stopped when a matrix is convergent. The condition of convergence is that the diagonal elements in a new matrix are the same as previous matrix. In the loop, it does QR factorization and create a new matrix with R and Q. Also, eigenvectors are calculated during this process. If the size of a matrix is n and the number of loops before convergence is m. The time complexity of the Hessenberg transformation is $\mathrm{O(n^3)}$. It eliminates lower-triangular elements to zeros in matrix

which is $O(n^2)$, and the process of elimination is $O(n)$. Moreover, the time complexity of the loop is $O(mn^2)$. Let the loop convergent after m times, and it does QR factorization every time. In the calculation of new matrix, it contains the elimination of lower triangular elements and calculation of eigenvectors. Both processes can be done together. Because there is a Hessenberg matrix in the loop, there are only n-1 lower triangular elements in it, and the process of elimination is $O(n)$, the time complexity of QR algorithm is $O(n^3) + O(mn^2)$.The calculation of Q matrices and eigenvectors can be done at the same time. The QR factorization can be done in parallel. However, implementation of QR algorithm needs to do elimination of elements one by one currently because each elimination can affect others.

```
while not convergence:
    shift_val = a[(dim-1)*dim + (dim-1)]

    if QR_checkdiagonal(a, dim):
        shift_val -= 1

    for i in range(0, dim):
        old[i] = a[i*dim+i]
        a[i*dim+i] -= shift_val

    for i in range(0, dim-1):
        j = i + 1
        s,c = Calc_Givens(a,dim,i,j, 0)
        QR_doGAG(a, dim, s, c, i, j)
        QR_doAG(eiv2, dim, s, c, i, j)

    for i in range(0, dim):
        a[i*dim+i] += shift_val

    convergence = QR_convergence(a, old, dim)
```

Figure 1.1.2 Loops of the QR Algorithm

As figure 1.1.2, the loop subtracts the diagonal elements in a matrix with a shift value first. And, it does the QR factorization. QR factorization eliminates lower-triangular elements in the matrix, it needs to calculate the value of sin or cos for every lower-triangular element. Also, it calculates a new matrix and eigenvectors when having the Givens matrices with specific sin and cos for each lower-triangular element. At the end, this loop adds shift value back to diagonal elements and checks whether a matrix is convergent. Figure 1.1.3 shown an approach of calculating the values of sin and cos for QR algorithm. This approach is mentioned in [7]. The Calc_Geivens gets 2 elements which are diagonal element and the element under the diagonal element at beginning, and it calculate the value of sin and cos.

```
def Calc_Givens(m, dim, p, q, diff):
    e1 = m[p*dim+(p-diff)]
    e2 = m[q*dim+(p-diff)]

    c = 0
    s = 0
    r = 0
    t = 0
    if e2 == 0:
        c = CopySign(1, e1)
        s = 0
        r = abs(e1)
    elif e1 == 0:
        c = 0
        s = -CopySign(1,e2)
        r = abs(e2)
    elif abs(e2) > abs(e1):
        t = e1/e2
        u = CopySign(sqrt(1+t*t), e2)
        s = -1/u
        c = -s*t
        r = e2*u
    else:
        t = e2/e1
        u = CopySign(sqrt(1+t*t), e1)
        c = 1/u
        s = -c*t
        r = e1*u

    return s, c
```

Figure 1.1.3 The Calculation of Sin and Cos for The QR Algorithm [7]

1.1.3 Jacobi Eigenvalue Method

The Jacobi eigenvalue method [8] is used for real symmetric matrices. Let a real symmetric matrix A_k is calculated with a Givens rotation matrix G_k to create a new symmetric matrix A_{k+1}. By applying the function, Jacobi method can eliminate non-diagonal elements in a matrix. As a result, there will be only diagonal elements in a matrix. A matrix is convergent. Eigenvalues are diagonal elements in the convergent matrix.

$$\mathrm{A_{k+1}} = \mathrm{G_k} A_k G_k^T$$

Let a 4x4 real symmetric matrix and $G_k = \mathrm{G}(2,4,\theta)$, the new $\mathrm{A_{k+1}}$ is

$$A_{k+1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c & 0 & -s \\ 0 & 0 & 1 & 0 \\ 0 & s & 0 & c \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c & 0 & s \\ 0 & 0 & 1 & 0 \\ 0 & -s & 0 & c \end{bmatrix}$$

$$= \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ ca_{21} - sa_{41} & ca_{22} - sa_{42} & ca_{23} - sa_{43} & ca_{24} - sa_{44} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ sa_{21} + ca_{41} & sa_{22} + ca_{42} & sa_{23} + ca_{43} & sa_{24} + ca_{44} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c & 0 & s \\ 0 & 0 & 1 & 0 \\ 0 & -s & 0 & c \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c & 0 & -s \\ 0 & 0 & 1 & 0 \\ 0 & s & 0 & c \end{bmatrix} \begin{bmatrix} a_{11} & ca_{12} - sa_{14} & a_{13} & sa_{12} + ca_{14} \\ a_{21} & ca_{22} - sa_{24} & a_{23} & sa_{22} + ca_{24} \\ a_{31} & ca_{32} - sa_{34} & a_{33} & sa_{32} + ca_{34} \\ a_{41} & ca_{42} - sa_{44} & a_{43} & sa_{42} + ca_{44} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11} & ca_{12} - sa_{14} & a_{13} & sa_{12} + ca_{14} \\ ca_{21} - sa_{41} & a'_{22} & ca_{23} - sa_{43} & a'_{24} \\ a_{31} & ca_{32} - sa_{34} & a_{33} & sa_{32} + ca_{34} \\ sa_{21} + ca_{41} & a'_{42} & sa_{23} + ca_{43} & a'_{44} \end{bmatrix}$$

where

$$a'_{22} = c(ca_{22} - sa_{42}) - s(ca_{24} - sa_{44}) = c^2 a_{22} - csa_{42} - sca_{24} + s^2 a_{44}$$
$$a'_{44} = s(sa_{22} + ca_{42}) + c(sa_{24} + ca_{44}) = s^2 a_{22} + csa_{42} + csa_{24} + c^2 a_{44}$$
$$a'_{24} = 0 = a'_{44}$$

Looking at elements in A_{k+1} generally, a'_{ii}, a'_{jj} and a'_{ij} can be calculated with functions below.

$$a'_{ii} = c^2 a_{ii} - 2csa_{ij} + s^2 a_{jj}$$
$$a'_{jj} = s^2 a_{ii} + 2csa_{ij} + c^2 a_{jj}$$
$$a'_{ij} = a'_{ji} = (c^2 - s^2)a_{ij} + cs(a_{ii} - a_{jj})$$

After Givens rotation, a'_{ij} and a'_{ji} in A_{k+1} should be eliminated to 0.

$$a'_{ij} = a'_{ji} = (c^2 - s^2)a_{ij} + cs(a_{ii} - a_{jj}) = 0$$

The value of sin and cos which are used for Givens rotation can be calculated with original elements in A_k

$$\frac{a_{jj} - a_{ii}}{a_{ij}} = \frac{c^2 - s^2}{cs} = \frac{1 - \left(\frac{s}{c}\right)^2}{\frac{s}{c}} = \frac{(1 - t^2)}{t} = 2w \quad i.e. t^2 + 2wt - 1 = 0$$

First, there are some definitions for this process.

$$t = tan\theta = \frac{s}{c} = \frac{sin\theta}{cos\theta}$$

$$w = \frac{a_{jj}-a_{ii}}{2a_{ij}} = \frac{c^2-s^2}{2cs} = \frac{\cos(2\theta)}{\sin(2\theta)} = \cot(2\theta)$$

With these definitions, t can be calculated. $t = -w \pm \sqrt{w^2+1}$, if w < 0, t is set to $-w - \sqrt{w^2+1}$, or t is $-w + \sqrt{w^2+1}$ for precision. As a result, $\sin\theta$ and $\cos\theta$ are

$$s = sin\theta = \frac{tan\theta}{\sqrt{1+\tan^2\theta}} = \frac{t}{\sqrt{1+t^2}}$$

$$c = cos\theta = \frac{tan\theta}{\sqrt{1+\tan^2\theta}} = \frac{t}{\sqrt{1+t^2}}$$

Finding the largest non-diagonal element in matrix can decide i and j. The element is called pivot. Jacobi method is similar to QR algorithm, it does the transformation repeatedly until it gets a final matrix. Eigenvalues are the diagonal elements in the final matrix. Also, eigenvectors can be calculated by Givens matrices.

$$EVs = G_1^T G_2^T \dots G_n^T$$

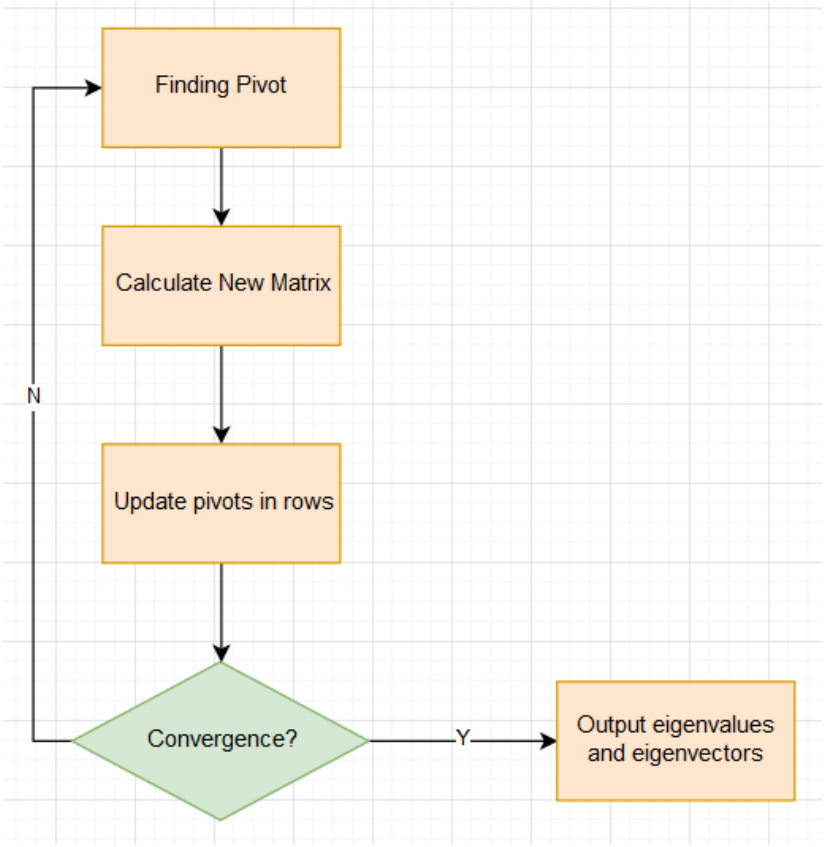

Figure 1.1.4 The Process of Jacobi Eigenvalue Method

The process of Jacobi method, it needs to find a pivot for generating sin and cos for a Givens matrix for rotation. In finding pivot, it looks for largest non-diagonal

element, the indexes of the largest non-diagonal element would be the pair for Givens matrix. After gaining the Givens matrix, it is used for calculating new matrix and eigenvectors. Also, the eigenvalues are the diagonal elements in final matrix. The time complexity of Jacobi method is $O(mn)$. In the loop of Jacobi method, it only affects two rows and columns each time. Also, matrices are symmetric. In terms of it only needs to calculate elements in two rows. As a result, the process of update pivots in rows can be done simultaneously. Moreover, just like QR algorithm, the calculation new matrix involves the elimination of non-diagonal elements and calculation of eigenvectors. The time complexity of finding largest pivot is O(number of rows), because there is an array which records the largest non-diagonal element of each row.

Figure 1.1.5 shown the loop for the Jacobi method. At the beginning of the loop, it finds the largest pivot element inside a matrix. With the largest element in a matrix, the loop calculates the value sin and cos for eliminating the element. After that, it can calculate new matrix and eigenvectors. Each iteration only eliminates one non-diagonal element which is not similar to QR algorithm. Figure 1.1.6 shown how to compute the sin and cos in Jacobi method. The function Calc_Givens() gets element a_{ii} , a_{jj} and a_{ij} at first. And, it calculates the value of sin and cos with equations that have been mentioned above.

```
while not convergence:
    count = count + 1
    for i in range(0,dim):
        old[i] = a[i*dim+i]
    k = 0
    for i in range(1, dim-1):
        x = pvspos[i]
        y = pvspos[k]
        if abs(a[x]) > abs(a[y]):
            k = i

    l = pvspos[k] - k*dim
    if abs(a[k*dim+l]) > 0:
        s,c = Calc_Givens(a, dim, k, l)
        JCB_doGAG(a, dim, s, c, k, l)
        JCB_doAG(evs, dim, s, c, k, l)

        update_pivot_onerow(a, pvspos, k, dim)
        if l != (dim-1):
            update_pivot_onerow(a, pvspos, l, dim)

    convergence = JCB_convergence(a, old, dim)
```

Figure 1.1.5 The Loop for The Jacobi Method

```
def Calc_Givens(m, dim, p, q):
    e1 = m[p*dim+p]
    e2 = m[q*dim+p]
    e3 = m[q*dim+q]

    w = (e3 - e1) / (2*e2)
    t = 0.0
    if w < 0:
        t = -w - sqrt((w*w)+1)
    else:
        t = -w + sqrt((w*w)+1)
    s = t / sqrt(1+(t*t))
    c = 1 / sqrt(1+(t*t))

    return s,c
```

Figure 1.1.6 The Calculation of Sin and Cos for Jacobi method

1.2 Resources and Techniques on FPGA

FPGA has different resources like Block RAM (BRAM), LUT (LookUp Table), DSP, FF (Flip-Flop) and URAM for simulating an ASIC. BRAM is internal block RAM inside FPGA, the transferring speed of BRAM is faster than DDR memory on board. LUT is 'LookUp Table' which is used for mechanisms like truth table. FF is 'Flip-Flop' which is used as a form of storage like storing results from LUT. LUT and FF are the basic module for FPGA to generate circuit. When FPGA performs float/double computation like addition, subtraction and multiplication, it utilizes DSP48 for computation. DSP48 is arithmetic logic unit (ALU) inside FPGA. These resources are used for simulating circuit on ASIC. There are some techniques which are useful for developing project on FPGA.

1.2.1 Techniques for Implementing Code on FPGA

The processes of developing programs on CPU and FPGA are different, some behavior which is normal on CPU, but it affects the performance on FPGA. For example, in C programming language, programmers can add '{0}' after an declaimed array like 'int array[SIZE] = {0}', however, it would make FPGA to do unnecessary initialization. Arrays on FPGA are put in BRAM, and the space of arrays would be set to zeros. So, there is no need to do zero initialization in FPGA.

Furthermore, the data transferring speed of BRAM on FPGA is far faster than DDR RAM. If there is no need to compute large dataset, programmers can put datasets in BRAM. But, the size of BRAM is much smaller than DDR RAM, considering large datasets, DDR RAM should be used for those datasets. Also, programmers should

figure out an approach to decrease the time of accessing DDR memory like implementing cache mechanism with BRAM for decreasing the time of accessing DDR memory. If the number of data transfer is decreased by caching mechanism, the total latency is decreased as well.

In Vivado HLS tool, independent operations are executed at the same cycle. Like figure 1.2.1, the operations in functionA are independent, in synthesis process, HLS tool would make FPGA to execute them simultaneously. This is important concept for unrolling for loops which will be mentioned later in this section. Also, whenever FPGA allocates space in BRAM, the space will be set to zero, there is no need to reset the space with zeros again.

```
void functionA()
{
    a = b + c;
    d = e + f;
}
```

Figure 1.2.1 Example of Independent Operation

Moreover, HLS tool provides many useful pragmas [32] like pipeline, unroll and so on. Those pragmas can be used for different purposes such as loop optimization, kernel optimization, array optimization, pipeline and so on. The following content introduces the pragmas that are used for developing this project.

HLS pipeline [24] is an important technique for designing project on FPGA. Figure 1.2.2 shown that a loop can be done with pipeline when adding the pragma. Also, each iteration is executed after previous iteration with specific interval.

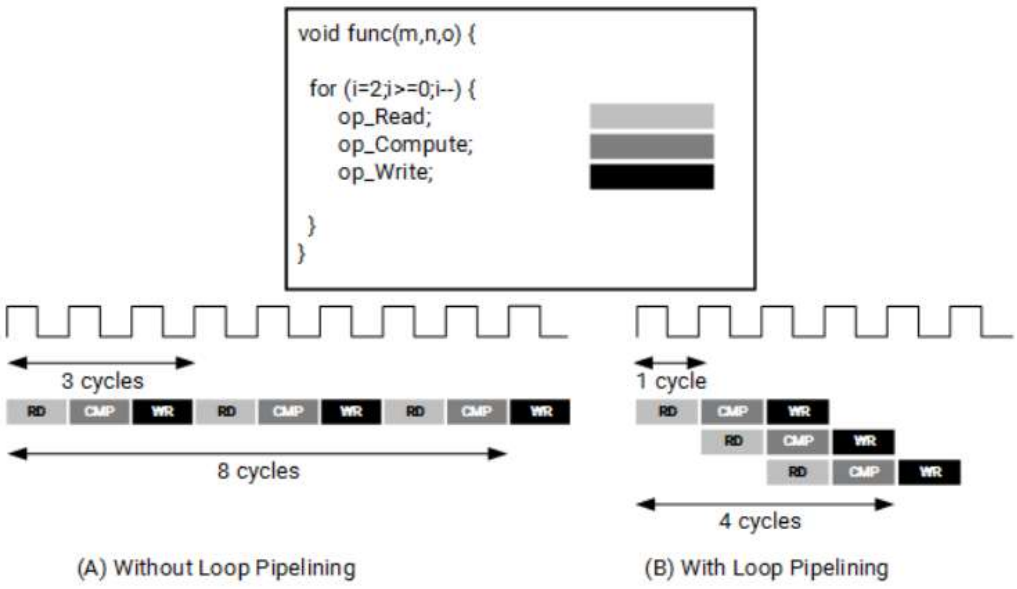

Figure 1.2.2 Pipelining on FPGA [24]

After doing pipeline in a for loop, the total cycle for a loop in func is decreased from 8

cycles to 4 cycles. And each iteration is executed after previous iteration with interval 1 cycle. HLS pipeline is useful for burst reading/writing. Burst reading/writing means read/write data from/to DDR memory with less cycles. In order to make the interval as short as possible, programmers should try to remove dependency in loops. If there are dependencies inside loops, HLS tool will increase interval of pipeline to keep dependencies safe.

HLS unroll [25] is a pragma which unrolls for-loops. It is useful when the inter dependency is not exist inside a loop. If there is no interdependency in a loop, it means the loop can be pipelined with smallest interval 1 cycle. Also, by unrolling the loop, some iterations in the loop can be done at the same cycle. Figure 1.2.3 shown the example which combined pipeline and unroll.

```
for (i=0; i<dim; i++) {
#pragma HLS LOOP_TRIPCOUNT min=2 max=500
#pragma HLS pipeline
#pragma HLS dependence array inter false
#pragma HLS unroll factor=2
    temp1[i] = evs[i*dim+x] * cos;
    temp2[i] = evs[i*dim+x] * sin;
}
```

Figure 1.2.3 Example for Combining Pipeline and Unroll

In the for-loop, the temp1 and temp2 are independent. By utilizing pipeline, the for-loop will be done in pipeline. Also, the factor of HLS unroll is 2 which means every 2 iterations will be done at the same cycle. As a result, this for-loop will take less cycles than a loop without unrolling. The default setting for unrolling a loop is all if there is no factor setting in HLS unroll pragma. However, for non-static loop, HLS is unable to unroll it. So, adding a factor is important for this type of loops.

HLS Loop_tripcount [26] is a pragma for estimating latency for loops, it doesn't affect implementations. But it is helpful for developers to understand the latency and interval of each function in implementations on FPGA.

HLS dependency [30] is used for indicating the data dependency inside loops, by making the dependency clear it can help HLS tool to utilize shortest interval of pipeline. Also, if there is no dependency in loops, this pragma can be used for indicating by adding false in dependency pragma.

In order to develop this project, it needs these techniques to convert C host codes to implementations on FPGA.

1.2.2 Fixed Type on FPGA

In Vivado Design Suite User Guide [27] mentioned a datatype ap_fixed. The ap_fixed type can let users to decide how many bits for integer parts and how many bits for decimal part.

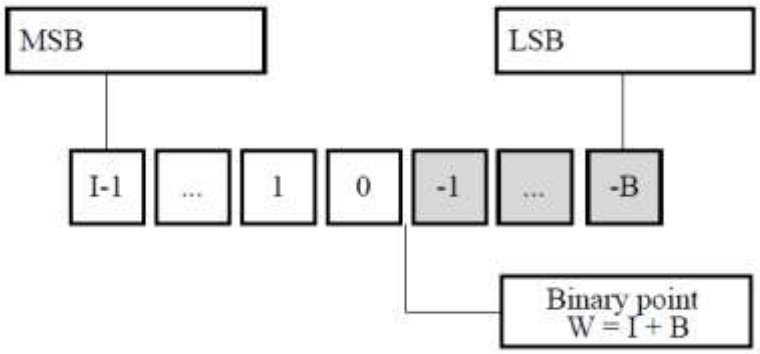

Figure 1.2.4 Fixed Point Data Type [27]

Take ap_fixed<16,8> as an example. 16 means it has 16 bits and 8 means which has 7 bits for the value before decimal point, 1 bit for sign and 8 bits for value after decimal point. So, the range of integer part is from -128 to 127. Also, the value after decimal point should larger than or equal to 0.00391 (calculated by 1/255). If I set 0.003 to the fixed type, it would become zero. Also, if I set 0.005 to fixed type, it would be 0.00391 instead of 0.005 because 2/255 is about 0.00781. So, in practical, setting an adequate number of bits for ap_fixed type is important. However, in this project, it didn't utilize this fixed type. This will be discussed in section 5.1.

1.3 Aims

This project aims to design custom intellectual property (IP) [17] in FPGA to accelerate the process of finding eigenvalues and eigenvectors in order to determine the usability of FPGA technology in HPC environments. Moreover, this project focuses on finding eigenvalues and eigenvectors for real symmetric matrices in order to compare these two algorithms fairly.

1.4 Data Requirement

This project needs a number of small matrices for preliminary test as we develop and test our implementation, and a number of large matrices for evaluation. The testing data sets were generated by programs, it is not related to any data in real world.

2. Related Work

Aslan [9] proposed an implementation based on the QR algorithm and Givens rotation. It utilized COordinate Rotation DIgital Computer (CORDIC) [19] and fixed point in order to achieve the best clock on FPGA. CORDIC is also known as Volder's algorithm. CORDIC is an algorithm for computing hyperbolic and trigonometric functions in easier on hardware like FPGA. This proposal focuses on 4x4 real symmetric matrices. This project didn't apply CORDIC and fixed type. F. Rotella [16] introduced that the process of QR factorization can be done in parallel. The approach is an extension of Household transformation [20]. As a result, QR factorization can be calculate with many blocks at the same time. Lin [10] proposed an implementation which is based on Approximate Jacobi method [21] for generalized symmetric matrices. Lin implemented Approximate Jacobi method on FPGA for general symmetric matrices, also, this paper proposed a special algorithm 'Algebraic Method' for 3x3 symmetric matrices. Moreover, this paper relied on CORDIC to get the value of trigonometric functions. For evaluation, this paper didn't provide exact execution time but formula for calculating execution time. It is impossible to compare the result of this project with the result of Lin. Unfortunately, the time of development for this project is not enough for implementing other approaches for comparison. fBLAS [28] is an open-sourced implementation for HLS [13] tools to develop Basic Linear Algebra Subprograms (BLAS) [29] on FPGA. It aims to make the process of developing and porting on FPGA easier on FPGA.

3. Implementation

This project aims to develop code on different platforms. PYNQ Z2 [11] is used for doing proof of concept. And code in PYNQ is ported to Xilinx Alveo U200 [31] platform. In section 3.1, it introduces the specification of platforms. The following subsections describe how to develop this project on different platforms. For implementations on PYNQ and Alveo, they utilized single-precision data type for matrices.

3.1 Platforms

PYNQ Z2 [11] is a development board which has a System on a Chip (SoC) [12] on it. The SoC involves 650 MHz dual-core ARM Cortex A9 and FPGA core which is equivalent to Artix-7 FPGA. It has 630 KB block RAM, 220 DSP slice, 106400 FFs and 53200 LUTs. It doesn't have URAM. Moreover, host part and FPGA part sharing the same DDR memory.

For the implementation for HPC environment, "livfpga" is an HPC environment at the University of Liverpool. It has one Xilinx Alveo U200 [31] card and a Xeon E5 649 which has 2.53 GHz clock. U200 has 7947 KB BRAM, 5867 DSP slice, 1831K FFs and 892K LUTs. Also, U200 has default data clock 300 MHz and default kernel clock 500 MHz. Also, it has 4 DDR banks, each back has 4 GB. U200 can be installed on PCI-e 3.0 and PCI-e 2.0.

3.2 Implementation on PYNQ Z2

PYNQ contains Python APIs for users to control FPGA easily. For FPGA part, users need Vivado HLS [13] to design their own IPs [17] for FPGA. After designing IPs, those IPs are utilized for generate overlays in Vivado Design Suite [14]. Vivado HLS supports High-Level Synthesis in terms of users can design hardware logic for FPGA with high level language like C/C++ or OpenCL. The definitions of HLS interfaces have a bundled control port for setting parameters to FPGA, and some bundled memory ports for accessing DDR memory. If users use C/C++ for developing, at beginning of top function, users need to define ports with pragmas. For example, the definition of ports for the QR algorithm and Jacobi method is figure 3.2.1.

```
void QR_Symm(M_DATA_TYPE *in_matrix,
             M_DATA_TYPE *out_matrix1,
             M_DATA_TYPE *out_matrix2,
             int32_t dim) {
#pragma HLS INTERFACE m_axi port = in_matrix offset = slave bundle = gmem
#pragma HLS INTERFACE m_axi port = out_matrix1 offset = slave bundle = gmem1
#pragma HLS INTERFACE m_axi port = out_matrix2 offset = slave bundle = gmem1
#pragma HLS INTERFACE s_axilite port = in_matrix bundle = control
#pragma HLS INTERFACE s_axilite port = out_matrix1 bundle = control
#pragma HLS INTERFACE s_axilite port = out_matrix2 bundle = control
#pragma HLS INTERFACE s_axilite port = dim bundle = control
#pragma HLS INTERFACE s_axilite port = return bundle = control
```

Figure 3.2.1 The Definitions of Ports in Vivado HLS

The IP module in Vivado Design Suite would look like Figure 3.2.2. It has an s_axi_control port for host part and some m_axi_gmem* ports for accessing DDR memory.

Figure 3.2.2 IP of QR Algorithm in Vivado Design Suite

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	15	0	1418	-
FIFO	-	-	-	-	-
Instance	24	47	15193	21194	0
Memory	1	-	0	0	0
Multiplexer	-	-	-	1679	-
Register	0	-	1693	56	-
Total	25	62	16886	24347	0
Available	280	220	106400	53200	0
Utilization (%)	8	28	15	45	0

Figure 3.2.3.1 Resource Utilization of QR

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	869	-
FIFO	-	-	-	-	-
Instance	44	89	21384	36412	0
Memory	16	-	0	0	0
Multiplexer	-	-	-	2225	-
Register	0	-	1885	128	-
Total	60	89	23269	39634	0
Available	280	220	106400	53200	0
Utilization (%)	21	40	21	74	0

Figure 3.2.3.2 Resource Utilization of Jacobi

Those s_axilite ports are used for setting addresses or values, and m_axi ports are used for accessing DDR memory on board. Those m_axi ports are able to be bundled with different memory bus which enables FPGA to access different part of DDR memory at the same time like in_matrix is bundled with gmem and out_matrices are bundled with gmem1. Choosing different DDR bank for each port will be discussed in later section. PYNQ has an AXI_ACP port on it which enables ARM core and FPGA core to share the same DDR memory. After building IPs, Vivado HLS generate report like figure 3.2.3.1 and figure 3.2.3.2 which contains usage of resources like BRAM, DSP, FF, LUT and URAM.

After IPs are created, users can start to generate overlay in Vivado Design Suite. The overlay in Vivado Design Suite is Figure 3.2.4.

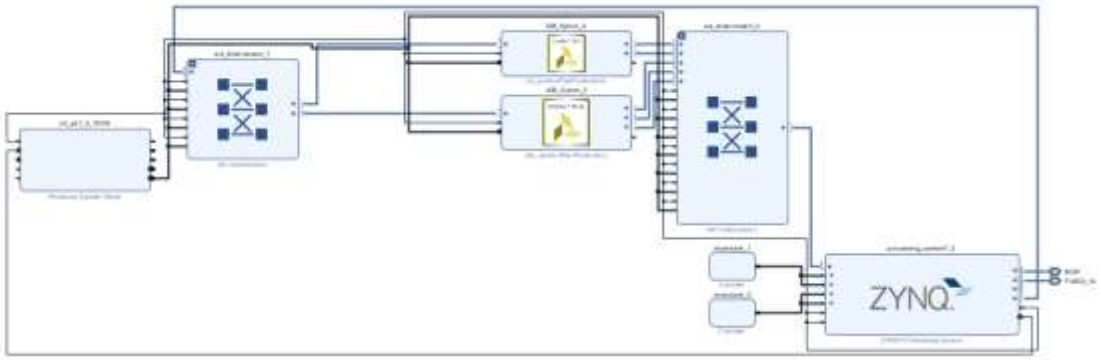

Figure 3.2.4 The overlay in Vivado Design Suite

IPs are connected by interconnectors, and those interconnectors are connected to ZYNQ core which is ARM core. After finishing the overlay, it can be used for creating bit stream files. Bit stream file is loaded by Python APIs which means users can write host code in Python and utilize those APIs to control FPGA part. Firstly, users can write a driver in Python host code like Figure 3.2.5.

```python
class QRSymmDriver(DefaultIP):
    def __init__(self, description):
        super().__init__(description=description)

    bindto = ['xilinx.com:hls:QR_Symm:1.0']

    @property
    def status(self):
        return self.read(0x00)

    @status.setter
    def status(self, value):
        self.write(0x00, value)

    @property
    def input(self):
        return self.read(0x10)

    @input.setter
    def input(self, addr):
        self.write(0x10, addr)

    @property
    def output1(self):
        return self.read(0x18)

    @output1.setter
    def output1(self, addr):
        self.write(0x18, addr)

    @property
    def output2(self):
        return self.read(0x20)

    @output2.setter
    def output2(self, addr):
        self.write(0x20, addr)
```

Figure 3.2.5 Part of Driver in Host Code for QR Algorithm

The register addresses in setters and readers are defined in a header file when generated IPs in Vivado HLS. A driver needs to set correct addresses in order to control IP in FPGA part properly.

```
#define XQR_SYMM_CONTROL_ADDR_AP_CTRL          0x00
#define XQR_SYMM_CONTROL_ADDR_GIE              0x04
#define XQR_SYMM_CONTROL_ADDR_IER              0x08
#define XQR_SYMM_CONTROL_ADDR_ISR              0x0c
#define XQR_SYMM_CONTROL_ADDR_IN_MATRIX_DATA   0x10
#define XQR_SYMM_CONTROL_BITS_IN_MATRIX_DATA   32
#define XQR_SYMM_CONTROL_ADDR_OUT_MATRIX1_DATA 0x18
#define XQR_SYMM_CONTROL_BITS_OUT_MATRIX1_DATA 32
#define XQR_SYMM_CONTROL_ADDR_OUT_MATRIX2_DATA 0x20
#define XQR_SYMM_CONTROL_BITS_OUT_MATRIX2_DATA 32
#define XQR_SYMM_CONTROL_ADDR_DIM_DATA         0x28
#define XQR_SYMM_CONTROL_BITS_DIM_DATA         32
```

Figure 3.2.6 Register Addresses of Each Port

By reading 0x00, users can know the status of FPGA. There are 5 different status for FPGA. The status ap_idle means FPGA is idle now, users can write ap_start which makes FPGA start to work. After job is done by FPGA, it sets ap_done to status.

```
Status 0x01 for ap_start (read/write/COH)
Status 0x02 for ap_done (read/COR)
Status 0x04 for ap_idle (read)
Status 0x08 for ap_ready (read)
Status 0x80 for ap_reset (read/write)
```

Figure 3.2.7 Status for FPGA

```
if (QR_Symm.status & ap_idle) == ap_idle:
    status = 0
    QR_Symm.dimension = dim
    QR_Symm.input = in_buffer.physical_address
    QR_Symm.output1 = out_buffer1.physical_address
    QR_Symm.output2 = out_buffer2.physical_address
    QR_Symm.status = ap_start
    while (status & ap_done) != ap_done:
        status = QR_Symm.status
```

Figure 3.2.8 Interaction between Host and FPGA

For example, in figure 3.2.8, it checks whether FPGA is idle. If it is idle, it sets addresses and dimension to FPGA. After setting those parameters, it writes ap_start to FPGA and wait for ap_done. The ap_done signal will be unset once host read status. The steps of executing bit stream files on PYNQ are list below.

1. Host part allocates input and output buffers
2. Host sets the dimension of matrix, physical addresses of space to FPGA through control ports. FPGA can access the same DDR RAM by AXI_ACP port at host.
3. Host write ap_start signal to FPGA
4. FPGA processes data in input matrix and put results to output space and sends ap_done signal to host.

5. Host read data from output buffer

3.3 Implementation on Xilinx Alveo

In order to develop projects for Alveo, SDAccel [15] is needed for this purpose. SDAccel is an IDE which contains Vivado Design Suite and Vivado HLS in it. SDAccel supports C/C++ for host part, and C/C++, OpenCL and RTL for FPGA part. Alveo cards are connect with Host by PCI Express interface. There are different kernels which were implemented on Alveo. They are Jacobi method with DDR RAM and BRAM, and the QR algorithm with DDR RAM and BRAM.

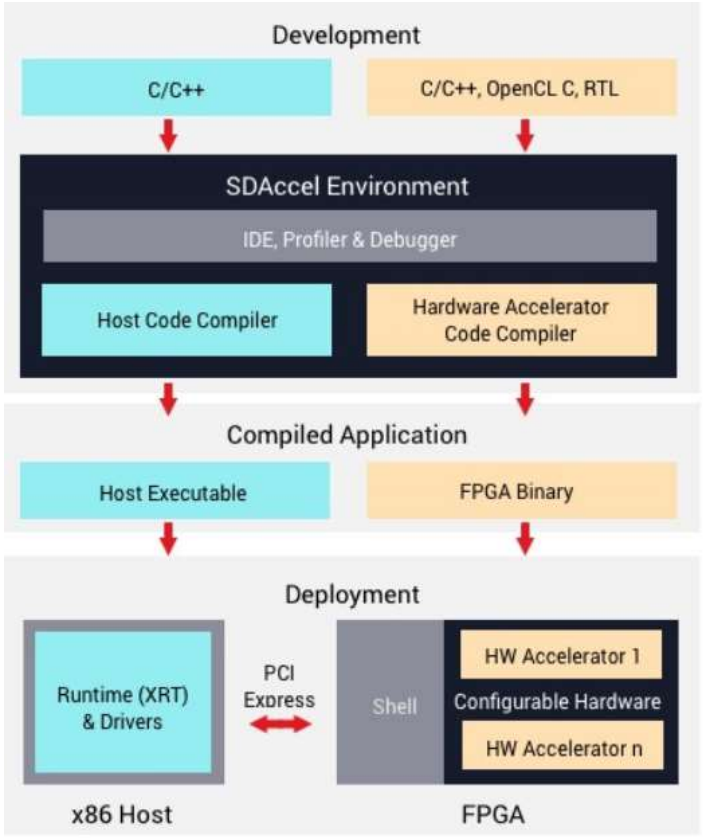

Figure 3.3.1 The Architecture of SDAccel [15]

There are three different build mode in SDAccel. They are Emulation-SW, Emulation-HW and System respectively. Emulation-SW enables users to check if their code is buildable. Also, users can debug their code within this mode. Emulation-HW verifies hardware design that is executed in FPGA and generates reports. Finally, in order to build bit stream files, System build is required. In SDAccel, users don't need to worry how to generate overlays like utilizing Vivado Design Suite when developing project on PYNQ. SDAccel handles it for users, but users have to follow the restriction of defining ports. IPs can only have one bundled control port, and an amount of

memory ports or streaming ports. In project settings of SDAccel, users need to create binary containers and add top functions into them.

A bitstream file needs at least one kernel in it. Like figure 3.3.2, there are two bitstream files qr_binary and jcb_binary. Both has an individual kernel QR_Symm and JCB_Symm. QR_Symm and JCB_Symm are called top functions.

Figure 3.3.2 Binary Containers in Project Setting Page

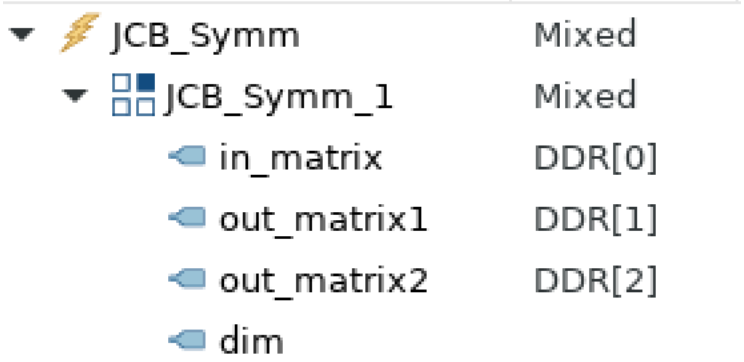

Figure 3.3.3 Selecting Different DDR Banks for Ports

SDAccel enables FPGA to assign different DDR banks for each memory ports. Like the memory ports in_matrix, out_matrix1 and out_matrix2. They can be assigned to DDR[0], DDR[1] and DDR[2] respectively. By assigning DDR banks to each memory port, it makes FPGA read/write DDR memory with those memory ports simultaneously.

The resource usage for kernels on U200 are listed below.

Area Information

Compute Unit	Kernel Name	Module Name	FF	LUT	DSP	BRAM
JCB_Symm_1	JCB_Symm	update_pivot_in_row	1536	2518	0	38
JCB_Symm_1	JCB_Symm	JCB_findLargest	1570	2347	0	38
JCB_Symm_1	JCB_Symm	JCB_checkCache	3073	2894	16	0
JCB_Symm_1	JCB_Symm	JCB_Calc_Givens	5171	3971	7	0
JCB_Symm_1	JCB_Symm	JCB_doCheckCacheandGivens	8312	6996	23	0
JCB_Symm_1	JCB_Symm	JCB_doGAG	4752	3384	32	76
JCB_Symm_1	JCB_Symm	JCB_doAG	4586	2840	32	76
JCB_Symm_1	JCB_Symm	JCB_doCalc	9342	6247	64	152
JCB_Symm_1	JCB_Symm	JCB_checkRemainPivotValue	135	269	0	0
JCB_Symm_1	JCB_Symm	JCB_update_pivot_double_flow	3276	5346	0	76
JCB_Symm_1	JCB_Symm	JCB_update_pivot_double	3448	5793	0	114
JCB_Symm_1	JCB_Symm	JCB_update_pivot_single_flow	1707	2819	0	38
JCB_Symm_1	JCB_Symm	JCB_doFinal	1675	1700	8	0
JCB_Symm_1	JCB_Symm	JCB_Symm	32171	34570	95	576

Figure 3.3.4.1 Resource Usage of Jacobi Method with DDR RAM on FPGA

Area Information

Compute Unit	Kernel Name	Module Name	FF	LUT	DSP	BRAM
JCB_Symm2_1	JCB_Symm2	update_pivot_in_row_1	1628	2626	0	4
JCB_Symm2_1	JCB_Symm2	JCB_findLargest	1565	2347	0	4
JCB_Symm2_1	JCB_Symm2	JCB_Calc_Givens	5171	3971	7	0
JCB_Symm2_1	JCB_Symm2	JCB_doGAG	6880	4958	43	12
JCB_Symm2_1	JCB_Symm2	JCB_doAG	5908	3346	48	8
JCB_Symm2_1	JCB_Symm2	JCB_doCalc	12792	8327	91	20
JCB_Symm2_1	JCB_Symm2	update_pivot_in_row	1563	2518	0	4
JCB_Symm2_1	JCB_Symm2	JCB_checkRemainPivotValue	135	269	0	0
JCB_Symm2_1	JCB_Symm2	JCB_update_pivot_double	2443	3812	8	10
JCB_Symm2_1	JCB_Symm2	JCB_update_pivot_single_2	2409	3582	8	8
JCB_Symm2_1	JCB_Symm2	JCB_doFinal_entry3215	2	55	0	0
JCB_Symm2_1	JCB_Symm2	JCB_doFinal_Loop_1_proc	173	221	0	0
JCB_Symm2_1	JCB_Symm2	JCB_doFinal_Loop_2_proc	440	282	4	0
JCB_Symm2_1	JCB_Symm2	JCB_doFinal	644	907	4	0
JCB_Symm2_1	JCB_Symm2	JCB_Symm2	31083	30388	130	949

Figure 3.3.4.2 Resource Usage of Jacobi Method with BRAM on FPGA

Area Information

Compute Unit	Kernel Name	Module Name	FF	LUT	DSP	BRAM
QR_Symm_1	QR_Symm	QR_Calc_Givens	3803	3729	5	0
QR_Symm_1	QR_Symm	QR_doCalc_entry1310	2	127	0	0
QR_Symm_1	QR_Symm	QR_doGAG	4695	3674	24	114
QR_Symm_1	QR_Symm	QR_doAG	3673	2955	16	114
QR_Symm_1	QR_Symm	QR_doCalc	8439	7455	40	228
QR_Symm_1	QR_Symm	QR_checkdiagonal	1846	1778	2	19
QR_Symm_1	QR_Symm	QR_convergence	1532	1408	2	19
QR_Symm_1	QR_Symm	QR_Symm	23391	22195	69	289

Figure 3.3.4.3 Resource Usage of QR Algorithm with DDR RAM on FPGA

Area Information

Compute Unit	Kernel Name	Module Name	FF	LUT	DSP	BRAM
QR_Symm2_1	QR_Symm2	QR_Calc_Givens	3803	3729	5	0
QR_Symm2_1	QR_Symm2	QR_doGAG	4885	3364	40	6
QR_Symm2_1	QR_Symm2	QR_doAG	2292	1709	16	4
QR_Symm2_1	QR_Symm2	QR_doCalc	7181	5096	56	10
QR_Symm2_1	QR_Symm2	QR_checkdiagonal	1437	1453	2	2
QR_Symm2_1	QR_Symm2	QR_convergence	1123	1074	2	2
QR_Symm2_1	QR_Symm2	QR_doFinal_entry2114	3	82	0	0
QR_Symm2_1	QR_Symm2	QR_doFinal_Loop_1_proc	173	221	0	0
QR_Symm2_1	QR_Symm2	QR_doFinal_Loop_2_proc	440	282	4	0
QR_Symm2_1	QR_Symm2	QR_doFinal_Block_proc	583	607	0	0
QR_Symm2_1	QR_Symm2	QR_doFinal	1238	1659	4	0
QR_Symm2_1	QR_Symm2	QR_Symm2	20370	18251	97	913

Figure 3.3.4.4 Resource Usage of QR Algorithm with BRAM on FPGA

The usage of resources on U200 were similar to PYNQ Z2 except BRAM. Because U200 has more space of BRAM than PYNQ, so the definition of size in implementations on U200 was larger than PYNQ. Also, the implementations with BRAM utilized more BRAM resources than the implementations with DDR RAM.

Moreover, SDAccel would change the clock for bitstream files if the bitstream cannot utilize the default clock on Alveo. In Xilinx devices, a LUT is coupled with a register. Users can connect output to the LUT to a register, or they can connect output to another LUT in order to make a bigger logic circuit. Whenever users add a LUT in the LUT chain, it increases the delay of the route. In terms of it cannot meet the target clock 300 MHz.

After finishing these steps, users can build projects in different emulation. There are some optional steps that can be added into linkers.

Figure 3.3.5 Optional Compiler Options for XOCC

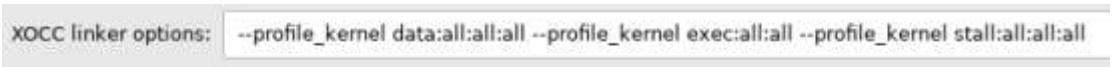

Figure 3.3.6 Optional Linker Options for XOCC

By adding '—profile_kernel data:all:all:all', '—profile_kernel exec:all:all' and '—profile_kernel stall:all:all:all' when linking bit stream files, host program is able to generate analysis files after executing bit stream files. For profiling stall information, users have to add '—profile_kernel stall:all:all:all' in time of compilation. These options won't affect the running time of FPGA kernel significantly, but host program needs

more time to terminate at the end. For evaluation, it only recorded the execution time of FPGA kernels rather than whole running time of host program.

The steps for executing bit stream files on Alveo are listed below.

1. Host reads bit stream file
2. Host initials FPGA with bit stream which is read before
3. Host part allocates input and output buffers
4. Host creates buffers in DDR RAM of FPGA, and it copies input data to the RAM of FPGA
5. FPGA processes data from input buffer and put results to output buffer
6. Host reads data from output buffers in FPGA

4. Result and Evaluation

In this chapter, we discuss the performance (in terms of time and accuracy) of our FPGA implementations of the two methods previously described to determine eigenvalues of real symmetric matrices. The testing platforms were PYNQ Z2 and livfpga that were described before. For both CPU only and FPGA, the optimization level is 0 in evaluation which for preventing optimization of compilers.

In section 4.1, it compares of CPU only and FPGA on PYNQ Z2. The following section compares them on livfpga. Section 4.3 compares the deviation of eigenvalue among Numpy [22] and both implementation on FPGA.

The matrices which were used for C host and FPGA were the same. Test data was generated by programs. In order to create real symmetric matrix with different sizes, the form of test matrix is:

$$\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & 2 & 2 & \cdots & 2 \\ 1 & 2 & 3 & \cdots & 3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 2 & 3 & \cdots & n \end{bmatrix}$$

The hypothesis is that the execution time is increased when the size of matrices is increased as well. Also, this trend is proved in later section. So, for evaluation, it used this form of real symmetric matrices to do experiments.

C programs and bitstream files for both algorithms were compiled with optimization level zero which prevents optimization from compilers. Moreover, the execution time for C programs was generated by 'clock_gettime()' C function in Linux. In order to compare the time of calculation on CPU and FPGA, it only recorded the starting and ending time of calculation of QR and Jacobi algorithms. The deviation of execution time on CPU was larger than FPGA. C programs were run 10 times with each specific size of a matrix, and the execution time was the smallest among them. For FPGA, the execution time didn't involve the data transfer between Host and DDR memory on FPGA because time of transferring data between host and FPGA usually is far smaller than execution time on FPGA. But, the execution time on FPGA included data transfer between kernel and DDR memory on FPGA. The deviation of execution time on FPGA between each execution was small, it wouldn't be larger than 10ms by observation. The bitstreams were only executed once with each size.

4.1 Execution Time on PYNQ Z2

The evaluation on PYNQ Z2 is comparing the QR algorithm and the Jacobi method

with different version of code. For host only part, both algorithms were written in C programming language, compiled with GNU Gcc compiler [18] and executed on ARM core. For FPGA part, it utilized python on ARM core and C for FPGA part. Because Vivado HLS supports high-level language like C for designing IPs for FPGA.

Size	QR (Sec)		
	C host only	Python + FPGA	Iterations
10	0.0012222	0.0011298	14
20	0.0109492	0.0057256	33
30	0.0312661	0.0144295	42
40	0.0723673	0.030241	54
50	0.1398536	0.0563406	68
60	0.2470352	0.0963079	81
70	0.5965278	0.2433249	540
80	0.763963	0.2905871	248
90	1.0682066	0.4012138	287
100	1.5640746	0.5840399	422

Table 4.1.1 Execution Time of QR on PYNQ Z2

The execution time of QR on PYNQ was faster than ARM CPU. Also, for a given matrix, the numbers of iterations were the same for the two different implementations of QR algorithms. In terms of the eigenvalues and eigenvectors which were calculated by both implementation of the QR algorithm were the same with given matrices. Figure 4.1.1 shows that the execution time of QR algorithm on FPGA part was faster than a single ARM core. These execution times didn't involve overhead of copying data from host to FPGA and vice versa. The trend of QR algorithm in figure 4.1 looks like $O(n^2)$.

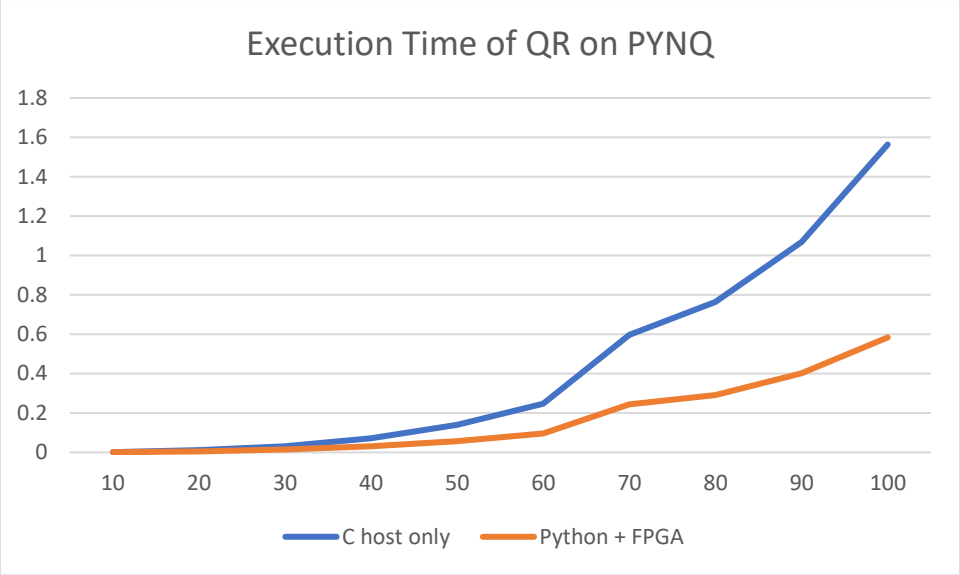

Figure 4.1.1 Execution Time of QR Algorithm on PYNQ Z2

For the Jacobi method, execution time was listed in table 4.1.2. The execution time of FPGA was still faster than CPU host only. Also, the numbers of iterations were the same among both different implementations of the Jacobi method with given matrices. Due to time complexity of Jacobi method is O(mn), the execution time was increased linearly with different sizes of matrices. The number of iterations of Jacobi was higher than QR algorithm because non-diagonal elements in QR were eliminated in each loop.

Size	Jacobi (Sec)		
	C host only	Python + FPGA	Iterations
10	0.0012748	0.0007925	85
20	0.0057706	0.0021672	219
30	0.0122985	0.0039274	329
40	0.0225231	0.006742	460
50	0.0293996	0.0084612	484
60	0.0358025	0.0102157	493
70	0.0499564	0.0132973	597
80	0.070121	0.0186138	723
90	0.0837204	0.0220417	775
100	0.0858889	0.0221593	719

Table 4.1.2 Execution Time of Jacobi Method on PYNQ Z2

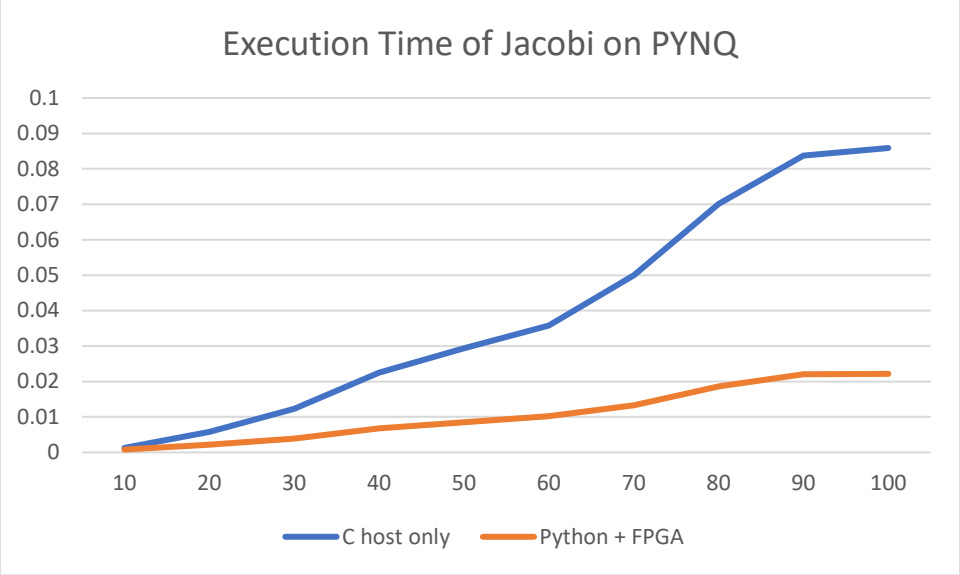

Figure 4.1.2 Execution Time of Jacobi Method on PYNQ Z2

4.2 Execution Time on Alveo Card(s)

This section is about the result and evaluation on HPC environment in terms of livfpga. Unlike CPU, FPGA has different memory resources like DDR RAM and BRAM. The transfer speed of BRAM is far faster than DDR RAM. In order to compare the difference between DDR RAM and BRAM. In section 4.2, it compares implementations with DDR RAM and BRAM on FPGA. Section 4.2.1 compares the implementation of FPGA with DDR memory and CPU. Section 4.2.2 compares the implementation of FPGA with BRAM and CPU. Finally, section 4.2.3 compares the execution time between executing kernels individually and simultaneously.

4.2.1 Execution Time with DDR Memory on FPGA

In this section is about the comparison for both algorithms on different platforms. In order to store large matrices for FPGA, the implementation for Alveo card utilized DDR memory on FPGA. The clocks of both implementations are listed in table 4.2.1. The timing paths in the implementation of Jacobi method were unable to meet the requirement of using maximum data clock, so SDAccel changed the data clock from 300 MHz to 242 MHz. The kernel time includes the time of scheduling and executing for kernel function. The computer units (CU) time means the execution time of computer units on FPGA. The execution time of FPGA involve data transfer between kernel and DDR memory on FPGA but no data transfer among host and FPGA.

	QR	Jacobi

Data Clock	300MHz	242MHz
Kernel Clock	500MHz	500MHz

Table 4.2.1 Clocks for Both Implementations

Size	QR algorithm (Sec)				
	C host only	Iterations of CPU	C++ + FPGA Kernel Time	C++ + FPGA CU Time	Iterations of FPGA
10	0.0001047	14	0.0009456	0.0004676	14
50	0.010818	68	0.0342894	0.0338184	68
100	0.1096975	422	0.359982	0.359504	422
200	0.6255859	268	1.92501	1.92461	268
300	2.7422642	989	8.43784	8.43774	989
400	4.6203722	529	14.3208	14.3208	529
500	8.9707666	660	29.2692	29.2701	660
600	24.4145338	2260	67.2841	67.2868	2260
700	33.2559055	1930	102.833	102.837	1930
800	43.8822578	1084	140.541	140.548	1084
900	80.9489243	8139	238.542	238.552	8139
1000	89.4663945	1504	247.72	247.731	1504

Table 4.2.2 Comparison of Execution Time between FPGA and CPU with OR

In table 4.2.2, it shown the execution time of CPU only and FPGA. The numbers of iterations of CPU and FPGA were the same with sizes of matrices from 10 to 1000. It meant the eigenvalues of CPU were the same as FPGA when the numbers of iterations were the same. Figure 4.3 shown the comparison between time of CPU only and CU time on FPGA.

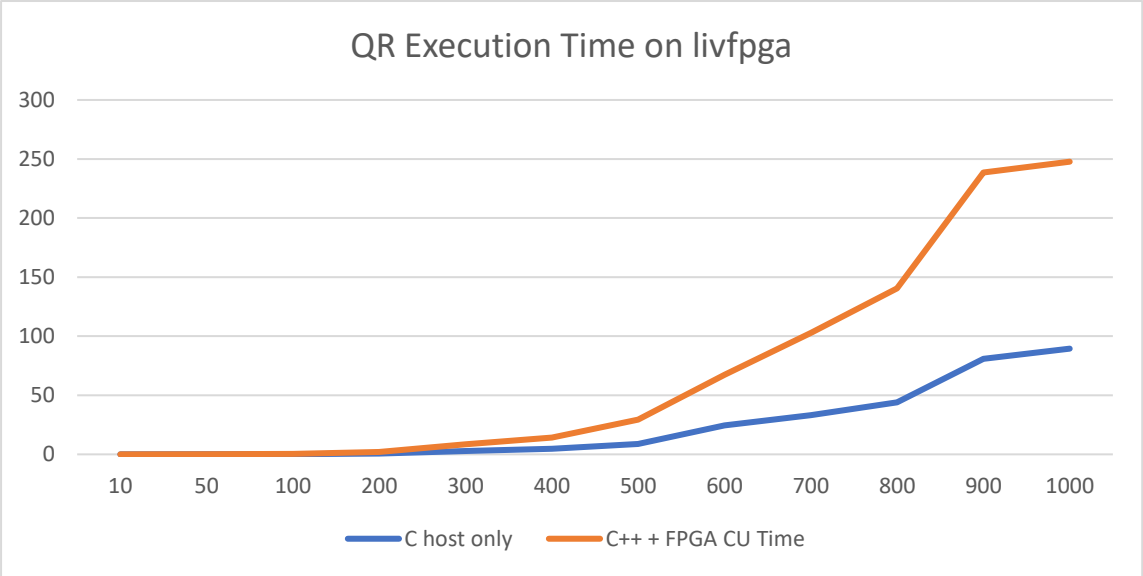

Figure 4.2.1 Comparison between Time of CPU and CU time on FPGA

The graphs of CPU and FPGA were similar, but the QR algorithm on FPGA was far slower than CPU only.

Size	Jacobi method (Sec)				
	C host only	Iterations of CPU	C++ + FPGA Kernel Time	C++ + FPGA CU Time	Iterations of FPGA
10	0.0002369	85	0.0007678	0.0003049	85
50	0.0032626	484	0.005431	0.0049479	484
100	0.0090677	719	0.0127531	0.012228	719
200	0.0246919	1070	0.0348528	0.0343852	1070
300	0.0541866	1534	0.0698871	0.0694075	1534
400	0.0897544	1912	0.110724	0.110151	1912
500	0.1487879	2235	0.163559	0.163023	2235
600	0.2155234	3020	0.253849	0.253329	3020
700	0.2667646	3154	0.311752	0.311271	3154
800	0.3147876	3145	0.402687	0.402188	3145
900	0.3909246	3485	0.445404	0.444913	3485
1000	0.4830077	3832	0.541875	0.5413	3832
2000	1.4854332	5449	1.57339	1.57292	5449
3000	3.0086925	7446	3.22777	3.22742	7446
4000	4.3502644	7869	5.44801	5.44774	7869
5000	5.9886706	8794	6.63137	6.63113	8794
6000	7.1836527	8522	8.25105	8.25094	8522
7000	9.961239	9906	11.2066	11.2066	9906
8000	13.3987016	11184	23.4107	23.4112	11184
9000	16.7333442	12568	18.3196	18.3199	12568
10000	11.7590399	7021	14.403	14.4031	7021

Table 4.2.3 Comparison of Execution Time between FPGA and CPU with Jacobi

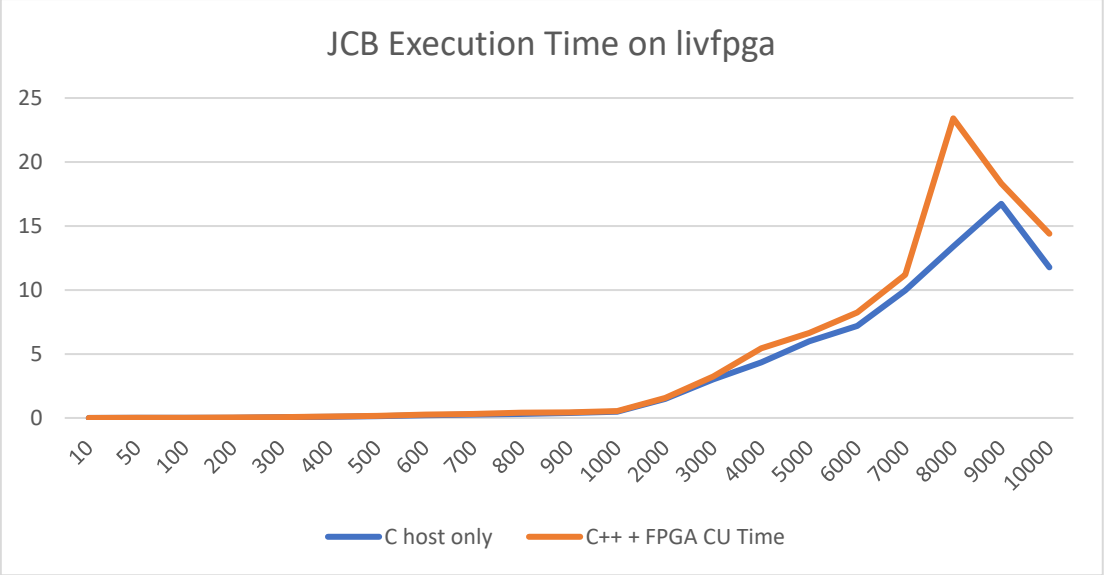

Figure 4.2.2 Comparison between Time of CPU and CU time on FPGA

In table 4.2.3, it shown the execution time on both CPU and FPGA, and the number of iterations. The implementation of the Jacobi method on FPGA was closer to CPU only implementation than QR implementation on FPGA. The time drop from 9000 to 10000 was due to the declination of the number of iterations when size was 10000. Moreover, the execution time at 8000 of FPGA was much slower than CPU. We explored this unexpected "peak" in the graph, by looking at matrix sizes from 7100 to 9000 in more detail.

In table A.3 (in Appendix), it shown the time of data transfer on different memory ports, and the summation of transferring time among all memory port. The variable in_matrix was the input matrix of kernel, and out_matrix1 and out_matrix2 were used for storing eigenvalues and eigenvectors respectively. Values in table A.3 was calculated by multiplying number of data transfer and average latency together. SDAccel provides profiling summary which include the number of transfer (table A.2 in Appendix) and average latency (ns) (table A.1 in Appendix) between kernel and DDR memory on FPGA, by multiplying them, the total time of transfer of each memory port is calculated. Table A.4 (in Appendix) shown the execution time of Jacobi on FPGA with sizes from 7100 to 9000.

Comparing the number of iterations in table A.4 and the number of data transfer in table A.2. It shown the relation between number of iteration and the number of data transmission between DDR RAM and kernel. The number of iterations was related to the number of data transfer of each memory port. Because whenever calculations like QR factorization in QR algorithm or Givens rotation in Jacobi method, the calculated elements would be written back to memory and read new elements from DDR memory. However, the number of iterations is not the only factor which affects execution time on FPGA.

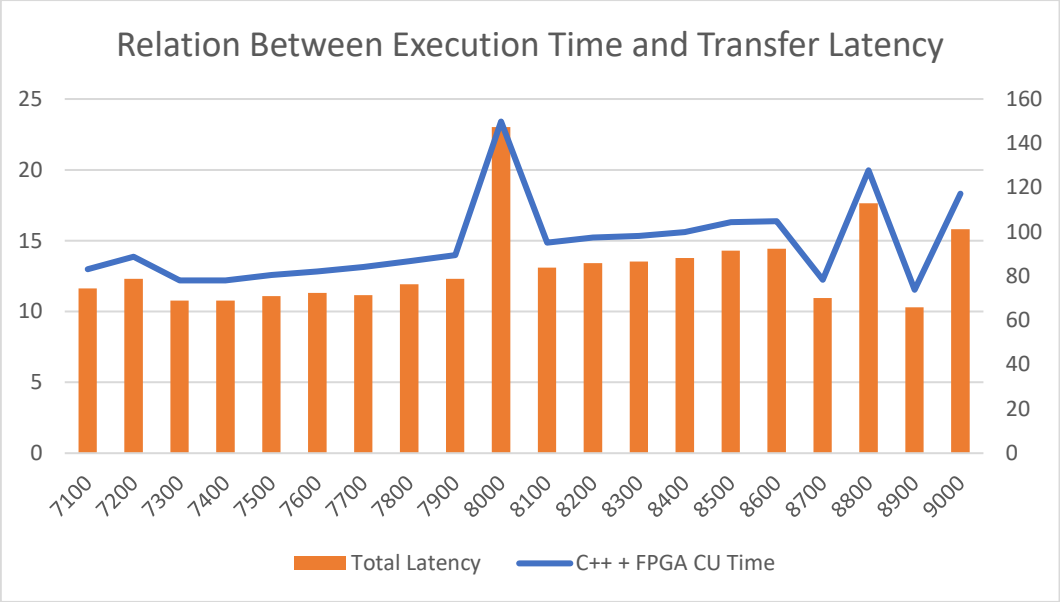

Figure 4.2.3 Relation Between Execution Time and Total Transfer Latency

The comparison in figure 4.2.3 which compares the total memory latency and execution time together. Also, it shown the relation between total memory latency and execution time on FPGA. The average latency of data transfer between kernel and DDR RAM is another factor for affecting execution time. There are some possible reasons that cause the fluctuation of the average latency of data transfer such as AXI burst size, burst length and reading/writing DDR memory concurrently. Adding time of transferring data of each memory port is not the correct total time for data transmission between kernel and DDR memory because some data transmission was done simultaneously, it is a good measure point.

The trend of execution time and the number of iterations is that both execution time and number of iterations are increased when the size of matrices is increased. Doing an experiment with more sizes of matrices between 7000 to 9000 can prove this trend exist. By looking at figure 4.2.4, it shown the slope of the number of iterations was very similar to the slope of execution time. In most time, the execution time was related to the number of iterations. For the other fluctuations, they were very possibly affected by the average data latency which is discussed previously.

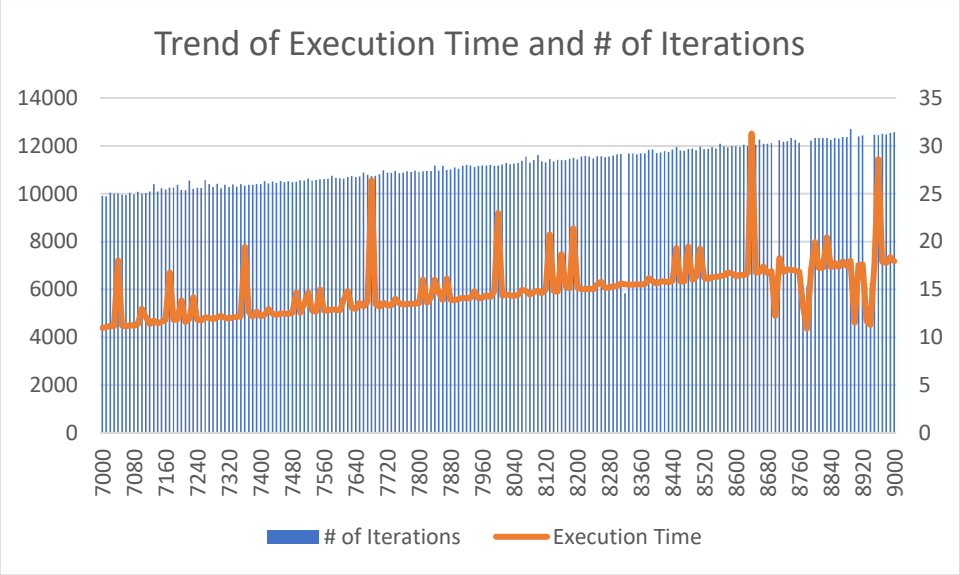

Figure 4.2.4 The Trend of Execution Time and Number of Iterations

4.2.2 Execution Time with BRAM

Putting data in BRAM instead of DDR RAM on-board can eliminate the overhead of data transfer between kernel and DDR RAM on FPGA. The data transfer between kernel and DDR RAM on FPGA are much slower than BRAM. If users utilize BRAM instead of DDR RAM for their implementation, they can achieve better performance. However, the size of BRAM on FPGA is much smaller than DDR RAM. On Alveo U200, it only has 7947 KB for BRAM. So, in this section, the size of matrices will be set from 10 to 500. The Jacobi method was used for the evaluation in this section.

Size	Jacobi Execution Time (Sec)			
	C host only	FPGA with DDR CU Time	FPGA with BRAM CU Time	Iterations
10	0.0001324	0.0003049	0.0002585	85
50	0.0031943	0.0049479	0.0027906	484
100	0.0086072	0.012228	0.0062867	719
150	0.0194226	0.0227997	0.0107087	923
200	0.0217383	0.0343852	0.0154517	1070
250	0.0441193	0.046872	0.0214045	1230
300	0.0538879	0.0694075	0.0309276	1534
350	0.0783253	0.0977631	0.0432183	1889
400	0.0839225	0.110151	0.0495305	1912
450	0.1362884	0.196443	0.0635534	2228
500	0.1540189	0.163023	0.0702775	2235

Table 4.2.4 Execution Time of Jacobi on CPU and FPGA with BRAM

By looking at table 4.2.4 and figure 4.2.5, they shown the execution time of FPGA with BRAM was faster than CPU only after 10. Also, it faster than the implementation with DDR memory on FPGA as well. By getting rid of the overhead of transferring data between kernel and DDR RAM, the execution time of the Jacobi implementation would be faster than CPU only. Because the latency of transferring data from/to BRAM is smaller than DDR RAM.

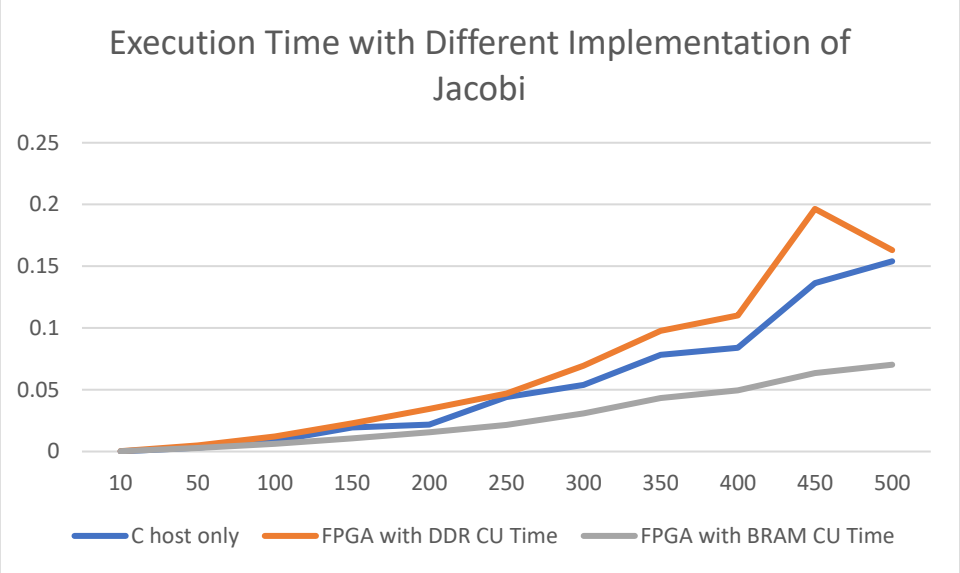

Figure 4.2.5 Execution Time of Jacobi with Different Settings

Moreover, the QR algorithm with BRAM was evaluated. Table 4.2.5 shown the execution time of the QR algorithm with BRAM.

Size	QR Execution Time (Sec)			
	C host only	FPGA with DDR CU Time	FPGA with BRAM CU Time	Iterations
10	0.0001047	0.0004676	0.0002059	14
50	0.0097629	0.0338184	0.0088463	68
100	0.1096975	0.359504	0.0770608	422
150	0.2540024	0.812629	0.156728	200
200	0.6255859	1.92461	0.356021	268
250	1.6047293	4.82088	0.864026	652
300	2.7422642	8.43774	1.49405	989
350	3.1379971	9.72959	1.68028	462
400	4.6094529	14.3208	2.46863	529
450	6.4912594	31.4502	3.49162	595
500	8.9707666	29.2701	4.80424	660

Table 4.2.5 Execution Time of QR Algorithm with BRAM

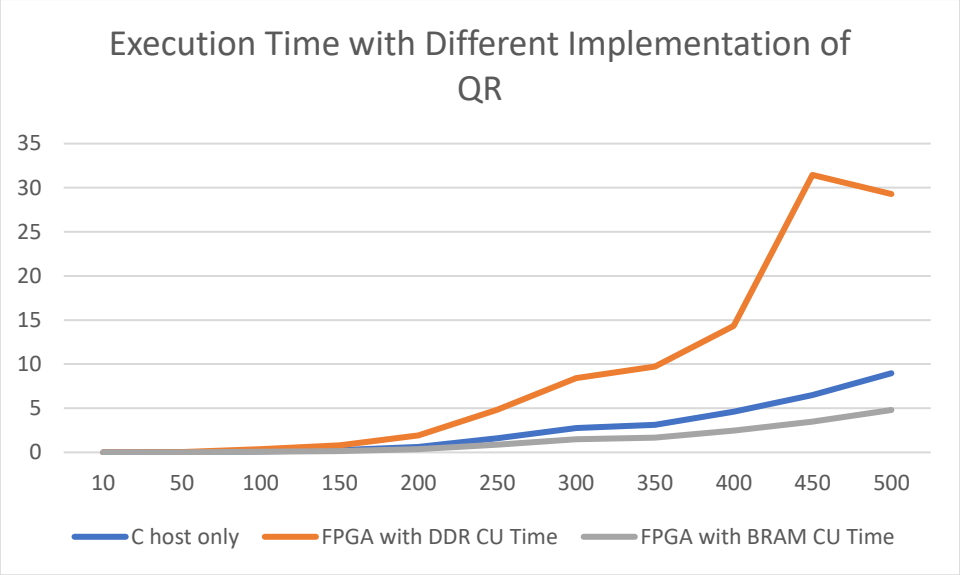

Figure 4.2.6 Execution Time of QR with Different Settings

By looking at figure 4.2.6, it shown the implementation of QR algorithm on FPGA with BRAM faster than CPU. Still QR implementation on FPGA with DDR RAM was the slowest one. According to results in this section, elimination of the overhead of transferring data from/to DDR RAM can increase the performance on FPGA. If programmers want to place large dataset in DDR memory on board, they should consider how to decrease the number of transferring data from/to DDR RAM. Maybe users can implement a cache mechanism for decreasing time of data transfer among kernel and DDR RAM.

4.2.3 Executing Kernels Simultaneously

Programmers can put multiple kernels in a bitstream file and execute those kernels concurrently. In this section, we evaluate the performance of a bitstream that contains kernels of Jacobi method both using DDRAM (as per section 4.2.1) and using BRAM (as per section 4.2.2). We compare the execution time of this combined bitstream to the sum of the times for running each kernel sequentially. The purpose of this section is to determine when run all kernels concurrently whether the execution time would be similar to execute kernels individually.

Because it compared the kernels in section 4.2.1 and 4.2.2, the sizes of matrices were set from 10 to 500. Table 4.2.6 shown the clock for each bit stream files for the evaluation in this section. The bitstream file for Jacobi with DDR RAM is 240 MHz, which is the same as section 4.2.1. For Jacobi with BRAM in section 4.2.2 is 180 MHz. And, the bitstream file which contains all kernels is 171 MHz.

Data Clock of bitstream files (MHz)		
JAC with DDR	JAC with BRAM	JAC with DDR and BRAM
240	180	171

Table 4.2.6 Data Clocks of Each Bitstream Files

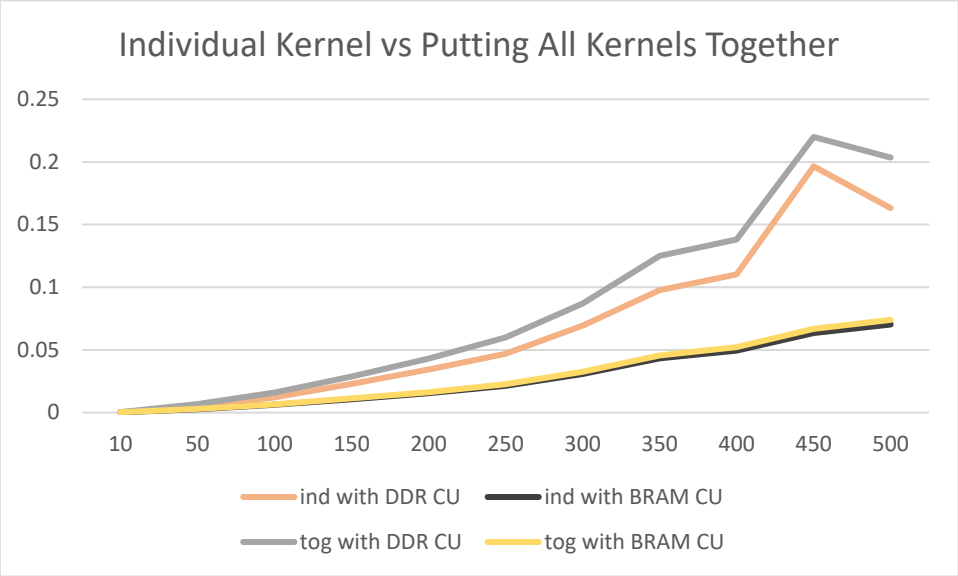

Figure 4.2.7 Comparison of Execution Time among Bitstreams

Looking at figure 4.2.7, it shown that the execution time among those bitstreams were similar. Kernels with BRAM were almost the same to each other because the difference of both clocks was small. On the other hand, the difference of clocks between Jacobi with DDR was about 70 MHz, so the execution time of running kernels together would be slower than executing individually. However, if all the clock were the same, they should have the same execution time. This is an advantage for FPGA, it can execute multiple kernels for multiple matrices simultaneously without affecting the performance. A CPU can only deal with matrices one-by-one.

Figure 4.2.8 shown the execution time on CPU and FPGA. Both calculated a specific matrix and a matrix which has 500 dimensions in one execution. Kernels were faster than host when the sizes were set from 500 to 1000, the execution time of the matrix with 500 dimensions were in involved in the execution time of sizes of matrix on FPGA. However, host needed to calculate both matrices one-by-one.

The purpose of this experiment aims to prove kernels can be executed for matrices simultaneously. And, it shown the running time of a bitstream with multiple kernels is the running time of slowest kernel inside the bitstream. However, in practical, a bitstream should involve kernels that are the same to and handle different matrices simultaneously.

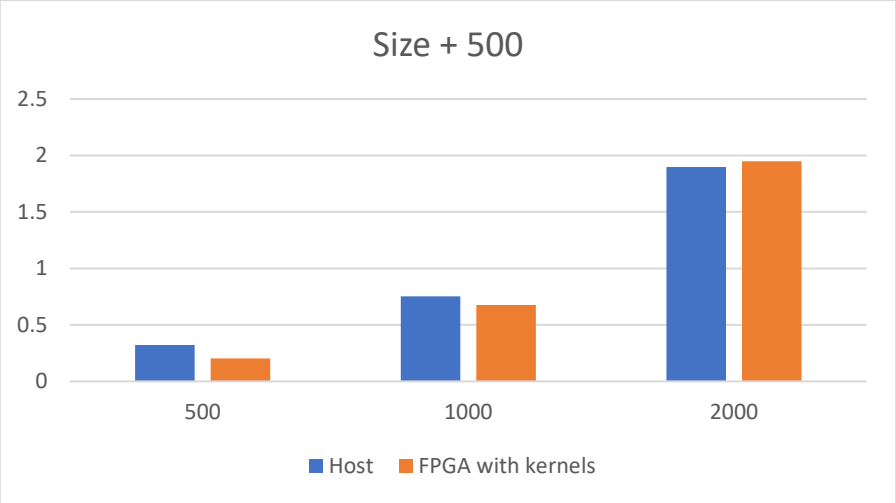

Figure 4.2.8 Execution Time of Host and FPGA with Two Kernels

4.3 Numpy VS Implementations on FPGA

In this section, it compared the deviation of eigenvalues which were generated by both implementations with eigenvalues from Numpy [22]. Numpy is a mathematical library in Python for calculating matrices. The sizes of matrices were from 10 to 100 in this section. The eigenvalues from Numpy may not be the correct answer for matrices in evaluation, but it can be used a comparison between both implementations in this project and other implementation from others. This comparison was done on PYNQ Z2. Because PYNQ Z2 provides Python API for controlling the FPGA part, and Numpy is a mathematic library for Python as well. By executing both implementations and Numpy with Python, they can be compared fairly.

Size	Eigenvalue Deviation						
	Numpy	QR	QR-Numpy	QR Error Rate	JCB	JCB-Numpy	JCB Error Rate
10	55	55.000034	3.4E-05	6.18182E-05	55.000049	4.9E-05	8.90909E-05
20	210	210.000139	0.000139	6.61905E-05	210.000507	0.00051	0.000241429
30	465	465.000277	0.000277	5.95699E-05	465.00115	0.00115	0.000247312
40	820	820.00065	0.00065	7.92683E-05	820.002266	0.00227	0.000276341
50	1275	1275.001075	0.001075	8.43137E-05	1275.005331	0.00533	0.000418118
60	1830	1830.00174	0.00174	9.5082E-05	1830.00608	0.00608	0.00033224
70	2485	2485.002306	0.002306	9.27968E-05	2485.009472	0.00947	0.000381167
80	3240	3240.001816	0.001816	5.60494E-05	3240.014629	0.01463	0.000451512
90	4095	4095.003243	0.003243	7.91941E-05	4095.019048	0.01905	0.000465153
100	5050	5050.004902	0.004902	9.70693E-05	5050.024932	0.02493	0.000493703

Table 4.3.1 Deviation among Numpy and Both Implementations

According to table 4.3.1, it shown the summations of absolute eigenvalues from

the QR algorithm, Jacobi method and Numpy. Also, table 4.3.1 lists the difference of total absolute eigenvalues between Numpy and the QR algorithm, and the difference of total absolute eigenvalues between Numpy and Jacobi method. The error rates of both implementations were far smaller than 1%.

On the other hand, the comparison of execution time of Numpy and both implementation with DDR RAM and BRAM on PYNQ.

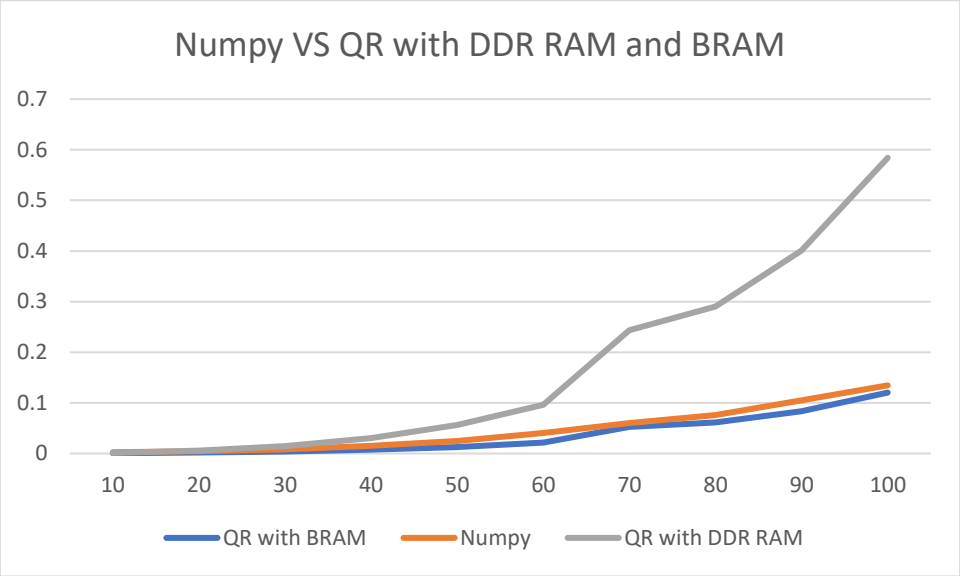

Figure 4.3.1 Comparing Running Time of Numpy and QR with DDR RAM and BRAM

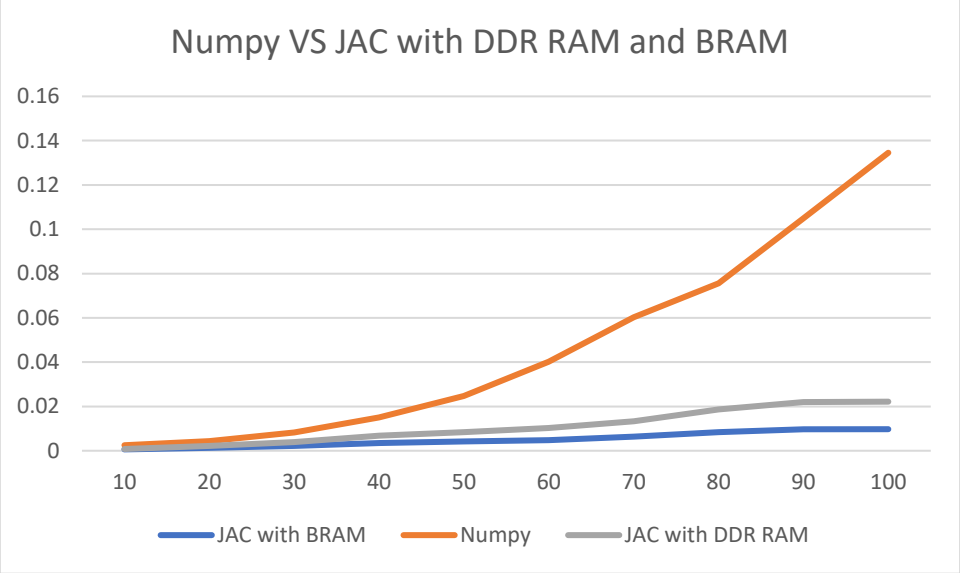

Figure 4.3.2 Comparing Time of Numpy and Jacobi with DDR RAM and BRAM

The execution time of Numpy was faster than QR algorithm with DDR RAM on PYNQ Z2 and similar to QR algorithm with BRAM. But it was much slower than Jacobi method no matter with DDR RAM or BRAM on PYNQ.

5. Discussion

In this chapter, I want to discuss some points that I had learnt in this project. This section aims to provide some ideas about improving the performance of both implementations on FPGA in future. In section 5.1, it discusses different data type for FPGAs. In section 5.2, it describes some learning points. Also, it discusses some professional issues at the last section.

5.1 Discussion of Data Type on FPGA

In this project, I chose to use single precision data type in terms of float for matrices. However, if float is replaced by double, the utilization of resource will exceed the resource on PYNQ. Figure 5.1.1 and 5.1.2 are the usage of resource of Jacobi method with DDR RAM on PYNQ. The utilization of DSP48E and LUTs was increased significantly with double precision data type. Take a single instruction as an example, for doing float addition/subtraction, it cost 2 DSP48 slices. For double addition/subtraction, it cost 3 DSP48 slices. For multiplication with float and double, it cost 3 DSP48 slices and 11 DSP48 slices respectively. However, for division and square root, they don't utilize DSP48 slices but LUTs and FFs. By looking the whole algorithm, the usages of DSP48E were 89 for float and 235 for double respectively. Also, the usages of LUTs were 39634 for float and 65324 for double.

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	869	-
FIFO	-	-	-	-	-
Instance	44	89	21384	36412	0
Memory	16	-	0	0	0
Multiplexer	-	-	-	2225	-
Register	0	-	1885	128	-
Total	60	89	23269	39634	0
Available	280	220	106400	53200	0
Utilization (%)	21	40	21	74	0

Figure 5.1.1 The Float Usage of Jacobi with DDR on PYNQ

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	933	-
FIFO	-	-	-	-	-
Instance	68	235	37806	62044	0
Memory	31	-	0	0	0
Multiplexer	-	-	-	2219	-
Register	0	-	2169	128	-
Total	99	235	39975	65324	0
Available	280	220	106400	53200	0
Utilization (%)	35	106	37	122	0

Figure 5.1.2 The Double Usage of Jacobi with DDR on PYNQ

On the other hand, ap_fixed type had been considered before, however, it didn't provide better performance on cycles of functions than float type. For example, the function for calculating sin and cos for Givens rotation took 83 cycles with float type, if I changed it from float to ap_fixed, it took more cycles. About the precision for implementations, I have explained the mechanism of ap_fixed type in section 1.2.2. However, it didn't work in implementations of the QR algorithm and Jacobi method. I've tried to utilize ap_fixed in both implementations on PYNQ and SDAccel. Both implementations return zeros on PYNQ and crushed when emulating them with sw_emu mode on Alveo. This situation was caused by sqrt function in hls_math.h, because the sqrt function returned zero after calculation. By converting ap_fixed to float before calling sqrt function can fixed this problem. Even though this problem is fixed, the eigenvalues and eigenvectors returned by both implementations were wrong with ap_fixed type.

5.2 Learning Points

I learnt how to implement projects on FPGA in this project. Familiarizing Vivado tools is the first step. Vivado HLS enables programmers to use high-level language to develop project on FPGA. However, there are some points that is not similar to develop program on x86 machine. For example, the initialization of x86 need to be done by C program itself, because allocated space might contain garbage values. For FPGA, the initialization in BRAM would be set to zeros automatically, on other words, programmers don't need to reset those arrays again. Moreover, programmers tend to remove redundant code in implementations in order to achieve better performance or readability of code. But redundant code for FPGA could improve the performance on FPGA which is discussed in section 1.2.1. By decreasing time of accessing DDR RAM on FPGA can improve the performance on FPGA as well. If programmers need to

implement project for large datasets on FPGA, they need to consider some approaches to decrease the time of accessing DDR memory. Furthermore, each instruction in code consumes resources on FPGA, we need to make sure the usage of resources doesn't exceed the amount of resources on FPGA board. After finishing code on Vivado HLS, it can generate an IP for generating overlays.

Vivado Design Suite is a tool for creating overlays for FPGA board such as PYNQ Z2. By using this tool, I understood how to create overlays for the FPGA part on PYNQ Z2. ARM core and FPGA core are connected by many AXI interfaces, users need to setup those interfaces for their requirements. For example, I set an AXI_ACP port and AXI_GP port for accessing memory and setting parameters on ARM core respectively. However, it is not enough for transmitting data from/to FPGA core by setting ports on ARM core. It needs other AXI interconnector to achieve this goal. By utilizing AXI interconnectors, ports on IPs can connect to ARM core. And, Vivado design suite can generate bitstreams when overlays are done.

In order to implement project on FPGA card in HPC environment, SDAccel is a tool for this purpose. SDAccel is more complicated than Vivado HLS and Vivado Design Suite. It contains host part and FPGA part. FPGA part is similar to the process of Vivado HLS tool, but host part is different. In SDAccel, users can set different DDR banks for memory ports. Also, adding options for profiling is possible for projects. Moreover, SDAccel could change the clock speed for bitstreams because of the design of bitstreams which is discussed in section 3.3. Also, the latency of data transfer between kernels and DDR RAM on FPGA can affect performance on FPGA significantly. There are some reasons that the average latency of data transfer is increased such as AXI burst size, burst length and reading/writing DDR memory concurrently.

In summary, I am familiar with using Xilinx tools to develop projects on Xilinx FPGA board. Also, I understood some techniques of writing code on FPGA like using pragmas to make operations be executed simultaneously. Moreover, I learnt the reason that affects data clock of implementations, and the reason which impacts the performance on FPGA when utilizing DDR memory to store dataset.

5.3 Professional Issues

According to BCS code of conduct [23], there are some points which are related to development of projects. This section aims to describe the relation of this project and BCS code of conduct.

- Public Interest

  This project aims to provide an idea about solving eigenvalues and eigenvectors

with FPGA on HPC environments. Which is useful in fields like machine learning and computer vision. Individuals can apply this idea if they have FPGA devices and they want to accelerate the process of solving eigenvalues and eigenvectors.

- Professional Competence and Integrity

During the development of this project, I learnt necessary knowledge in order to develop this project. Also, I utilized skills that I knew to implement programs and bitstreams for different FPGA platforms.

- Duty to Relevant Authority

The test dataset in this project is generated by programs, it is not related to any real data or human data. Also, the test results in this project are represented correctly in this dissertation.

- Duty to The Profession

I discussed with my supervisor and a classmate who did similar project like mine in order to figure out better methods for improving my project and learning new knowledge about development on FPGA. Also, I tried to improve my project with any possible approach.

6. Future Work

For the QR algorithm, F. Rotella [16] introduced a method to parallel the process of QR factorization. In my implementation of the QR algorithm, it needs to eliminate every non-diagonal element in a matrix one-by-one. If it can be parallelized, the performance of the QR algorithm should be improved. On the other hand, the clock of the Jacobi implementation was not default data clock 300 MHz of Xilinx Alveo U200. In my opinion, by rewriting some code in Jacobi method can make the implementation meet the requirement of executing the bitstream of Jacobi with default clock. As a result, the execution time of Jacobi method on FPGA with DDR RAM could be faster than Intel CPU on livfpga. Moreover, finding an approach for making ap_fixed type work properly on implementations is another work can be done in future. For the purpose of comparing this project with others approaches, the approaches in Lin [10] can be implemented for this purpose. Also, putting implementations into open-sourced library like fBLAS [28] can be done in future works.

7. Conclusion

In recent years, FPGA has become a popular technology for many different fields such as machine learning. FPGA technology enables programmers to develop their own circuit for specific purposes without having to produce an ASIC. As a result, FPGAs can be deployed in an HPC environment because programmers are able to change the overlays in FPGA anytime, which means they can design specific applications in FPGA. On the other hand, finding Eigenvalues is an important mathematic technique for machine learning and other fields. We have shown that this process can be implemented on an FPGA. Popular algorithms for eigenvalues and eigenvectors include the QR algorithm and Jacobi method. This project has implemented the QR algorithm and Jacobi method on two FPGA platforms and we have quantified performance improvement for accelerating the process of finding eigenvalues and eigenvectors for real symmetric matrices.

Implementations FPGA of PYNQ Z2 were faster than ARM core. However, Implementations for FPGA on HPC environment were slower than Intel CPU when placing matrices in DDR RAM. On the other hand, by placing matrices in BRAM which eliminates latency and overhead of transferring data between kernel and DDR RAM. But, the size of matrices is restricted due to the size of BRAM. Moreover, adding the number of kernels on FPGA can enable FPGA to handle multiple matrices at the same time which is an advantage than CPU. For accuracy, the QR algorithm is more accurate than Jacobi method if we compare the eigenvalues from Numpy with both implementations.

There are some points that can be used for improving the performance on FPGA. By optimizing HLS code which enables bitstream files are executed at default clock in terms of maximum clock. Also, by paralleling some process like QR factorization in the QR algorithm can decrease execution time as well.

8. References

[1] Principle Component Analysis,
https://en.wikipedia.org/wiki/Principal_component_analysis
[2] K. Ovitcharov et al., "Accelerating deep convolutional neural networks using specialized hardware." Microsoft Research Whitepaper 2.11 (2015)
[3] Conventional Neural Network,
https://en.wikipedia.org/wiki/Convolutional_neural_network
[4] Xilinx Alveo, https://www.xilinx.com/products/boards-and-kits/alveo.html
[5] QR algorithm. https://en.wikipedia.org/wiki/QR_algorithm
[6] Jacobi method. https://en.wikipedia.org/wiki/Jacobi_eigenvalue_algorithm
[7] Givens rotation, https://en.wikipedia.org/wiki/Givens_rotation
[8] Jacobi Eigenvalue Algorithm for Symmetric Matrices,
http://fourier.eng.hmc.edu/e176/lectures/ch1/node1.html
[9] S. Aslan et al., “FPGA Implementation of Fast QR Decomposition Based on Givens Rotation”, Proc. 55th IEEE Int. Midwest Circuits Sys*t.*, pp. 470-473, 2012.
[10] Y. Liu et al., “Hardware Efficient Architectures for Eigenvalue Computation”, In: Proc. Design Automation & Test in Europe, p. 202 (2006).
[11] PYNQ Z2, http:// http://www.tul.com.tw/ProductsPYNQ-Z2.html
[12] System on a Chip, https://en.wikipedia.org/wiki/System_on_a_chip
[13] Vivado HLS, https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html
[14] Vivado Design Suite, https://www.xilinx.com/products/design-tools/vivado.html
[15] SDAccel, https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html
[16] F. Rotella et al., “Block Householder transformation for Parallel QR Factorization”, Applied Mathematics Letters 12 (1999) 29-34.
[17] Intellectual Property. https://www.xilinx.com/products/intellectual-property.html
[18] GNU GCC, https://gcc.gnu.org/
[19] CORDIC, https://en.wikipedia.org/wiki/CORDIC
[20] Household Transformation,
https://en.wikipedia.org/wiki/Householder_transformation
[21] J. Gotze, et al, “An efficient Jacobi-like al-gorithm for parallel eigenvalue computation,”IEEE Transac-tions on Computers, vol. 42, no. 9, pp. 1058 – 65, Sept. 1993.

[22] Numpy, https://www.numpy.org/
[23] BCS code of conduct, https://www.bcs.org/membership/become-a-member/bcs-code-of-conduct/
[24] HLS pipeline, https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/fde1504034360078.html
[25] HLS unroll, https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/uyd1504034366571.html
[26] HLS Loop_tripcount, https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/sty1504034367099.html
[27] Vivado Design Suite User Guide, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf
[28] T. De. Matteis et all, “fBLAS: Streaming Linear Algebra Kernels on FPGA”, Departmentof Computer Science, ETH Zurich.
[29] Basic Linear Algebra Subprograms, https://zh.wikipedia.org/wiki/BLAS
[30] HLS dependency, https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/dxe1504034360397.html
[31] Xilinx Alveo U200 Datasheet, https://www.xilinx.com/support/documentation/data_sheets/ds962-u200-u250.pdf
[32] HLS pragmas, https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/okr1504034364623.html

Appendix

Size	avg latency (ns)				
	in_matrix read	in_matrix write	out_matrix1 write	out_matrix2 read	out_matrix2 write
7100	254	161	196	234	159
7200	254	185	195	208	177
7300	253	133	195	217	129
7400	254	125	195	217	121
7500	253	125	195	217	122
7600	254	125	198	217	121
7700	253	125	194	194	122
7800	253	125	196	217	122
7900	253	127	197	217	123
8000	254	339	197	217	349
8100	253	125	196	217	122
8200	253	128	196	223	124
8300	253	125	196	217	121
8400	253	125	197	217	121
8500	253	129	195	217	125
8600	253	125	193	217	121
8700	251	159	195	228	157
8800	253	173	195	217	158
8900	250	132	195	219	129
9000	253	125	195	217	121

Table A.1 Average Latency for Read/Write of Each Memory Port

Size	# of data transfer of ports					
	in_matrix read	in_matrix write	out_matrix1 write	out_matrix2 read	out_matrix2 write	Total # of data transfer
7100	32550545	124648196	444	117306200	117313300	391818685
7200	33476400	128397601	450	120844800	120852000	403571251
7300	34381547	131256198	457	123523300	123530600	412692102
7400	35342207	135157108	463	127198600	127206000	424904378
7500	36323952	139130203	469	130935000	130942500	437332124
7600	37241900	142087701	475	133729600	133731200	446790876

7700	38263154	146228705	482	137614400	137622100	459728841
7800	39228884	149540385	488	140735400	140743200	470248357
7900	40244680	153424031	494	144388300	144396200	482453705
8000	41242000	157046001	500	147808000	147816000	493912501
8100	42499848	164514199	507	154823400	154831500	516669454
8200	43321339	164701840	513	155004600	155012800	518041092
8300	44454811	169888562	519	159882900	159891200	534117992
8400	45453975	172886176	525	162716400	162724800	543781876
8500	46549534	176909785	532	166489500	166498000	556447351
8600	47646922	181106023	538	170443400	170452000	569648883
8700	44499979	112938530	544	106287900	106296600	370023553
8800	49937250	190618451	550	179405600	179414400	599376251
8900	46570279	118184130	557	111223300	111232200	387210466
9000	52199136	198642637	563	186948000	186957000	624747336

Table A.2 Number of Transmission of Each Memory Port

Size	# of transfer * avg latency (Sec)					
	in_matrix read	in_matrix write	out_matrix1 write	out_matrix2 read	out_matrix2 write	Total Latency
7100	8.26784	20.0684	8.7E-05	27.4497	18.6528	74.4388
7200	8.50301	23.7536	8.8E-05	25.1357	21.3908	78.7832
7300	8.69853	17.4571	8.9E-05	26.8046	15.9354	68.8957
7400	8.97692	16.8946	9E-05	27.6021	15.3919	68.8657
7500	9.18996	17.3913	9.1E-05	28.4129	15.975	70.9692
7600	9.45944	17.761	9.4E-05	29.0193	16.1815	72.4213
7700	9.68058	18.2786	9.4E-05	26.6972	16.7899	71.4463
7800	9.92491	18.6925	9.6E-05	30.5396	17.1707	76.3278
7900	10.1819	19.4849	9.7E-05	31.3323	17.7607	78.7598
8000	10.4755	53.2386	9.9E-05	32.0743	51.5878	147.376
8100	10.7525	20.5643	9.9E-05	33.5967	18.8894	83.803
8200	10.9603	21.0818	0.0001	34.566	19.2216	85.8298
8300	11.2471	21.2361	0.0001	34.6946	19.3468	86.5247
8400	11.4999	21.6108	0.0001	35.3095	19.6897	88.1099
8500	11.777	22.8214	0.0001	36.1282	20.8123	91.539
8600	12.0547	22.6383	0.0001	36.9862	20.6247	92.3039
8700	11.1695	17.9572	0.00011	24.2336	16.6886	70.049
8800	12.6341	32.977	0.00011	38.931	28.3475	112.89

8900	11.6426	15.6003	0.00011	24.3579	14.349	65.9498
9000	13.2064	24.8303	0.00011	40.5677	22.6218	101.226

Table A.3 The Data Transfer Time on Different Memory Port

Size	Jacobi from 7100 to 9000 (Sec)		
	C++ + FPGA Kernel Time	C++ + FPGA CU Time	Iterations
7100	12.9762	12.9762	9995
7200	13.8674	13.8675	10164
7300	12.2029	12.203	10224
7400	12.1985	12.1986	10339
7500	12.5743	12.5744	10564
7600	12.8298	12.8298	10643
7700	13.1799	13.1399	10818
7800	13.5517	13.5518	10910
7900	13.9876	13.9877	11050
8000	23.4107	23.4112	11184
8100	14.8616	14.8617	11617
8200	15.2174	15.2177	11428
8300	15.3437	15.3438	11647
8400	15.6042	15.6044	11697
8500	16.3036	16.3038	11827
8600	16.3665	16.3667	11983
8700	12.2418	12.2419	6109
8800	19.9633	19.9636	12324
8900	11.5334	11.5334	6249
9000	18.3196	18.3199	12568

Table A.4 Execution Time of Jacobi with Size from 7100 to 9000