Matrix Inversion on a Many-Core Platform

By

ZHANGFAN ZHAO

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Chair, Bevan M. Baas

Member, Hussain Al-Asaad

Member, Soheil Ghiasi

Committee in charge
2021

Abstract

Matrix operations are a fundamental problem of scientific computation and industry computation, which are widely used in many applications. Among them, the inversion of matrices plays an essential role in multiple-input and multiple-output (MIMO) systems, image signal processing, least-squares analysis, etc. With dramatically increasing data sizes, the speed of inverting matrices usually becomes the key that affects the overall system performance. Therefore, this thesis proposes a many-core matrix inversion method based on Gaussian Jordan Elimination (GJE), which includes two implementations: a 603-processor design using only on-chip memory with a 16-bit fixed point and a 635-processor design using external off-chip memory with a 32-bit fixed point and a 32-bit float point. Details of the parallel algorithm based on the GJE are presented. All the unique programs loaded to the many-core platform and the mapping of the parallel architecture are described. The accuracy of using different data types are analyzed. Due to the word length and the computation complexity, the accuracy of the 16-bit fixed point is 2^{-1}, the accuracy of the 32-bit fixed point is 2^{-6} and the accuracy of the 32-bit float point is 2^{-9}. Due to the limitation of on-chip memory size, the implementation that uses only on-chip memory cannot invert the large matrices.

Therefore, the proposed implementation that uses off-chip memory with the 32-bit float point is compared against a general-purpose processor (i7-9700k) and a graphics processing unit (GPU) chip (NVIDIA GTX1070). The designs for the many-core chip, general-purpose processor and GPU are evaluated using the metrics of throughput per area (MatInv/sec/mm^2) and matrix inversions per energy (MatInv/J). Since different fabrication technologies are used, throughput, area and energy dissipation for all platforms are scaled to 14 nm. The improvement in throughput per area achieved from experiments is 20–60$\times$ among all simulated matrices versus the general-purpose processor implementation, and 3.7–19$\times$ versus the GPU implementation. The improvement in matrix inversions per energy achieved from experiments is 45–131$\times$ versus the general-purpose processor implementation, and 8.5–41$\times$ versus the GPU implementation.

Acknowledgments

I want to take this opportunity to thank a few people who have offered invaluable assistance throughout my years at UC Davis.

My deepest gratitude goes first and foremost to Dr. Baas, my major professor, who teaches me how to be an active researcher and overcome each difficulty step by step. His precision insight, constant encouragement, and critical thinking have greatly inspired me in academic research and career planning. I would also like to thank Professor Al-Asaad and Ghiasi for their patient support and valuable advice on my thesis.

It was also a great experience to be a member of the VLSI Computation Laboratory (VCL), which gives me an excellent opportunity to work with my talented peers and learn from them. I want to show my appreciation to Shifu Wu and Brent Bohnenstiehl from VCL, who gave me many ideas for my research project and helped me with tools.

In the end, I want to give my special thanks to my family. Thank you for supporting my study and always encouraging me to do better.

Contents

Abstract ii

Acknowledgments iii

List of Figures vi

List of Tables ix

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Thesis Organization . . . 2

2 Background 3
- 2.1 Definition of Matrix Inversion . . . 3
- 2.2 Gauss Jordan Elimination . . . 3
  - 2.2.1 Sequential Gauss Jordan Elimination with partial pivoting . . . 3
  - 2.2.2 Parallel Gauss Jordan Elimination . . . 5
- 2.3 Fixed Point . . . 7

3 AsAP Platform 9
- 3.1 Processors . . . 9
- 3.2 Inter-Processor Communication . . . 10
- 3.3 Independent Memory Modules . . . 11
- 3.4 Fine-Grain Clocking . . . 12
- 3.5 KiloCore (AsAP3) . . . 12

4 Implementations of Matrix Inversion on a Many-Core Platform 14
- 4.1 Matrix Inversion Kernels . . . 14
- 4.2 Block Diagram . . . 15
  - 4.2.1 InversionBigMem . . . 15
  - 4.2.2 InversionExternalMem . . . 18
- 4.3 Input Data Distribution . . . 20
- 4.4 Memory Architecture . . . 26
  - 4.4.1 InversionBigMem . . . 26
  - 4.4.2 InversionExternalMem . . . 27
- 4.5 Sort . . . 32
- 4.6 Logic Control . . . 34
- 4.7 Computational Array . . . 35

4.7.1 Normalization Chain . . . 37
4.7.2 Elimination Array . . . 40

5 Number Scaling in Matrix Inversion 45
5.1 Input Quantization and Scaling . . . 45
5.1.1 Quantization . . . 45
5.1.2 Input Scaling . . . 46
5.2 Scaling During Calculation . . . 46
5.3 Experiment and Results . . . 47
5.3.1 16-bit Fixed Point . . . 47
5.3.2 32-bit Fixed Point . . . 49

6 Comparison of Matrix Inversion Methods on AsAP3 52
6.1 Performance Comparison between Implementations on AsAP3 . . . 53
6.2 Error Analysis and Precision . . . 54
6.2.1 Condition Number . . . 54
6.2.2 Residual and Accuracy . . . 55
6.3 Accuracy Comparison between Different Data Types . . . 55
6.4 Summary . . . 56

7 Comparison of Matrix Inversion Methods on AsAP3 with General-Purpose Processor and GPU 62
7.1 Matrix Inversion Data Set . . . 62
7.2 Matrix Library . . . 62
7.3 Measurement and Simulation Methodology . . . 63
7.4 Comparison with Other Work . . . 65

8 Thesis Summary and Future Work 70
8.1 Thesis Summary . . . 70
8.2 Future Work . . . 71

Bibliography 75

List of Figures

2.1 Invert a 3 by 3 matrix using Gaussian Jordan elimination step by step. 4

3.1 Major components and connections of the seven-stage processor pipeline [1]. 10

3.2 Overview of inter core communication using circuit and packet networks. Writes are source-synchronous; responses include asynchronous wake-up signals for sleeping processors. Circuit links include configurable registers and an eastwest connection for one layer is expanded on the right [1]. 11

3.3 Components used in streaming instructions from a shared memory to a neighboring processor. Streaming logic is shared between two processors, with only the port 0 connection shown here [1]. 12

3.4 Top-level processor array of KiloCore (AsAP3) [1]. 13

4.1 Block diagram of InversionBigMem architecture. Input data is loaded evenly to different BigMems through InputDistribution block. All data is processed through Normalization and Elimination blocks. The LogicControl sends commands to different Memory modules about which part of data is going to be processed. Memory modules push data to Normalization and Elimination blocks and update memory with processed data. Div block is the core uses built-in divider ("/", division operator in c++) to calculate the division only. In each loop, Sort block finds the max pivot and sync with the LogicControl to start next iteration. After all the computations are done, each memory module output the results in sequence. The details of each block are shown in later sections. 16

4.2 Detailed version of Figure 4.1 that shows everything inside of each block. Blue circles are AsAP processors and yellow circles are memory modules. Red lines are packet links, black lines are circuit links, and green lines are memory links. There are eight BigMems (BigMem0-7) in this architecture. For simplicity, two of them are shown in the figure. There are 603 processors used in this architecture. 17

4.3 Block diagram of InversionExternalMem. The overall data flow of InversionExternalMem is similar to the InversionBigMem. The difference is that all data is stored in the external memories. The external memory has a 100 ns access latency. During the calculation, the BigMem works as buffer to store part of data and write back to the external memories after the data is processed. Div block is the core uses built-in divider ("/", division operator in c++) to calculate the division only. 18

4.4 Detailed version of Figure 4.3 that shows everything inside of each block. Blue circles are AsAP processors and yellow circles are memory modules. There are eight external memories (ExternalMem0-7) and eight BigMem (BigMem0-7) in this architecture. Red lines are packet links, black lines are circuit links, and green lines are memory links. For simplicity, three of them are shown in the figure. There are 635 processors in this architecture. 20

4.5 Block diagram of Input Distribution. 22

4.6 The memory interface of InversionBigMem architecture for one BigMem. MemController, SubMemController and Buffer are regular AsAP processors. Each AsAP processor can have two input ports (memory link and circuit link) and one packet link. Buffer core is used to merge two input ports into one. Instantiate more of this architecture can store a larger input matrix and improve parallelism 28

4.7 The memory interface of InversionExternalMem architecture. The figure shows the data transfer between one BigMem and one External Memory port. The external memory has a 16-bit I/O port with 100 ns access latency. To increase the bandwidth, instantiate more blocks like this. ExternalMemController, BigMemControllerA and BigMemControllerB are regular AsAP processors. Each AsAP processor can have two input ports (memory link and circuit link) and one packet link. Packet link is only used to transfer some low throughput information. 29

4.8 An example that shows the sort range for each iteration. The rank of the example matrix is eight, and the current pivot index is 3. Before starting the next iteration, the Sort kernel needs to find the element with the max absolute value in the red circle. That element with its corresponding row will be the new pivot row in the next iteration. 33

4.9 The figure shows the computation process in the third iteration. a_{22} is the pivot element, row 2 is called pivot row, and column 2 is called pivot column. The normalization phase is to normalize the pivot row by scaling the pivot element to 1. The Elimination phase uses all other rows to subtract the pivot row to eliminate the pivot column to 0. 36

4.10 Block diagram of Normalization Chain. 37

4.11 Example of a simplified elimination array with size 4×5. The blue processors only spread some configuration flags. The green and yellow cores process data simultaneously. Therefore, ideally, the accelerate ratio is 16. Blue processors are loaded with Elimination_First kernel, processors in green colour are loaded with Elimination_Mid kernel and processors in yellow colour are loaded with Elimination_End kernel. . . . 41

5.1 Matrices with different data ranges for 16-bit fixed version. Before the calculation, the input is scaled to different ranges. If the scaling factor equals one, the data is not scaled. If the scaling factor equals 1/2, the data is scaled to half of its original value before the calculation starts. The figure shows the relative error with the different scaling factors being used. Lower is better. The legend shows the original data range of each data set. The x-axis shows the scaling factor. The product of these two numbers is the scaled data range before the calculation. 49

5.2 Matrices with different data ranges for 32-bit fixed version. Before the calculation, the input is scaled to different ranges. If the scaling factor equals one, the data is not scaled. If the scaling factor equals 1/2, the data is scaled to half of its original value. The figure shows the relative error with the different scaling factors being used. Lower is better. The legend shows the original data range of each data set. The x-axis shows the scaling factor. The product of these two numbers is the scaled data range before the calculation. 51

6.1 The residual of a 300×300 matrix with 300 condition number by using 16-bit fixed point with 8.8 format. It has the largest condition number and matrix size, which is considered as the worst-case of all data sets. The first plot does not show any similar pattern to the identity matrix and both difference (the third plot) and ratio (the fourth plot) is very large. Therefore, the 16-bit fixed point is not usage practically for now. 59
6.2 The residual of a 300×300 matrix with 300 condition number by using 32-bit fixed point with 16.16 format. It has the largest condition number and matrix size, which is considered as the worst-case of all data sets. The first plot shows the result is very close to the identity matrix and both difference (the third plot) and ratio (the fourth plot) is very small. Therefore, the 32-bit fixed point can keep a good accuracy. . . . 60
6.3 The residual of a 300×300 matrix with 300 condition number by using 32-bit float point. It has the largest condition number and matrix size, which is considered as the worst-case of all data sets. The first plot shows the result is very close to the identity matrix and both difference (the third plot) and ratio (the fourth plot) are smaller than 32-bit fixed point. Therefore, the 32-bit float point has a higher accuracy than the 32-bit fixed point. 61

7.1 Scaled Throughput per Area for different sized matrices on different platforms. Higher is better. Source data is given in Table 7.4. 65
7.2 Scaled relative Throughput per Area of the implementation on AsAP3 vs the implementations on GPU and general-purpose processor. Source data is given in Table 7.4 66
7.3 Matrix inversions per energy for different sized matrices on different platforms. Higher is better. Source data is given in Table 7.5. 67

8.1 IEEE 754 half-precision format 71

List of Tables

2.1 Notation of an N-bit two's complement fixed point with m.n format, where $N = m+n$, m bits are used to represent the integer part, and n bits are used to represent the fraction part. 7

2.2 Range and precision for 16-bit fixed point. 8

4.1 The number of code for each unique kernel and the number of cores that are loaded with different kernels for the implementation with only on-chip memory. Number of code is measured as the number of lines of code in C++ after removing all the comments and blank lines. 19

4.2 The number of codes for each unique kernel and the number of cores that are loaded with different kernels for the implementation with off-chip memory. Number of code is measured as the number of lines of code in C++ after removing all the comments and blank lines. 21

5.1 Data set information for 16-bit fixed point 48

5.2 Data set information for 32-bit fixed point 50

6.1 Throughput per watt for different many-core implementations with various input matrices. The unit is MatInv/sec/W, which is matrix inversion per second per W. . 53

6.2 The accuracy of using the 16-bit fixed point implementation. Each unique matrix size and condition number contains ten different randomly generated matrices. Ten means all test cases are passed. A test case is considered as passed when both direct error and residual are smaller than the tolerance. Tolerance is denoted as $e_k = 2^{-k}$, where $k = 1, 2, ..., 8, 9$. For example, e_1 means the largest error is not above 2^{-1} and e_3 means the largest error is not above 2^{-3}. 0.5 is a large tolerance but most test cases are failed especially when the condition number and matrix sizes increase. Therefore, it is not usable practically. 56

6.3 The accuracy of using the 32-bit fixed point implementation. Each unique matrix size and condition number contains ten different randomly generated matrices. Ten means all test cases are passed. A test case is considered as passed when both direct error and residual are smaller than the tolerance. Tolerance is denoted as $e_k = 2^{-k}$, where $k = 1, 2, ..., 8, 9$. For example, e_1 means the largest error is not above 2^{-1} and e_9 means the largest error is not above 2^{-9}. For all test cases the max error is not greater than 2^{-6}. It can be used if a real application does not need very high precision. 57

6.4 The accuracy of using the 32-bit float point implementation. Each unique matrix size and condition number contains ten different randomly generated matrices. Ten means all test cases are passed. A test case is considered as passed when both dierect error and residual are smaller than the tolerance. Tolerance is denoted as $e_k = 2^{-k}$, where $k = 1, 2, ..., 8, 9$. For example, e_1 means the largest error is not above 2^{-1} and e_9 means the largest error is not above 2^{-9}. For all test cases the max error is not greater than 2^{-9}, which is the most accurate one among all three data types. 58

7.1 Details of general-purpose processor and GPU utilized for matrix inversion comparison. 63
7.2 Multiple matrices simulation results for the many-core implementation. All results are unscaled raw data in 32 nm CMOS. 64
7.3 The scaling factor used to convert original raw data to 14nm numbers. For example, the second row indicates the scaling factors used to scale the data from 32 nm to 14 nm. 65
7.4 Comparison of fabrication technology, key throughput and area across 5 matrix sizes for three hardware platforms. Throughput and area data are scaled to 14 nm CMOS. Throughput is measured as matrix inversions per second (MatInv/sec). Raw data is given in Table 7.2. 68
7.5 Matrix inversions per energy for various platforms. To ensure a fair comparison, numbers in column 4 and column 5 are scaled to 14 nm CMOS. The many-core implementation offers the highest matrix inversions per energy on all different matrix sizes. Raw data is given in Table 7.2. 69

8.1 The accuracy of using 16-bit half-precision floating point. Each unique matrix size and condition number contains ten different randomly generated matrices. Ten means all test cases are passed. A test case is considered as passed when both direct error and residual are smaller than the tolerance. Tolerance is denoted as $e_k = 2^{-k}$, where $k = 1, 2, ..., 8, 9$. For example, e_1 means the largest error is not above 2^{-1} and e_7 means the largest error is not above 2^{-7}. For all test cases the max error is not greater than 2^{-3}. It can be used if a real application does not need very high precision. 72

Chapter 1

Introduction

1.1 Motivation

Matrix inversion of the form $A^{-1}A = I$ is a fundamental tool of linear algebra, which is useful in many areas of applied mathematics, statistics, physics, economics, and engineering. With the increased size of matrix used in a wide variety of applications, the speed of calculating matrix inversion becomes the bottleneck that limits the performance of overall systems, especially in many scientific and engineering applications, such as image signal processing and MIMO system. Therefore, it is crucial to find a way to accelerate the process of calculating the inversed matrix.

Using parallel processing for matrix inversion is challenging and not as efficient as other matrix operations such as multiplication and transpose. There is a lot of data dependency during the calculation. Many custom implementations have been developed to exploit parallel computing in matrix inversion. There are many architectures and algorithm have been proposed on general-purpose processors [2], GPUs [3], FPGAs [4], and many-core platforms [5], [6], which indicates the current trend of increased parallelism in high performance computer architectures. However, with the increased number of cores used energy usage and die area also increases dramatically. It is necessary to develop an energy-efficient method and small area usage method on the many-core platform.

Therefore, this thesis presents a small area used and energy-efficient, scalable matrix inversion method, which was simulated on a fine-grained many-core array of low-power, simple Multiple Instruction Multiple Data (MIMD) processors (AsAP3). The method contains two matrix

inversion implementations. Each of the implementations consists of small modular program kernels operating on each core, making them scalable to different array sizes and data types.

1.2 Thesis Organization

The remainder of this thesis is organized as follows.

Chapter 2 goes over the definition of the matrix inversion and introduces Gauss Jordan Elimination. This algorithm is used to implement the matrix inversion on the many-core platform. Then, the representation for the fixed point is reviewed.

Chapter 3 introduces the target fine-grained many-core architecture, AsAP3 (KiloCore), which is used throughout the thesis.

Chapter 4 first gives the architecture of the proposed many-core matrix inversion methods, followed by different kernels that are mapped to the processors in AsAP3.

Chapter 5 explains how data is scaled during the input and calculation phase. Then, it shows the relationship between mean error and different input scaling.

Chapter 6 presents the throughput per watt comparison for all the implementations on a many-core processor array (AsAP3) and compares the accuracy between different data types.

Chapter 7 presents the simulation results of implementation on AsAP3, compared to the implementations on Intel general-purpose processor and Nvidia GPU.

Chapter 8 gives a summary of the thesis and some ideas for future work.

Chapter 2

Background

2.1 Definition of Matrix Inversion

Matrix inversion is an essential concept in matrix theory. For a matrix A with size $n \times n$, if there exists another matrix B with size $n \times n$ that makes $A \times B = B \times A = I$, we say the matrix A is invertible, and matrix B is the inverse matrix of matrix A notated as A^{-1}.

There are various ways to invert a matrix, such as LU decomposition, QR decomposition, and Gauss Jordan Elimination. The LU decomposition and Gauss Jordan Elimination are widely used because these two algorithms are suitable for any invertible dense matrix. The computational complexity for both algorithms is $2N^3$.

2.2 Gauss Jordan Elimination

2.2.1 Sequential Gauss Jordan Elimination with partial pivoting

The basic idea of Gauss Jordan elimination is to make a partitioned matrix $[A|I]$ where A is the input matrix with size $n \times n$, and I is the identity matrix with size $n \times n$ as well. Then, use elementary row operations to convert a matrix into reduced row-echelon form. The matrix A becomes an identity matrix, and the matrix I becomes the inverse of matrix A.

$$[A \mid I] \rightarrow [I \mid A^{-1}]$$

The result will not come out until completes n times iterations, where n is equal to the input matrix's rank. For example, for a 3×3 matrix, it needs three iterations to calculate the final result.

In each iteration i, the following steps are needed to be done on the partitioned matrix $[A\ I]$:

1. Find the largest absolute value in column i and swap its row with the pivot row.
2. Normalize the updated pivot row by dividing each element in the pivot row by the pivot element.
3. Eliminate all the elements that are in the same column with the pivot element.
4. Increment i and repeat these steps until i equals the rank of the input matrix.

The Figure 2.1 shows how to invert a 3×3 matrix using Gaussian Jordan elimination step by step.

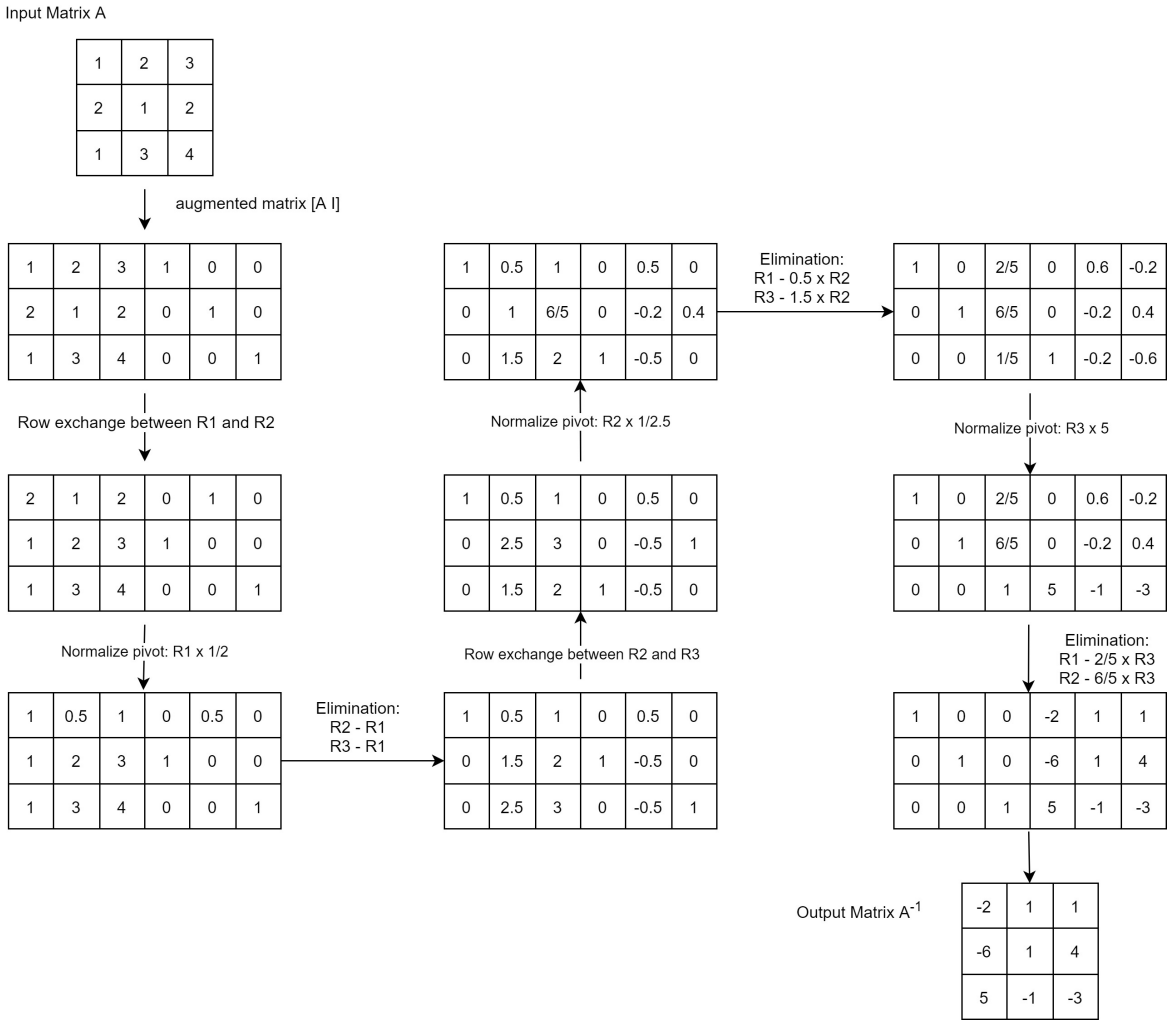

Figure 2.1: Invert a 3 by 3 matrix using Gaussian Jordan elimination step by step.

Algorithm 1 Pseudo code for sequential Gauss Jordan elimination with partial pivoting.

Require: matrix A , Rank

Ensure: inverse matrix A^{-1}

1: **function** GJE($A, Rank$)

2: $M \leftarrow [A\ I]$

```
3:   for step = 0 → rank do
4:     pivot_row ← step
5:     max_value ← 0
6:     for i = step → rank do                          ▷ finding the next pivot
7:       if |M[i, step]| > |M[step, step]| then
8:         max_value ← M[i, step]
9:         pivot_row ← i
10:      end if
11:    end for
12:    M[step, :] ↔ M[pivot_row, :]                    ▷ Row exchange
13:    for j = step → 2 × rank do                      ▷ Normalization Phase
14:      M[step, j] ← M[step, j] ÷ M[step, step]
15:    end for
16:
17:    for i = 0 → rank do                             ▷ Elimination Phase
18:      if i! = step then
19:        factor ← M[i, step]
20:        M[i, :] = M[i, :] − factor × M[step, :]
21:      end if
22:    end for
23:  end for
24: end function
```

2.2.2 Parallel Gauss Jordan Elimination

As shown in Algorithm 1, there is a data dependency between each iteration, which means the whole process has to be done in sequence. However, in the same iteration, there is no data dependency. There are 4 phases in each iteration: selecting pivot, row swap, normalizing the pivot row, and elimination. The most computational efforts are spent on the normalization and elimination phases. For a $n \times n$ matrix, in each iteration, the normalization phase requires processing 2n elements, and the elimination phase requires processing $(n-1)$ rows with $2n$ elements in each row. Without any paralleling, the overall complexity of the program is proportional to $n \times (n-1) \times 2n$,

notated as $O(n^3)$.

Since there is no data dependency in the normalization and elimination phases, we can distribute the work to different threads, both column-wise and row-wise. In the normalization phase, spawn m threads normalize the whole pivot row that each thread only processes n/m elements. In the elimination phase, $(n-1)$ rows can be distributed to different threads, and each element can be distributed to different threads. The parallel version of Gauss Jordan Elimination is shown in Algorithm 2.

Algorithm 2 Pseudo code for parallel Gauss Jordan elimination with partial pivoting.

Require: matrix A , Rank

Ensure: inverse matrix A^{-1}

function $GJE\_parallel(A, Rank)$
  $M \leftarrow [A\ I]$
  for $step = 0 \rightarrow rank$ **do**
    $pivot\_row \leftarrow step$
    $max\_value \leftarrow 0$
    for $i = step \rightarrow rank$ **do**
      if $|M[i, step]| > |M[step, step]|$ **then**
        $max\_value \leftarrow M[i, step]$
        $pivot\_row \leftarrow i$
      end if
    end for
    $M[step, :] \leftrightarrow M[pivot\_row, :]$
    for $j = step \rightarrow 2 \times rank$ **do** ▷ distributing the workload in Normalization phase
      $M[step, j] \leftarrow M[step, j] \div M[step, step]$
  unroll the loop and distribute the work to m threads. each thread processes $2 \times rank/m$ elements
    end for

    for $i = 0 \rightarrow rank$ **do** ▷ distribute the workload in Elimination phase
      if $i! = step$ **then**
        $factor \leftarrow M[i, step]$ broadcast factor to every threads
        $M[i, :] = M[i, :] - factor \times M[step, :]$

unroll the loop and distribute the work to $n \times m$ threads. each thread processes $(2 \times rank/m) \times ((rank-1)/n)$ elements

21: **end if**

22: **end for**

23: **end for**

24: **end function**

By doing this each thread needs to process $n \times (n-1)/m$ elements. Ideally, if there are unlimited threads, each thread only needs to process one element in each iteration, and the overall time complexity can be reduced to $O(n)$. In the next chapter, the target many-core platform, which is used to reach the high parallelism, is introduced.

2.3 Fixed Point

The fixed point data type is essentially an integer scaled by an implicit specific factor determined by the type. For example, an unsigned binary number 1100 can represent 12 in decimal if there is no fraction bit. If using 3 bit as fraction part, it can represent 1.5 in decimal. The fixed point numbers in this thesis are represented as two's complement, shown in Table 2.1. A fixed point number with N-bit width is represented as m.n, where m is the number of bits used to indicates the integer part and n is the number of bits for the fraction part.

m (integer width)	n (fraction width)

Table 2.1: Notation of an N-bit two's complement fixed point with m.n format, where $N = m + n$, m bits are used to represent the integer part, and n bits are used to represent the fraction part.

An N-bit two's complement fixed-point in m.n format is shown in Eq. 2.1. The implicit fraction point is between a_n and a_{n-1}.

$$a_{n+m-1}a_{n+m-2}.....a_n \ . \ a_{n-1}.....a_1a_0 \tag{2.1}$$

Eq. 2.2 is used to convert a fixed point two's complement shown in Eq. 2.1 to a decimal number.

$$-a_{n+m-1} \times 2^{m-1} + a_{n+m-2} \times 2^{m-2} + ... + a_1 \times 2^{m-(N-1)} + a_0 \times 2^{m-N} \tag{2.2}$$

16-bit and 32-bit are two common data length used in computer architecture. The Table 2.2 shows the range and precision of 16-bit fixed point with different fraction width. To balance the

Table 2.2: Range and precision for 16-bit fixed point.

Format	**Precision**	**Range**
16.0	1	(-32768,32767)
15.1	0.5	(-16384,16383.5)
14.2	0.25	(-8192,8191.75)
13.3	0.125	(-4096,4095.875)
12.4	0.0625	(-2048,2047.9375)
11.5	0.03125	(-1024,1023.96875)
10.6	0.015625	(-512,511.984375)
9.7	0.007813	(-256,255.9921875)
8.8	0.003906	(-128,127.9960938)
7.9	0.001953125	(-64,63.99804688)
6.10	0.0009765625	(-32,31.99902344)
5.11	0.00048828125	(-16,15.99951172)
4.12	0.00024414062	(-8,7.999755859)
3.13	0.00012207031	(-4,3.99987793)
2.14	0.00006103515	(-2,1.999938965)
1.15	0.00003051757	(-1,0.999969482)

input range and precision, the proposed implementations use 8.8 for 16-bit fixed point and use 16.16 for 32-bit fixed point.

Chapter 3

AsAP Platform

This thesis proposes an implementation of matrix inversion based on Gauss Jordan Elimination with partial pivoting on AsAP3 platform. AsAP3 is the third generation of Asynchronous Array of Simple Processors [7], which is a fine-grained many-core platform that includes 1000 independent, in order, single-issue processors and 12 independent 64KB memory modules. Therefore, it is also called KiloCore.

3.1 Processors

There are a 128×40-bit instruction memory, a 256×16-bit data memory, three programmable address generators, two 32×16-bit input FIFO buffers and a 16×16 fixed point multiplier with the 40-bit accumulator (MAC) in each processor [7]. Each processor also supports 72 instruction types includes both singed and unsigned operations, which are not algorithm specific. Moreover, the two conditional execution masks, static branch prediction and hardware automated looping for accelerating inner loops, are supported by every processors. Though the natural word width of the data path and memory are 16-bit, other data types such as 32-bit fixed point and floating point are supported easily through software.

In every clock cycle, each processor issues a single 40-bit instruction into its seven stage pipeline from either its local instruction memory or from on-chip independent memory module if programs are large. The input operands of instructions are normally from the two input FIFO buffer and from the local data memory. The output results goes to the local data memory, one or multiple of circuit-switched network ports, the packet router ports, a neighboring on-chip memory

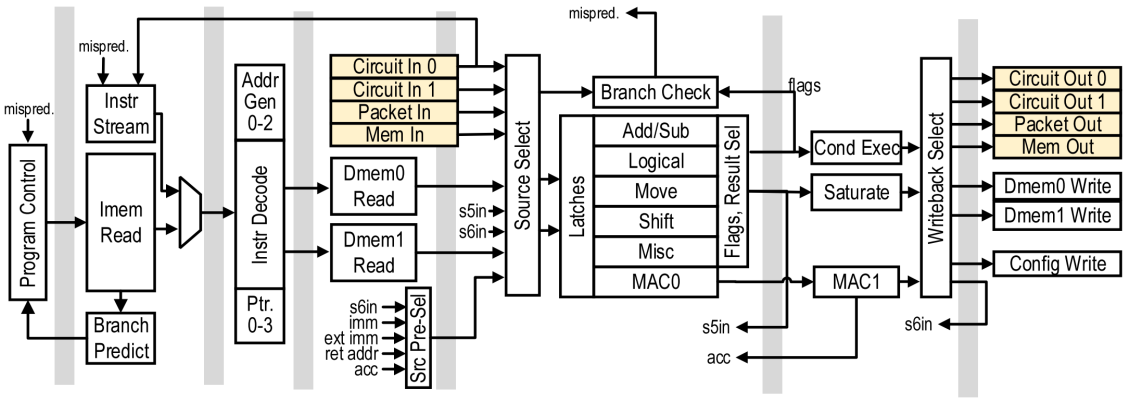

Figure 3.1: Major components and connections of the seven-stage processor pipeline [1].

module, a pipeline forwarding path and special registers which used for dereferenceable pointers, address generator, oscillator frequency selection and other software-accessible core configuration fields. The pipeline structure is shown in Figure 3.1.

3.2 Inter-Processor Communication

The processors and independent memory modules on AsAP platform are connected through 2-D mesh, a topology which maps well to planar integrated circuits and scales simply as the number of processors per die increases [8]. Communication on-chip is accomplished by two complementary means: a very high throughput and low-latency circuit-switched network [9] and a very-small-area packet router [10]. Details are provided in Figure 3.2. The network supports communication between adjacent and distant Processors, as resources permit, with each link supporting a maximum 28.5Gb/s transimision rate with optionally inserted registers to maintain data integrity over long distances. The circuit-switched links are source-synchronous, so the source clock travels with the data to the destination, where it is translated to the destination of processsor's clock domain. The packet router inside each processor occupies only 9% of each processor's area and is especially effective for high fan-in and high fan-out communication, as well as for administrative messaging. Each router supports 45.5 Gb/s of throughput with a maximum of 9.1 Gb/s per port. Routers operate autonomously from their host processors and contain their own clock oscillators, so they can power down to zero active power when there are no packets to process. Each circuit or packet link terminates in a dual-clock FIFO memory, which reliably transfers data between clock domains [11].

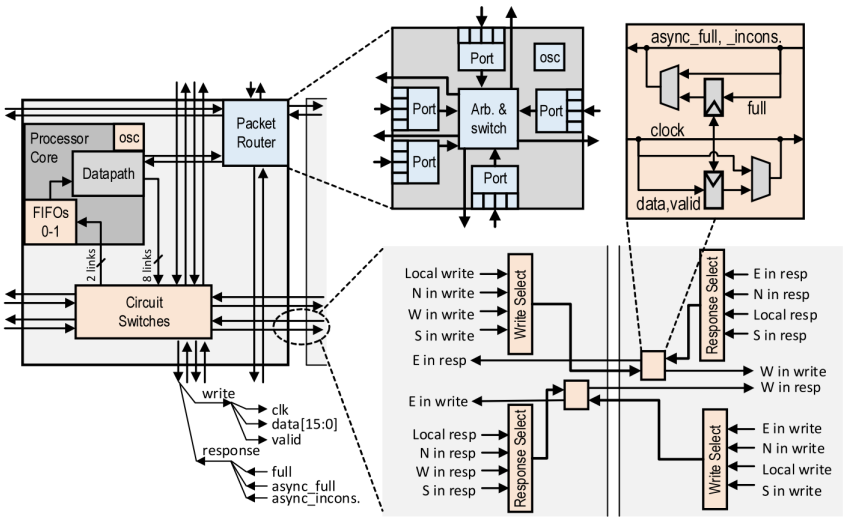

Figure 3.2: Overview of inter core communication using circuit and packet networks. Writes are source-synchronous; responses include asynchronous wake-up signals for sleeping processors. Circuit links include configurable registers and an eastwest connection for one layer is expanded on the right [1].

Moreover, links contain the necessary asynchronous wake-up signals, which inform idle modules when they need to activate their local clock to process new work or to verify when FIFOs are full or empty.

3.3 Independent Memory Modules

Besides the data memory in each processors, another on-chip storage is independent memory module. There are 12 independent memory modules on the many-core platform, named as BigMem. Each independent memory module contains a 64 KB SRAM and is shared between two neighboring processors. Modules support random and a variety of programmable burst access patterns for data reading and writing, and are also capable of streaming instructions for large-program execution to an adjoining processor using an internal control module. When executing an instruction stream from an independent memory, a processor transfers program control and branch prediction control to dedicated circuits inside the memory block to more efficiently execute across branches. Each memory module contains two 32×18-bit input buffers, two 32×16-bit output buffers, and one 16×2-bit processor response buffer, and supports 28.4 Gbit/s of I/O bandwidth. The details of the module's internal blocks are shown in Figure 3.3.

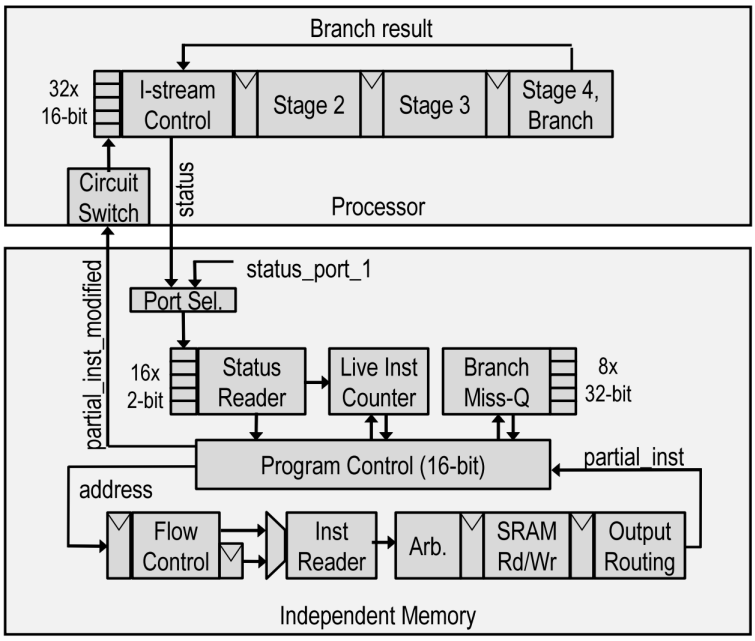

Figure 3.3: Components used in streaming instructions from a shared memory to a neighboring processor. Streaming logic is shared between two processors, with only the port 0 connection shown here [1].

3.4 Fine-Grain Clocking

Many-core applications often require processors to remain idle or operate at low activity for substantial periods of time. In KiloCore, each core, each packet router inside each core and each independent memory module contains its own local programmable clock oscillator in an independent fully synchronous clock domain [12]. Each oscillator is allowed to change its frequency, halt or restart arbitrarily including with respect to other clock domains. Halting is very helpful in saving energy when there is no work to do.

3.5 KiloCore (AsAP3)

The KiloCore chip [13] is a processor array that containing 1,000 independent processors and 12 memory modules, which was fabricated in 32-nm partially depleted silicon on insulator CMOS. Processors are arrayed in 32 columns and 31 rows with eight processors and 12 independent memories in a 32nd row as shown in Figure 3.4. Processors and independent memory modules with no work to do dissipate exactly zero active power (leakage only)–this is an important capability in the 1000-processor era due to the difficulty in implementing complex software workloads that spread

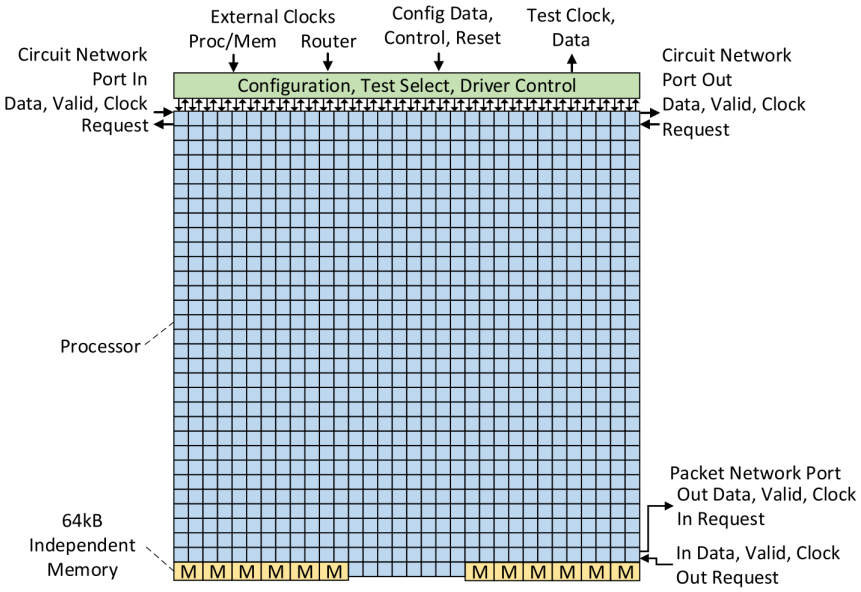

Figure 3.4: Top-level processor array of KiloCore (AsAP3) [1].

evenly over thousands of processors, which leads to the increasing prevalence of processors with widely varying activity levels [14]. Under most conditions, the processors array has a near-optimal proportional scaling of power dissipation over a wide range of activity levels. The Kilocore has a total of 2012 globally asynchronous locally synchronous (GALS) [15] clock domains.

Chapter 4

Implementations of Matrix Inversion on a Many-Core Platform

4.1 Matrix Inversion Kernels

The matrix inversion implementations, described in this chapter, utilize basic kernels in each processor of the array. Each kernel is designed for generalization, which can be easily mapped to any processor in any part of the array. The overall implementations are designed as a parameterized type, taking any dimension of the matrix without recompiling. This chapter will discuss two architectures: InversionBigMem and InversionExternalMem. InversionBigMem uses eight 64 KB on-chip memories, which can take an input matrix as large as 320×320 with the 16-bit fixed point. InversionExternalMem uses eight 8 GB off-chip memories with 100 ns access latency to store data and eight on-chip memories as buffers, taking an input matrix as large as 1024×1024, which is bounded by the AsAP processor's DataMem. The proposed design spreads a row of data to 64 cores evenly. So each processor's DataMem needs to hold both data and variables used in the kernel. 1024 is the tested largest number that makes the kernel can be compiled. There are two versions for the external memory architecture: the 32-bit fixed point version and the 32-bit single float point version. The kernels for different versions can be reused, and only the arithmetic operators are different for the different data types. Since two different memory architectures are proposed in this thesis, the data distribution and memory interface kernels are different. This chapter will discuss the block diagram of two architectures and the programs that make up these

blocks as the list below:

- Block diagram
  - InversionBigMem
  - InversionExternalMem
- Input Distribution
- Memory Architecture
- Sort
- Logic Control
- Matrix Operation
  - Normalization Chain
  - Elimination Array

4.2 Block Diagram

4.2.1 InversionBigMem

InversionBigMem is an architecture that uses only on-chip BigMem to store data and the block diagram is shown in Figure 4.1 The Input distribution kernel distributes input data evenly to different on-chip memories. The logic control sends the command to each memory to ask them to push data into computation blocks: Normalization kernel and Elimination kernel. The Normalization kernel and Elimination kernel do some matrix operations and send processed data back to update memory. The sort kernel finds the next pivot row and sends the flag to logic control. After an n times iteration where n is the rank of the input matrix, the results are sent out by each BigMem in sequence. Division only needs to be calculated once during the beginning of each iteration. Therefore, the latency of division will not limit the overall performance. A detailed version of the block diagram is shown in Figure 4.2. Div is the core that uses the built-in divider (division operator in C++, the compiler can compile it) to calculate the division only. Since each processor can only have two circuit link input ports, $Buffer$ is the simple kernel loaded to the

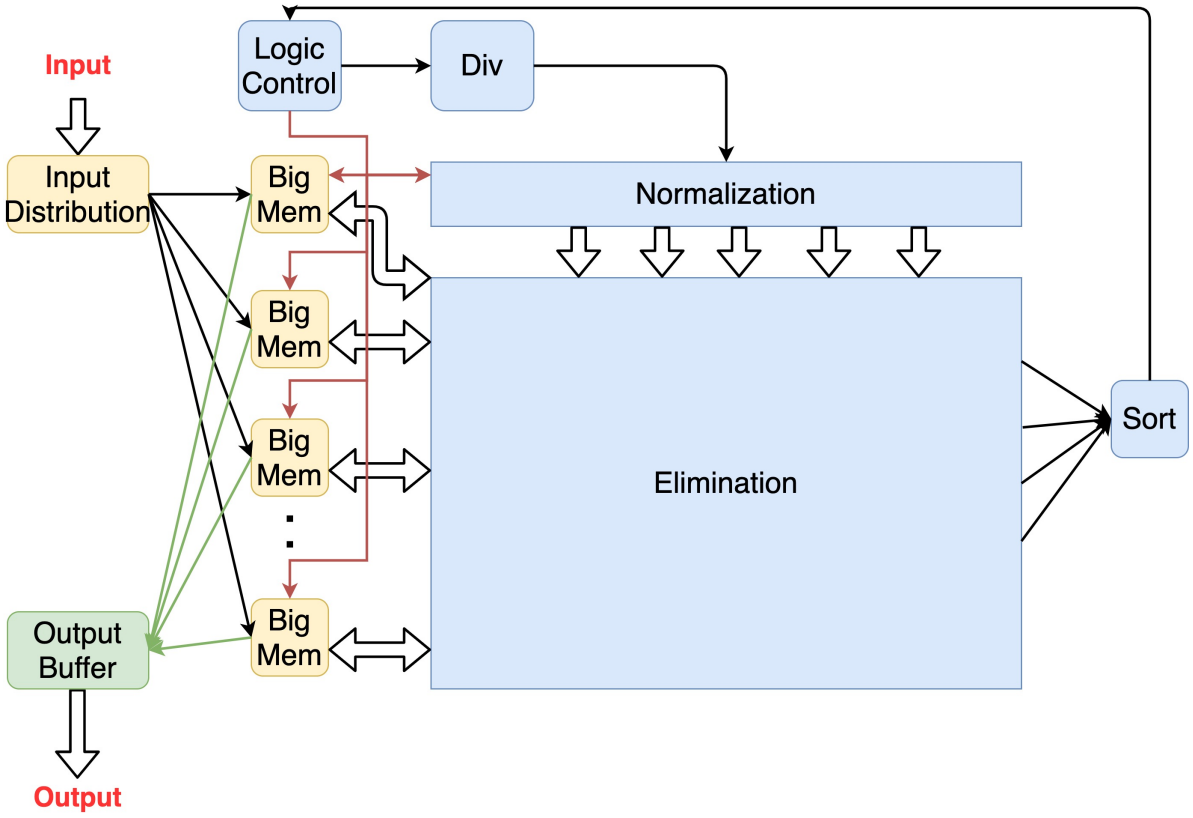

Figure 4.1: Block diagram of InversionBigMem architecture. Input data is loaded evenly to different BigMems through InputDistribution block. All data is processed through Normalization and Elimination blocks. The LogicControl sends commands to different Memory modules about which part of data is going to be processed. Memory modules push data to Normalization and Elimination blocks and update memory with processed data. Div block is the core uses built-in divider ("/", division operator in c++) to calculate the division only. In each loop, Sort block finds the max pivot and sync with the LogicControl to start next iteration. After all the computations are done, each memory module output the results in sequence. The details of each block are shown in later sections.

AsAP processors to merge two inputs into one. $PacketBuffer$ is the AsAP processor that reads from the router and output data through the circuit link. $Trash$ is used to only read input but not output anything. The $Trash$ kernel is designed to be used at the end of a chain to receive some trash data. For example, if all cores in a chain spread data or flags to the right, the last core of the chain will spread to the right as well but there is no cores on the right. By using the $Trash$ core at the end of chain, all cores in the chain can use the same kernel, which simplifies the design process. *Spread* reads destination address first and transfer data from input to the destination core through the packet link. Besides these simple cores, the kernel for each processor shown in Figure 4.2 is explained in later sections.

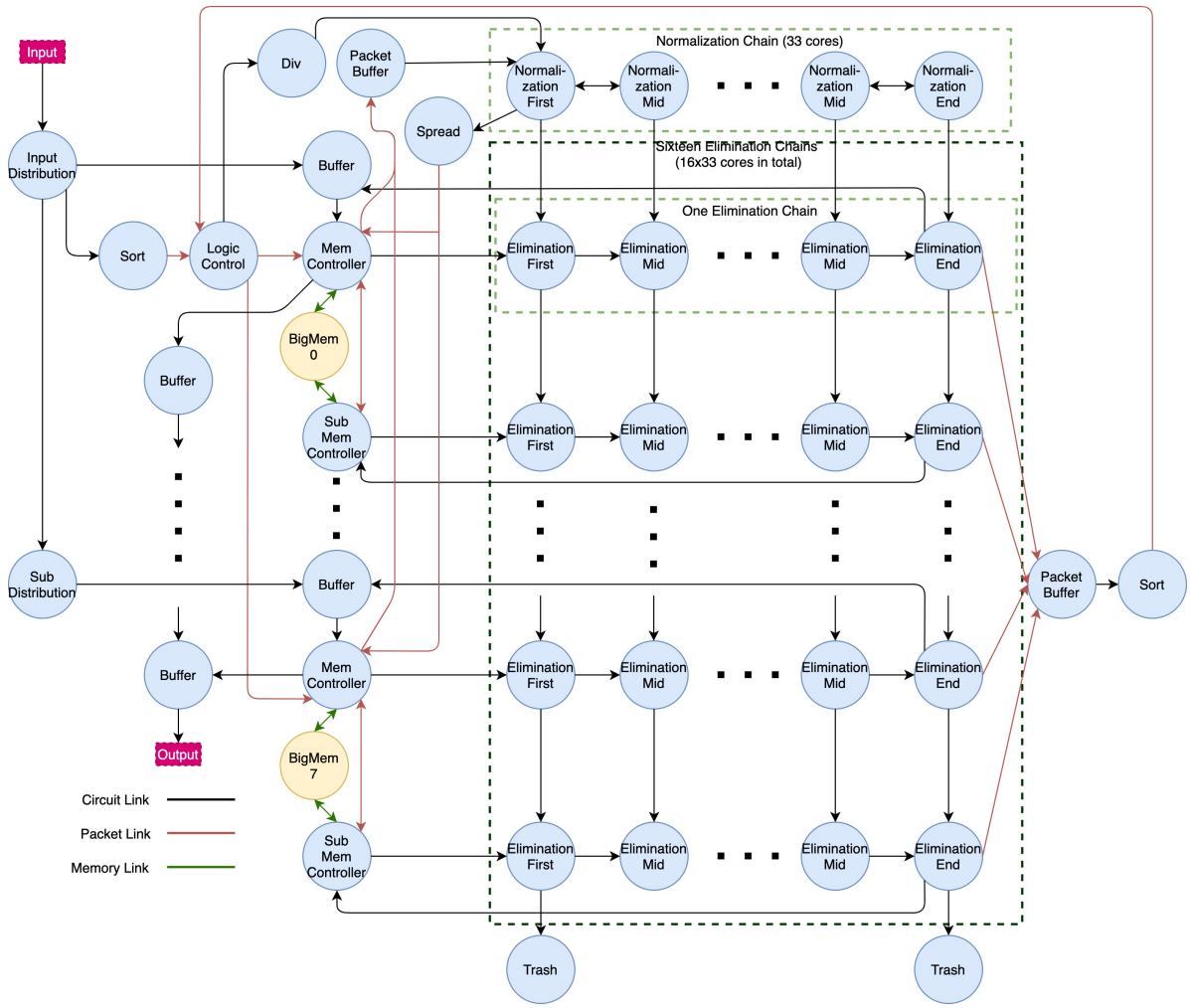

Figure 4.2: Detailed version of Figure 4.1 that shows everything inside of each block. Blue circles are AsAP processors and yellow circles are memory modules. Red lines are packet links, black lines are circuit links, and green lines are memory links. There are eight BigMems (BigMem0-7) in this architecture. For simplicity, two of them are shown in the figure. There are 603 processors used in this architecture.

4.2.2 InversionExternalMem

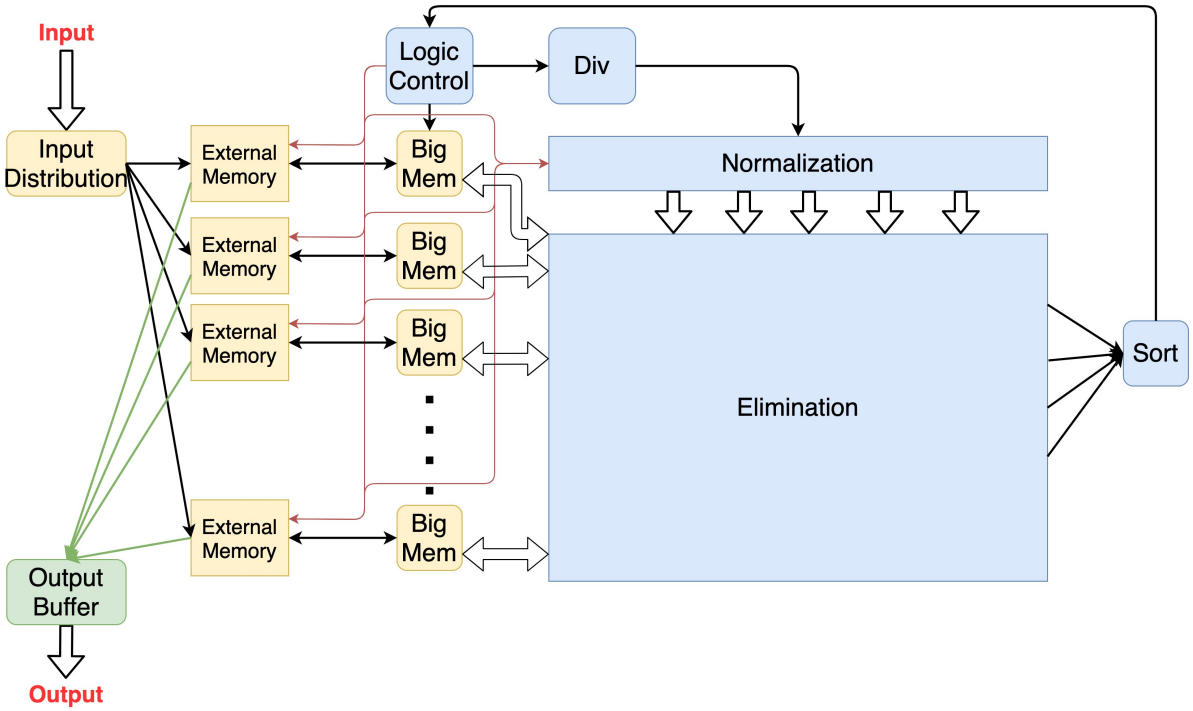

Figure 4.3: Block diagram of InversionExternalMem. The overall data flow of InversionExternalMem is similar to the InversionBigMem. The difference is that all data is stored in the external memories. The external memory has a 100 ns access latency. During the calculation, the BigMem works as buffer to store part of data and write back to the external memories after the data is processed. Div block is the core uses built-in divider ("/", division operator in c++) to calculate the division only.

InversionExternalMem is an architecture that uses external memories to store data and uses on-chip BigMem as buffers to compute the inverse. The block diagram is shown in Figure 4.3. Since all the program is designed for modularity, most of the kernels are similar to InversionBigMem architecture. The only difference is the memory hierarchy. Eight huge external memory modules are placed on the edge of the chip. Each external memory module has a single 16-bit I/O port. The data is sorted in the external memory first. During the computation, a block of data will be moved from the external memory to the on-chip BigMem, and the block size equals the size of BigMem, which is 64 KB. A detailed version of the block diagram is shown in Figure 4.4. *Div* is the core that uses the built-in divider ("/", division operator in c++) to calculate the division only. Since each processor can only have two circuit link input ports, $Buffer$ is the simple kernel loaded to the AsAP processors to merge two inputs into one. $PacketBuffer$ is the AsAP processor that reads from the router and output data through the circuit link. $Trash$ is used to only read input but

Kernel Name	Number of code	Number of cores loaded
Input Distribution	86	1
Sub Distribution	32	1
Sort	22	2
Logic Control	189	1
Div	5	1
Buffer	13	15
Packet Buffer	6	2
Spread	14	1
Trash	3	2
Elimination First	25	16
Elimination Mid	49	465
Elimination Mid (last row)	38	31
Elimination End	50	16
Normalization First	30	1
Normalization Mid	31	31
Normalization End	22	1
Mem Controller	94	8
Sub Mem Controller	34	8
Total	—	603

Table 4.1: The number of code for each unique kernel and the number of cores that are loaded with different kernels for the implementation with only on-chip memory. Number of code is measured as the number of lines of code in C++ after removing all the comments and blank lines.

not output anything. *Spread* reads destination address first and transfer data from input to the destination core through the packet link. Besides these simple cores, the kernel for each processor shown in Figure 4.4 is explained in later sections.

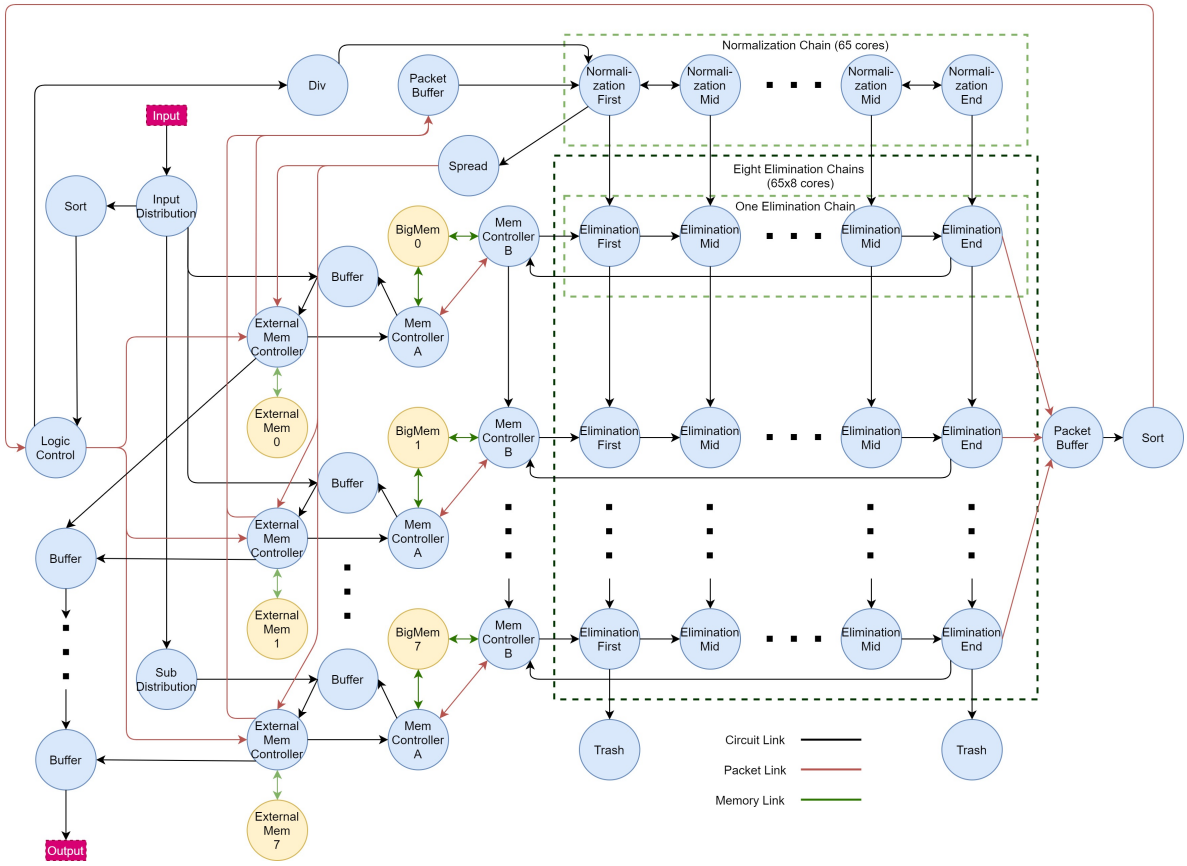

Figure 4.4: Detailed version of Figure 4.3 that shows everything inside of each block. Blue circles are AsAP processors and yellow circles are memory modules. There are eight external memories (ExternalMem0-7) and eight BigMem (BigMem0-7) in this architecture. Red lines are packet links, black lines are circuit links, and green lines are memory links. For simplicity, three of them are shown in the figure. There are 635 processors in this architecture.

4.3 Input Data Distribution

As shown in chapter 2, the Gaussian Jordan Elimination takes an augmented matrix as input. For a $n \times n$ matrix A, the augmented matrix M is $n \times 2n$. Here A is the matrix that needs to be inverted, and I is the identity matrix with $n \times n$ dimension.

$$M = [A|I]$$

Kernel Name	Number of code	Number of cores loaded
Input Distribution	86	1
Sub Distribution	32	1
Sort	28	2
Logic Control	143	1
Div	6	1
Buffer	14	15
Packet Buffer	6	2
Spread	22	1
Trash	3	2
Elimination First	25	8
Elimination Mid	40	441
Elimination Mid (last row)	38	63
Elimination End	50	8
Normalization First	36	1
Normalization Mid	32	63
Normalization End	21	1
Mem Controller A	47	8
Mem Controller B	51	8
External Mem Controller	182	8
Total	—	635

Table 4.2: The number of codes for each unique kernel and the number of cores that are loaded with different kernels for the implementation with off-chip memory. Number of code is measured as the number of lines of code in C++ after removing all the comments and blank lines.

The input file contains $N \times (2N + 1) + 1$ elements. The first element is the rank of matrix, and the next is the row index and followed by a row of data. The detailed format is shown in the table below.

rank	Index(0)	a row of data	Index(1)	a row of data	...	Index(n)	a row of data

The proposed implementations read augmented matrix M as input and distribute the input row-wise to eight different memory ports. The implementation consists of two processors that use two individual kernels. The block diagram is shown in Figure 4.5.

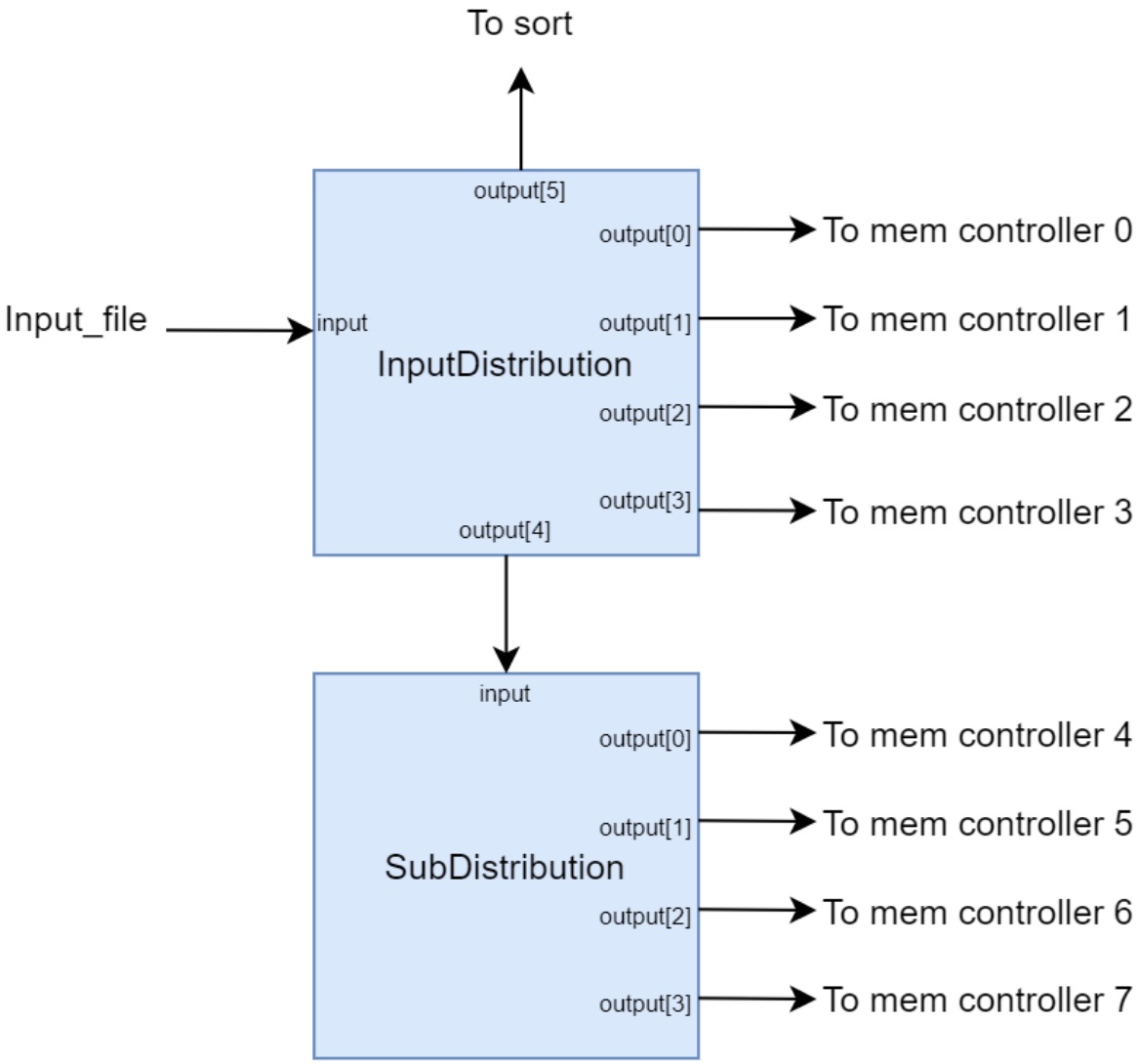

Figure 4.5: Block diagram of Input Distribution.

InputDistribution reads data from the input file and output data to 4 memory controllers, a

sort core, and SubDistribution. SubDistribution reads data from InputDistribution and output data to another four memory controllers. As shown in Algorithm 3 and Algorithm 4, InputDistribution calculates how many rows are stored in each memory and sends data row-wise to different memory controllers. It also sends the first two elements, which are RowIndex and the element in the first column, to the sorting core. This information is used to find the max value in column 0 and swapping the corresponding row with the pivot row. After that, InputDistribution will send the remaining data to SubDistribution, and SubDistribution will distribute this data to another four memory controllers.

Algorithm 3 pseudo code for Input Distribution

1: **function** INPUTDISTRIBUTION($input, output[5]$)
2: $rank \leftarrow input$
3: $RowLength \leftarrow 2 \times rank + 1$
4: $NumOfRow \leftarrow rank \div 8$
5: $remainder \leftarrow rank \mod 8$
6: **for** $i = 0 \rightarrow 8$ **do**
7: $NumOfRowInMem\_i \leftarrow (remainder > i)?NumOfRow + 1 : NumOfRow$
8: **end for**
9: **for** $i = 0 \rightarrow NumOfRowInMem\_0$ **do**
10: $RowIndex, PivotValue \leftarrow input$
11: $output[5] \leftarrow RowIndex, PivotValue$
12: $output[0] \leftarrow RowIndex, PivotValue$
13: **for** $j = 0 \rightarrow RowLength - 1$ **do**
14: $output[0] \leftarrow input$
15: **end for**
16: **end for**
17: **for** $i = 0 \rightarrow NumOfRowInMem\_1$ **do**
18: $RowIndex, PivotValue \leftarrow input$
19: $output[5] \leftarrow RowIndex, PivotValue$
20: $output[1] \leftarrow RowIndex, PivotValue$
21: **for** $j = 0 \rightarrow RowLength - 1$ **do**
22: $output[1] \leftarrow input$

```
23:     end for
24:   end for
25:   for i = 0 → NumOfRowInMem_2 do
26:     RowIndex, PivotValue ← input
27:     output[5] ← RowIndex, PivotValue
28:     output[2] ← RowIndex, PivotValue
29:     for j = 0 → RowLength − 1 do
30:       output[2] ← input
31:     end for
32:   end for
33:   for i = 0 → NumOfRowInMem_3 do
34:     RowIndex, PivotValue ← input
35:     output[5] ← RowIndex, PivotValue
36:     output[3] ← RowIndex, PivotValue
37:     for j = 0 → RowLength − 1 do
38:       output[3] ← input
39:     end for
40:   end for
41:   output[4] ← NumOfRowInMem 4, 5, 6, 7
42:   for i = 0 → ∑ NumOfRowInMem4, 5, 6, 7 do
43:     RowIndex, PivotValue ← input
44:     output[5] ← RowIndex, PivotValue
45:     output[4] ← RowIndex, PivotValue
46:     for j = 0 → RowLength − 1 do
47:       output[4] ← input
48:     end for
49:   end for
50: end function
```

Algorithm 4 pseudo code for sub Input Distribution

```
1: function SubDistribution(input, output[3])
2:   RowLength ← input
```

```
    NumOfRowInMem4, 5, 6, 7 ← input
    for i = 0 → NumOfRowInMem_4 do
        RowIndex, PivotValue ← input
        output[0] ← RowIndex, PivotValue
        for j = 0 → RowLength − 1 do
            output[0] ← input
        end for
    end for
    for i = 0 → NumOfRowInMem_5 do
        RowIndex, PivotValue ← input
        output[1] ← RowIndex, PivotValue
        for j = 0 → RowLength − 1 do
            output[1] ← input
        end for
    end for
    for i = 0 → NumOfRowInMem_6 do
        RowIndex, PivotValue ← input
        output[2] ← RowIndex, PivotValue
        for j = 0 → RowLength − 1 do
            output[2] ← input
        end for
    end for
    for i = 0 → NumOfRowInMem_7 do
        RowIndex, PivotValue ← input
        output[3] ← RowIndex, PivotValue
        for j = 0 → RowLength − 1 do
            output[3] ← input
        end for
    end for
end function
```

4.4 Memory Architecture

4.4.1 InversionBigMem

As discussed in Chapter 3, there are 12 BigMems on KiloCore, and each of them has two independent I/O ports. Figure 4.6 shows a memory interface that is used in InversionBigMem architecture. Since each core only has two circuit-switched input ports, the buffer is a processor used to collect data from two different sources and send it to the destination core through a single port. In this architecture, all input data is stored in on-chip memory through MemControllers as shown in Algorithm 5 and Algorithm 6. MemControllers store input data to the BigMem first and then receive commands such as which row will be processed from logic control through packet link. To reach a higher parallel ability, the MemController communicates with SubMemController through packet link, and these two MemControllers can process two rows of data at the same time. For example, if four rows are needed to be processed, the MemController will send the first two rows of data to the computation block, and the SubMemController will send the last two rows to computation blocks. Since the two controllers are working on different parts of memory, there is no memory conflict during reading and writing. By instantiating more structures shown in Figure 4.6, the program can invert a larger matrix and reach a higher parallel ability.

Algorithm 5 Pseudo code for MemController.

```
1: rank, NumOfRows ← input
2: for i = 0 → NumOfRows do
3:    inputs → Memory
4: end for
5: while 1 do
6:    mode ← PacketIn
7:    if mode == Normalization then
8:       RowIndex ← LogicControl
9:       Memory[RowIndex] → Normalizationkernel      ▷ push a row of data from memory to Normalization kernel
10:      Memory[RowIndex] ← NormalizationKernel      ▷ receive the processed data and write back to memory
11:   end if
```

```
12:   if mode == Elimination then
13:     NumOfRow0 = NumOfRows ÷ 2                          ▷ split data into half.
14:     NumOfRow1 = NumOfRows − NumOfRow0
15:     NumOfRow0, NumOfRow1 → SubMemController  ▷ ask the other port to process the
      other half of data
16:     for i = 0 → NumOfRow0 do
17:       Memory[i] → EliminationKernel
18:       Memory[i] ← EliminationKernel
19:     end for
20:     sync ← SubMemController
21:   end if
22:   if mode == Check then
23:     for i = 0 → NumOfRows do
24:       Memory[i] → Output
25:     end for
26:   end if
27: end while
```

Algorithm 6 Pseudo code for SubMemController.

```
1: NumOfRow0, NumOfRow1 ← MemController
2: for i = NumOfRow0 → NumOfRow1 do
3:   Memory[i] → EliminationKernel
4:   Memory[i] ← EliminationKernel
5: end for
6: sync → MemController
```

4.4.2 InversionExternalMem

The InversionExternalMem architecture has a two-level memory hierarchy shown in Figure 4.7. Data is loaded to the external memory first. The external memory has a 100 ns access latency. During the computation phase, the External Memory Controller transfers a block of data to the BigMem through the BigMemControllerA. After the transferring is done, BigMemControllerB grabs data from BigMem to computation Blocks and updates BigMem with processed data. When

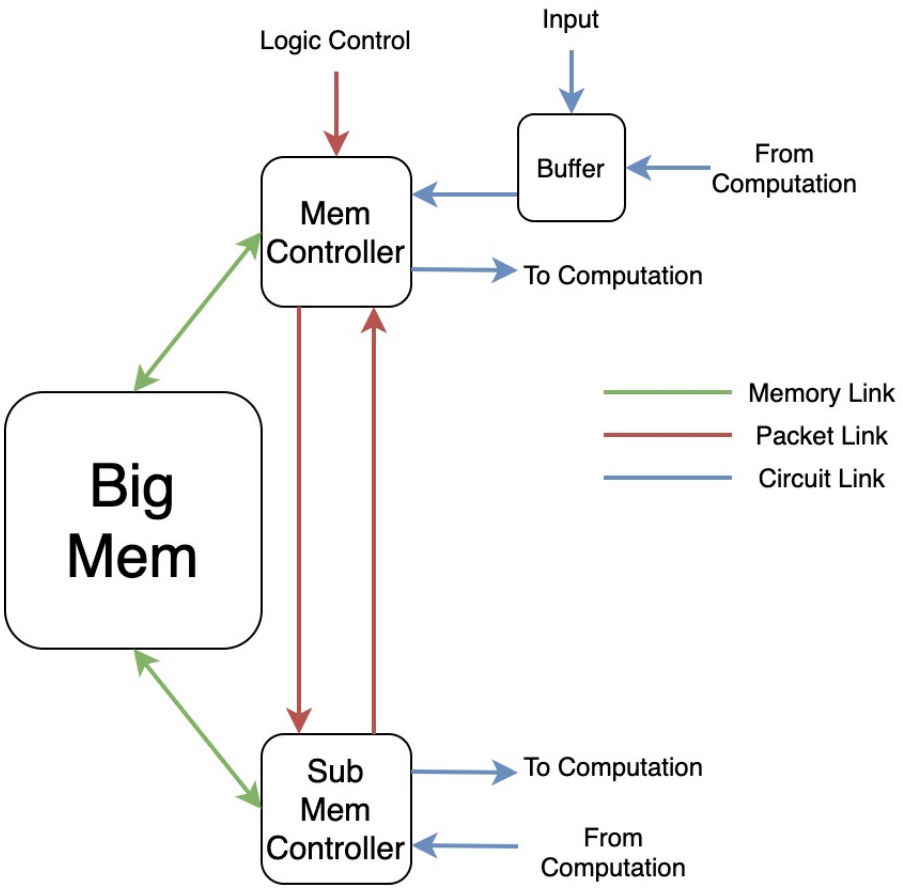

Figure 4.6: The memory interface of InversionBigMem architecture for one BigMem. MemController, SubMemController and Buffer are regular AsAP processors. Each AsAP processor can have two input ports (memory link and circuit link) and one packet link. Buffer core is used to merge two input ports into one. Instantiate more of this architecture can store a larger input matrix and improve parallelism

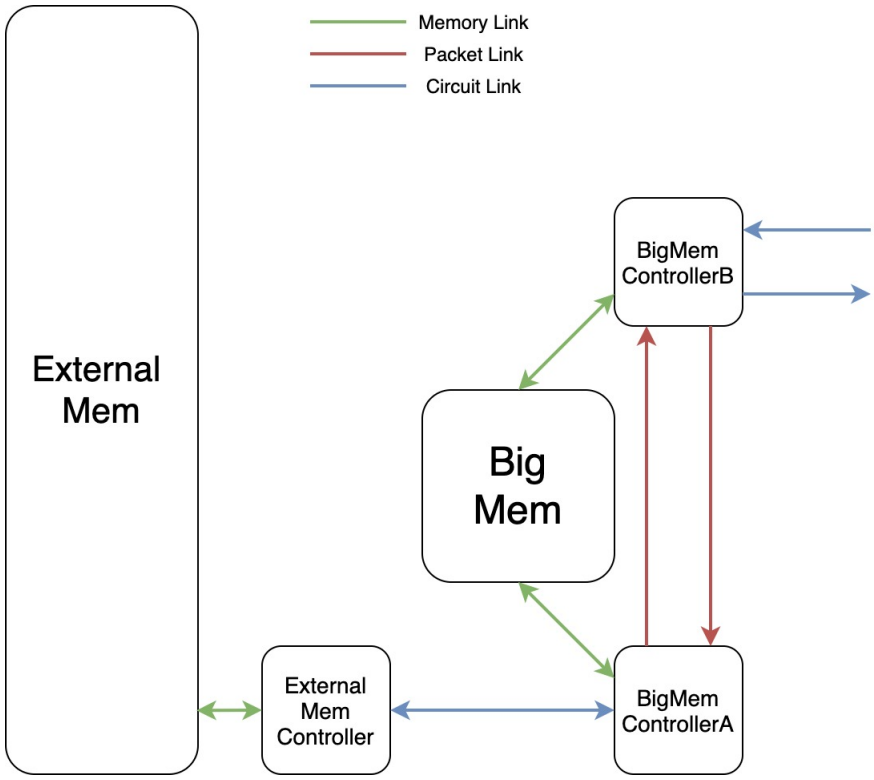

Figure 4.7: The memory interface of InversionExternalMem architecture. The figure shows the data transfer between one BigMem and one External Memory port. The external memory has a 16-bit I/O port with 100 ns access latency. To increase the bandwidth, instantiate more blocks like this. ExternalMemController, BigMemControllerA and BigMemControllerB are regular AsAP processors. Each AsAP processor can have two input ports (memory link and circuit link) and one packet link. Packet link is only used to transfer some low throughput information.

all data in BigMem has been processed, BigMemControllerA sends data back to External Memory. The Algorithm 7 shows the mechanism of how to access the external memory. The external memory has a 64 KB max burst size, which equals the size of a BigMem. Since the data is stored and processed in row-major, the memory is split into different blocks. Each block contains as many rows as possible. During the Normalization phase, the external memory controller receives the pivot row address, grab data from external memory, sends that row to the Normalization blocks and receives the processed row from the Normalization block to update memory. During the Elimination phase, the external memory controller transfers a block of data to Bigmem for computing. After the computation is done, the external memory controller will transfer the processed data back and transfer the next data block until all data has been processed.

The overall process of using on-chip BigMem as buffers to process the data can be split into three steps. First, BigMemControllerA receives data from external memory, writes data

to BigMem, and sends some flags to BigMemControllerB when the transfer is done. Secondly, BigMemControllerB push data from BigMem to computation Kernel and update BigMem with computed data. After updating all the data, send a ready flag back to BigMemControllerA. Lastly, when controller A receives the flag, it sends all the data back to external memory. To reduce the latency, instead of sending the start flag to BigMemControllerB after writing all the data to BigMem, the BigMemControllerA sends the start flag after the first row has finished written to the BigMem because writing to BigMem is faster than processing and updating the memory. The detailed process is shown in Algorithm 8 and Algorithm 9.

Algorithm 7 Pseudo code for ExternalMemController.

$TotalRows \leftarrow input$
$BlockRows \leftarrow MaxBurstSize \div RowLength$
$NumOfBlock \leftarrow TotalRows \div BlockRows$
$RemainingRows \leftarrow TotalRows \mod BlockRows$
$BurstSize \leftarrow BlockRows \times RowLength$
for $i = 0 \rightarrow NumOfBlock$ **do**
    $StartAddress \leftarrow i \times BlockRows \times RowLength$
    $Memory \leftarrow write, StartAddress, BurstSize$
    for $j = 0 \rightarrow BurstSize$ **do**
        $Memory \leftarrow input$
    end for
end for
while 1 **do**
    $mode \leftarrow LogicControl$
    if $mode == Normalization$ **then**
        $RowIndex \leftarrow LogicControl$
        $Memory[RowIndex] \rightarrow Normalizationkernel$
        $Memory[RowIndex] \leftarrow NormalizationKernel$
    end if
    if $mode == Elimination$ **then**
        for $i = 0 \rightarrow NumOfBlock$ **do**
            $StartAddress \leftarrow i \times BlockRows \times RowLength$

```
        Memory ← read, StartAddress, BurstSize
        for j = 0 → BurstSize do
          Memory → BigMemControllerA
        end for
        Memory ← write, StartAddress, BurstSize
        for j = 0 → BurstSize do
          Memory ← BigMemControllerA
        end for
      end for
    end if
    if mode == Check then
      for i = 0 → NumOfBlock do
        StartAddress ← i × BlockRows × RowLength
        Memory ← read, StartAddress, BurstSize
        for j = 0 → BurstSize do
          Memory ← output
        end for
      end for
      break while loop
    end if
end while
```

Algorithm 8 Pseudo code for BigMemControllerA.

```
NumOfRows, RowLength, BurstSize ← ExternalMemController
write, StartAddress(0), BurstSize → BigMem
for i = 0 → RowLength do
  BigMem ← ExternalMemController
end for
NumOfRows, RowLength → BigMemControllerB
for i = 0 → BurstSize − RowLength do
  BigMem ← ExternalMemController
end for
```

10: $sync \leftarrow BigMemControllerB$

11: $read, StartAddress(0), BurstSize \rightarrow BigMem$

12: **for** $i = 0 \rightarrow BurstSize$ **do**

13: $BigMem \rightarrow ExternalMemController$

14: **end for**

Algorithm 9 Pseudo code for BigMemControllerB.

1: $NumOfRows, RowLength \leftarrow BigMemControllerA$

2: **for** $i = 0 \rightarrow NumOfRows$ **do**

3: $StartAdders \leftarrow i \times RowLength$

4: $BurstSize \leftarrow RowLength$

5: $read, StartAddress, BurstSize \rightarrow BigMem$

6: **for** $j = 0 \rightarrow BurstSize$ **do**

7: $BigMem \rightarrow EliminationKernel$

8: **end for**

9: $write, StartAddress, BurstSize \rightarrow BigMem$

10: **for** $j = 0 \rightarrow BurstSize$ **do**

11: $BigMem \leftarrow EliminationKernel$

12: **end for**

13: **end for**

14: $flag\_ready \rightarrow BigMemControllerA$

4.5 Sort

The Sort kernel obtains multiples data column-wise and returns the max absolute value and its row index. In order to reduce the error, the row with the largest leading element needs to be swapped to the next pivot location before starting the next iteration. As shown in Figure 4.8, *SortRange* are the rows below the current pivot rows.

The Sort Kernel is shown in Algorithm 10. It gets *rank* and *PivotIndex* as configuration. Based on these two variables, the *SortRange* can be calculated.

$$SortRange = Rank - PivotIndex \tag{4.1}$$

Then, it runs a for loop with iteration equals *SortRange*. Each iteration reads in two values: the

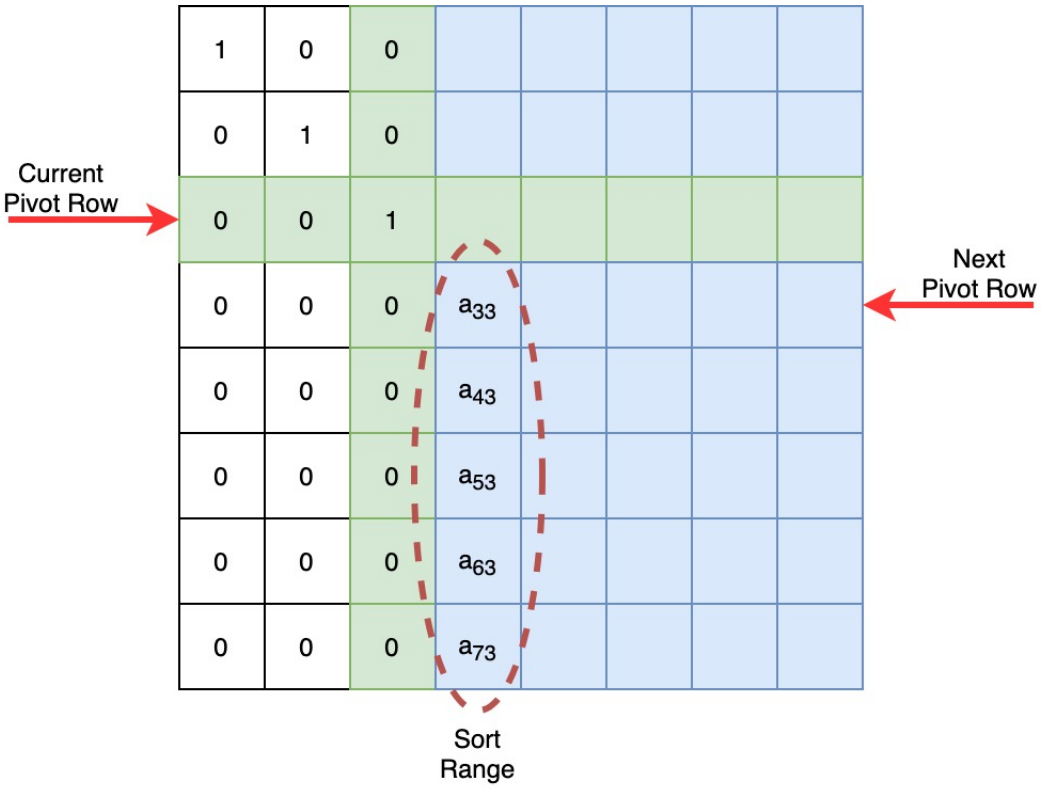

Figure 4.8: An example that shows the sort range for each iteration. The rank of the example matrix is eight, and the current pivot index is 3. Before starting the next iteration, the Sort kernel needs to find the element with the max absolute value in the red circle. That element with its corresponding row will be the new pivot row in the next iteration.

matrix element and its rowIndex and returns the max element and its corresponding index when the iteration is done.

Algorithm 10 Pseudo code for sort kernel.

```
1:  function SORT(input,output)
2:      Rank, pivotRow ← input
3:      SortRange ← Rank − PivotIndex
4:      Max ← 0
5:      newPivotIndex ← PivotIndex
6:      for i = 0 → SortRange do
7:          RowIndex, value ← input
8:          if abs(value) >= abs(max) then
9:              max ← value
10:             newPivotIndex ← RowIndex
11:         end if
12:     end for
13:     Return max, newPivotIndex
14: end function
```

4.6 Logic Control

As discussed in Chapter 2, there are 4 phases in each iteration: finding pivot, row swap, normalization, and elimination. Since there is a data dependency between each phase, these 4 phases are done in sequence. The purpose of the logic control kernel is to maintain this sequence, send commands to different processors, and sync with them. According to the Algorithm 11, the logic control kernel first reads some information from the input, including the input matrix dimension and the number of rows of data stored in each memory. Then, it starts an iteration with the size equals the rank of the input matrix. In each iteration, the kernel will receive the new pivot value and its row index, send a command to memory controllers to swap this row with pivot row, normalize the pivot and do a row elimination. After the iteration is done, the kernel will send the command to all the memory controllers to output the results from memories. Swapping the row in memory requires a lot of time and energy. Instead of physically switching them in memory, the proposed

implementation tracks each row's address and only changes the RowIndex.

Algorithm 11 Pseudo code for logic control.

1: $rank, RowsInEachMem \leftarrow input$

2: **for** $i = 0 \rightarrow rank$ **do**

3: $newPivotRow, newPivot \leftarrow input$

4: $memI \leftarrow findlocation(newPivotRow)$ ▷ calculate which memory contains the data of new pivot row

5: $newPivot \rightarrow Div\ core$

6: $swap\ newPivotRow\ with\ row\ i \rightarrow memI$ ▷ ask memI only swap the index of two rows

7: $normalization \rightarrow memI$ ▷ send command to memI to push the pivot row to normalization chain

8: $elimination \rightarrow mem\ 0, 1, 2, 3, 4, 5, 6, 7$ ▷ send command to all the memories to push data into the elimination blocks in parallel

9: **end for**

10: $output\ results \rightarrow mem\ 0, 1, 2, 3, 4, 5, 6, 7$ ▷ ask all memories to output the results

4.7 Computational Array

As discussed in Chapter 2, much computation is spent on updating all the elements in an augmented matrix in each iteration. The calculation in each iteration can be split into two-phase such as the Normalization phase and Elimination phase. The Algorithm 12 shows the behaviour of the computation of a single iteration in a sequential version, and Figure 4.9 helps to visualize the computation process. The proposed computational array is explicitly designed to accelerate the computation process by distributing the work and computing in parallel. There is no data dependency in each iteration while doing normalization and elimination. The proposed implementation allocates the pivot row column-wise and distributes the workload both column-wise and row-wise during the elimination phase. The computational array consists of a Normalization chain and an Elimination array. The kernels used in the computational array are designed for modularity. So, the array can be scaled to any dimension as long as the normalization chain's length equals the array's length.

Algorithm 12 Pseudo code for computational array.

for $j = pivot \rightarrow 2 \times rank$ **do**

$M[pivot][j] = M[pivot][j]/M[pivot][pivot]$

end for

for $i = 0 \rightarrow rank$ **do**

if $i != pivot$ **then**

for $j = pivot \rightarrow 2 \times rank$ **do**

$M[i][j] = M[i][j] - M[i][pivot] \times M[pivot][j]$

end for

end if

end for

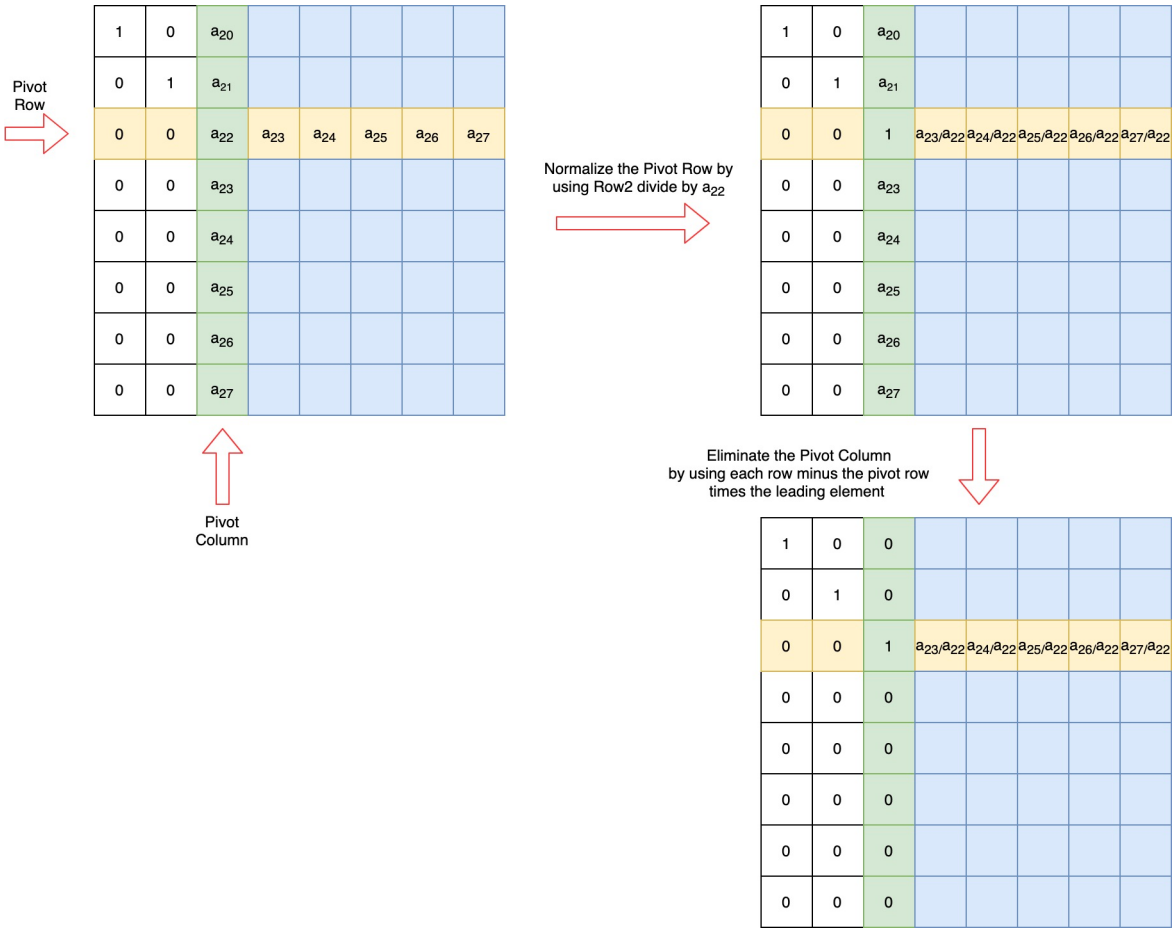

Figure 4.9: The figure shows the computation process in the third iteration. a_{22} is the pivot element, row 2 is called pivot row, and column 2 is called pivot column. The normalization phase is to normalize the pivot row by scaling the pivot element to 1. The Elimination phase uses all other rows to subtract the pivot row to eliminate the pivot column to 0.

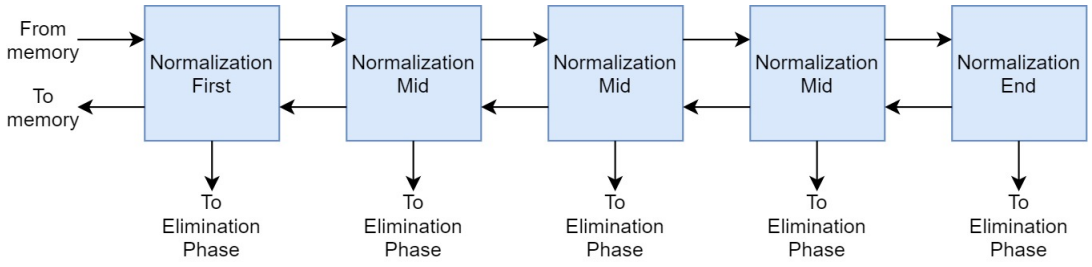

Figure 4.10: Block diagram of Normalization Chain.

4.7.1 Normalization Chain

There are three different kernels used in the Normalization chain, which are Normalization_First, Normalization_Mid and Normalization_End. All Kernels are designed for modularity. So, the length of the chain can be scaled to any dimension depending on the data size. Since the data is distributed to all the cores in the chain evenly, the chain's length should not be greater than the rank of the input matrix to avoid waste. The Figure 4.10 shows a sample implementation of the chain with size 1×5.

The first core of the chain uses the kernel Normalization_First which is shown in Algorithm 13. It gets some configuration includes the length of pivot row saved as $RowLength$ and the reciprocal of pivot element saved as $Factor$. During the Normalization phase, instead of dividing each element in the pivot row by the pivot value, multiplying each element with the pivot element's reciprocal can increase the computation efficiency because multiplication takes fewer cycles than division. The user predefines the number of cores in the chain as $NumOfCore$. To distribute the data, $ProcessLength$ is sent to the right core in the chain. The first core in the chain calculates some configuration flags and spread them to the other cores. Therefore the $CoresLeft$ equals the $NumOfCores$. After that, it will pass the pivot row from memory to the right cores, collect processed data from the right, and write back to the memory.

$$CoreLeft \;=\; NumOfCores \tag{4.2}$$

$$ProcessLength \;=\; Cell(\frac{RowLength}{NumOfCores}) \tag{4.3}$$

Algorithm 13 Pseudo code for Normalization_First.

$RowLength, Factor \leftarrow input$

$CoresLeft \leftarrow NumOfCores$

$ProcessLength \leftarrow cell(RowLength \div NumOfCores)$

$TrashCount \leftarrow ProcessLength \times NumOfCores \ - \ RowLength$

$ProcessLength, CoresLeft, Factor \rightarrow OutputRight$

for $j = 0 \rightarrow RowLength$ **do**

$Memory \rightarrow OutputRight$

end for

for $j = 0 \rightarrow TrashCount$ **do**

$0 \rightarrow OutputRight$

end for

$RowLength, TrashCount \rightarrow OutputBottom$

for $j = 0 \rightarrow RowLength$ **do**

$InputRight \rightarrow Memory$

end for

for $j = 0 \rightarrow TrashCount$ **do**

$InputRight \rightarrow Trash$

end for

The Normalization_Mid kernel, shown in Algorithm 14, reads in $ProcessLength$ and $CoresLeft$ from left and calculate $PassingLength$, which indicates the number of data that needs to be passed to the right. Each core gives the data and the configuration flags to the right cores based on the calculated $PassingLength$ and saves the rest of the data based on the $Passinglength$ to the processor's data memory.

$$PassingLength \ = \ ProcessLength \times CoresLeft \tag{4.4}$$

When Data distribution is finished, each processor starts to process the data at the same time. Then, collect the data from right to left. In the end, the calculated data will be sent to Output_Bottom to be used in the Elimination phase.

Algorithm 14 Pseudo code for Normalization_Mid.

$ProcessLength, CoresLeft, Factor \leftarrow InputLeft$

$CoresLeft = CoresLeft \ - \ 1$

$PassingLength = CoresLeft \times ProcessLength$

$ProcLength, CoresLeft, Factor \rightarrow outputRight$

for $j = 0 \rightarrow PassingLength$ **do**

$InputLeft \rightarrow OutputRight$

end for

for $j = 0 \rightarrow ProcessLength$ **do**

$buffer[j] \leftarrow InputLeft$

end for

for $j = 0 \rightarrow ProcessLength$ **do**

$buffer[j] \leftarrow buffer[j] \times factor$

end for

for $j = 0 \rightarrow PassLength$ **do**

$InputRight \rightarrow OutputLeft$

end for

$ProcLength, PassingLength \rightarrow OutputBottom$

for $j = 0 \rightarrow ProcessLength$ **do**

$buffer[j] \rightarrow OutputLeft$

$buffer[j] \rightarrow OutputBottom$

end for

The Normalization_End kernel, shown in Algorithm 15, is very similar to the Normalization_Mid kernel. The only difference is that it only receives data based on $ProcessLength$ from the left core and send this amount of data back to the left after processing them. Since the core is at the end of the chain, and there is no core on the right, it only needs to send the computed data back to the left and downward to the Elimination array.

Algorithm 15 Pseudo code for Normalization_End.

$ProcessLength, CoresLeft, Factor \leftarrow InputLeft$

for $j = 0 \rightarrow ProcessLength$ **do**

$buffer[j] \leftarrow InputLeft$

end for

for $j = 0 \rightarrow ProcessLength$ **do**

$buffer[j] \leftarrow buffer[j] \times factor$

end for

$ProcLength, PassingLength \rightarrow OutputBottom$

for $j = 0 \rightarrow ProcessLength$ **do**

$buffer[j] \rightarrow OutputLeft$

$buffer[j] \rightarrow OutputBottom$

end for

4.7.2 Elimination Array

The elimination phase is the most time consuming and computation consuming part of the overall inversion algorithm. The Algorithm 16 shows the purpose of the elimination phase, which is a part of the code in Algorithm 1 elimination array is used to accelerate the computation in row elimination. Since there is no data dependency, neither row-wise nor column-wise, unrolling the loop and distributing the work to different cores is the best way to increase throughput. Elimination array consists of an array of processors with dimension $m \times n$. Since the implementation is designed for modularity, the array can be scaled to any dimension easily. The parameter m is used to unroll the loop i, which means m rows can be computed simultaneously. The parameter n is used to unroll the loop j, which indicates that a row of data is distributed to j cores and computed simultaneously. Ideally, if there is no communication delay between each core, the overall accelerate ratio is $m \times n$. The Figure 4.11 shows a simplified elimination array which dimension is 4×5.

Algorithm 16 Row Elimination sequential version.

```
1: for i = 0 → rank do
2:   for j = pivot → 2 * rank do
3:     if i! = pivot then
4:       M[i][j] ← M[i][j] − M[i][pivot] × M[pivot][j]
5:     end if
6:   end for
7: end for
```

The first column of cores shown in Figure 4.11 as blue is used to calculate some configuration flag and spread to other cores with data. The kernel Elimination_First loaded to these cores is shown in Algorithm 17.

Since the length of the elimination array is the same as the normalization chain's length, some configuration flags used in the normalization chain can be reused. The Elimination_Mid kernel, shown in Algorithm 18, reads $ProcessLength$, $PassingLength$ and normalized pivot row data from Input_Top first and spreads downward. When a row of data comes in, the processors spread data to

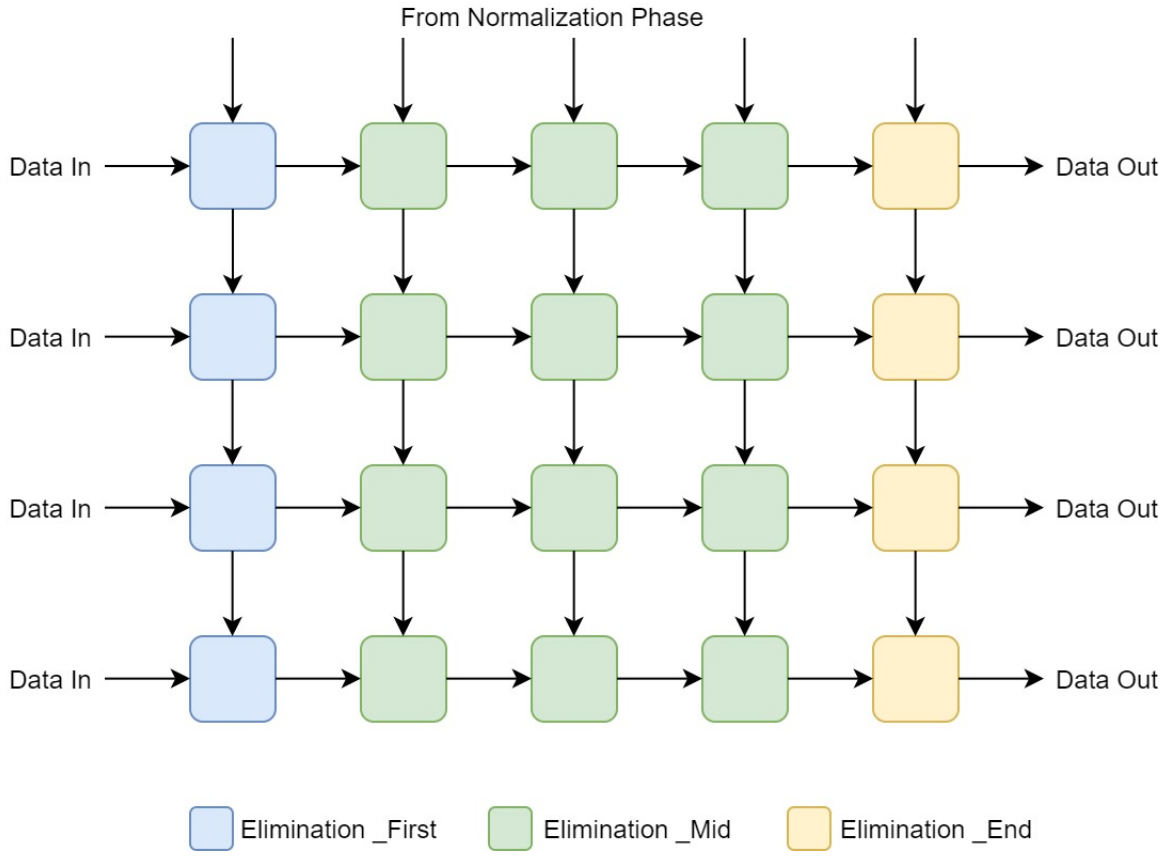

Figure 4.11: Example of a simplified elimination array with size 4×5. The blue processors only spread some configuration flags. The green and yellow cores process data simultaneously. Therefore, ideally, the accelerate ratio is 16. Blue processors are loaded with Elimination_First kernel, processors in green colour are loaded with Elimination_Mid kernel and processors in yellow colour are loaded with Elimination_End kernel.

the right based on *PassingLength* and store the rest of the data locally for computation. When all locally stored data has been processed, each processor collects data from the left and sends the locally processed data to the right. A modified version of the Elimination_Mid kernel is loaded to the last row of the elimination array. The only difference is that the the kernel loaded to the last row does not spread data downward.

The Elimination_End Kernel is loaded in the last column of the elimination array shown in Algorithm 19. Like the Elimination_Mid, it reads the normalized pivot row data from InputTop and spreads the information downward. Since the kernel is loaded to the last column of the array, it collects all processed data from the left and writes it back to the memory to update the matrix. Also, if the current row is below the pivot, it needs to send the element in the next column and RowIndex to the sort processor to find the next pivot row.

Algorithm 17 Pseudo code for Elimination_First.

1: $DataCount, TrashCount \leftarrow InputTop$
2: $DataCount, TrashCount \rightarrow OutputBottom$
3: $NumOfRows \leftarrow InputLeft$
4: **for** $i = 0 \rightarrow NumOfRows$ **do**
5: $RowIndex, Factor \leftarrow InputLeft$
6: **for** $j = 0 \rightarrow DataCount$ **do**
7: $InputLeft \rightarrow OutputRight$
8: **end for**
9: **for** $j = 0 \rightarrow TrashCount$ **do**
10: $0 \rightarrow OutputRight$
11: **end for**
12: **end for**

Algorithm 18 Pseudo code for Elimination_Mid.

1: $ProcessLength, PassingLength \leftarrow InputTop$
2: $ProcessLength, PassingLength \rightarrow OutputBottom$
3: **for** $i = 0 \rightarrow ProcessLength$ **do**
4: $\quad refBuffer[i] \leftarrow InputTop$
5: $\quad refBuffer[i] \leftarrow OutputBottom$
6: **end for**
7: $NumOfRows \leftarrow InputLeft$
8: **for** $i = 0 \rightarrow NumOfRows$ **do**
9: $\quad RowIndex, Factor, DataCount, TrashCount \leftarrow InputLeft$
10: $\quad RowIndex, Factor, DataCount, TrashCount \rightarrow OutputRight$
11: $\quad CollectLength \leftarrow DataCount + TrashCount - PassingLength - ProcessLength$
12: $\quad$ **for** $j = 0 \rightarrow PassingLength$ **do**
13: $\quad\quad InputLeft \rightarrow OutputRight$
14: $\quad$ **end for**
15: $\quad$ **for** $j = 0 \rightarrow ProcessLength$ **do**
16: $\quad\quad rowBuffer[j] \leftarrow InputLeft$
17: $\quad$ **end for**
18: $\quad$ **for** $j = 0 \rightarrow ProcessLength$ **do**
19: $\quad\quad rowBuffer[j] \ - \ factor \times refBuffer[j] \rightarrow OutputRight$
20: $\quad$ **end for**
21: $\quad$ **for** $j = 0 \rightarrow CollectLength$ **do**
22: $\quad\quad InputLeft \rightarrow OutputRight$
23: $\quad$ **end for**
24: **end for**

Algorithm 19 Pseudo code for Elimination_End.

$ProcessLength \leftarrow InputTop$
$ProcessLength \rightarrow OutputBottom$
for $i = 0 \rightarrow ProcessLength$ **do**
  $refBuffer[i] \leftarrow InputTop$
  $refBuffer[i] \leftarrow OutputBottom$
end for
$NumOfRows, PivotIndex \leftarrow InputLeft$
for $i = 0 \rightarrow NumOfRows$ **do**
  $RowIndex, Factor, DataCount, TrashCount \leftarrow InputLeft$
  $CollectLength \leftarrow DataCount - ProcessLength$
  for $j = 0 \rightarrow ProcessLength$ **do**
    $rowBuffer[j] \leftarrow InputLeft$
  end for
  for $j = 0 \rightarrow ProcessLength$ **do**
    $rowBuffer[j] \leftarrow rowBuffer[j] \ - \ factor \times refBuffer[j] \rightarrow Memory$
  end for
  for $j = 0 \rightarrow CollectLength$ **do**
    $InputLeft \rightarrow Memory$
  end for
  for $j = 0 \rightarrow CollectLength$ **do**
    $InputLeft \rightarrow Trash$
  end for
  if $RowIndex > PivotIndex$ **then**
    $RowIndex, rowBuffer[1] \rightarrow Sort\_Processor$
  end if
end for

Chapter 5

Number Scaling in Matrix Inversion

Data scaling is very important for fixed-point processing. This chapter discusses how numbers are scaled during the input quantization phase and calculation.

5.1 Input Quantization and Scaling

5.1.1 Quantization

Quantization is the process by which a high precision number is converted to a lower-precision shorter-word number. It is necessary for our application, so data can be efficiently processed in binary fixed point words that do not waste computational and memory resources. A value of a fixed point data type is just an integer that is scaled by an implicit factor defined by users. For example, 3.14 is represented as 3140 in a fixed point data type with a scaling factor of 1/1000 or represented as 314 with a scaling factor of 1/100. For computational efficiency, the scaling factor in the computer is set to the power of 2.

$$Fixed\_value = round\_to\_int(decimal\_value \times 2^{FWL}) \tag{5.1}$$

$$decimal\_value = fixed\_value \times 2^{-FWL} \tag{5.2}$$

The Eq. 5.1 and Eq. 5.2 shows how to convert a decimal number to a binary fixed number and how to convert a fixed number back to a decimal number. FWL indicates the fraction word length. This thesis uses 8.8 for 16-bit fixed point and 16.16 for 32-bit fixed point. As discussed in Chapter 2, 8.8 means for a 16-bit two's complement, 8 bits represent the fraction parts. Similarly, for a 32-bit

2's complement, 16 bits are used to represent fraction parts. Therefore, the fraction word length (FWL) for 16-bit is eight and for 32-bit is 16.

5.1.2 Input Scaling

The fixed point does not have a large dynamic range as a floating point. Therefore, the input needs to be scaled before the calculation starts to avoid overflow and increase the accuracy. For the input matrix A, $AA^{-1} = I$. If A is scaled by a factor of K, the inverse A^{-1} should be scaled by a factor of $\frac{1}{K}$ to hold the equality shown in Eq. 5.3. X is the scaled input that will be pushed to the program, and X^{-1} is the calculated result. To get the exact result A^{-1}, the calculated results X^{-1}need to be multiplied by the scaling factor K shown in Eq. 5.6.

$$KA * \frac{1}{K}A^{-1} = I \tag{5.3}$$

$$X = KA \tag{5.4}$$

$$X^{-1} = \frac{1}{K}A^{-1} \tag{5.5}$$

$$A^{-1} = KX^{-1} \tag{5.6}$$

5.2 Scaling During Calculation

Fixed point multiplication is the same as 2's complement integer multiplication but requires the implicit decimal point's position to be determined after the multiplication to interpret the correct result.

$$Product = (Multiplicand \times Multiplier) >> FWL \tag{5.7}$$

For example, if two 8.8 fixed point numbers multiply together, the whole product is 32-bit, in which 16 bits are integer part, and 16 bits are fraction part. To make the output still in 8.8 format, this complete product needs to be scaled by rounding the last 8 bits and truncate them. Then, check if the remaining 24-bit overflow. If not, throw the first 8 bits and save the last 16 bits as output. If it overflows, the program will raise an exception and print out the error message.

Fixed point division is a bit more complicated than fixed point multiplication and usually takes much more cycles than performing a multiplication. The method is similar to integer division. If both dividend and divisor have the same fraction word length (FWL), the dividend needs to be

shift left by FWL bits to make the result has the same fraction word length. The equation is shown below.

$$Quotient = (dividend << FWL) \div divisor \tag{5.8}$$

For the division, the overflow exception checking is before the calculation. In the matrix inversion, the division will only be used to calculate reciprocal, which means the dividend is always one. The smallest resolution of a 16-bit fixed point with 8.8 format is 1/256, but the largest absolute value of a 16-bit fixed point is 128. Therefore, if the divisor is 0 or 1/256, the division will raise an exception.

During the calculation, if there is an overflow, the program will raise the exception by printing out the error message and stop. To deal with the exception, users need to either scale down the input or change the data type. Here is what I believe should be done if anyone wants to use the proposed Matrix Inversion code for a real application that needs to do matrix inversion. First, scale the input to the appropriate range. Users can scale down the input if there is an overflow exception and if the quantization loss is not too much. If an overflow exception still occurs, users can try more complex data types. For example, if they currently use a 16-bit fixed point, they can switch to use 32-bit fixed point. If currently using a 32-bit fixed point, users can switch to use 32-bit single-precision float point.

5.3 Experiment and Results

5.3.1 16-bit Fixed Point

In this section, five different data sets with single precision will be applied to test the 16-bit fixed point accuracy. Each data set contains ten randomly generated matrices with the same matrix size and data range. The condition number of all randomly generated matrices are limited to 100. Matrices are generated in Matlab, shown below. The input matrix A is generated with size $N \times N$ and range $[-range, range]$ through Eq.5.9. The condition number is controlled by using Eq.5.10 and Eq.5.11. Apply the SVD decomposition of matrix A, and s is a diagonal matrix vector. Since condition number is defined by the largest diagonal divide by the smallest diagonal, Eq. 5.11 is used to linearly stretch the diagonal to ensure the condition number (c). After modifying the diagonal matrix, multiply the U, s and V together to generate the matrix A using Eq. 5.12. By doing this,

both the range and condition number of the randomly generated matrices can be controlled.

$$A = 2 \times range \times rand(N, N) - range \tag{5.9}$$

$$[U, s, V] = svd(A) \tag{5.10}$$

$$s = s(1) \times (1 - \frac{c-1}{c} \times \frac{(s(1) - s)}{s(1) - s(end)}) \tag{5.11}$$

$$A = U \times s \times V \tag{5.12}$$

The matrices used to test the 16-bit fixed point accuracy is shown in Table 5.1. These matrices are converted to fixed point and scaled to different ranges during the calculation to discover the best scaling factor for a 16-bit fixed point with 8.8 two's complement. The accuracy of the calculated

Table 5.1: Data set information for 16-bit fixed point

Data set number	**Matrix size**	**Data range**
0	100x100	[-1,1]
1	100x100	[-2,2]
2	100x100	[-4,4]
3	100x100	[-8,8]
4	100x100	[-16,16]

results is evaluated by using relative error shown in Eq. 5.13. $inv(A)$ is the inversed matrix calculated by using Matlab's built-in function $inv()$. This thesis measures every matrix's relative error and calculates the average of the relative errors for each data set. The results are shown in Figure 5.1. According to the figure, use the blue line as an example. The original input range is [-16,16] and the lowest relative error occurs when the scaling factor is 1/8, which means the input is scaled to [-2,2] ($16 \times 1/8 = 2$) before the calculation. As shown in Figure 5.1, the calculated results have the best accuracy when input is scaled to [-2,2], and there is no overflow when the input is scaled to this range.

$$relative\ error = \frac{||A^{-1} - inv(A)||}{||inv(A)||} \times 100\% \tag{5.13}$$

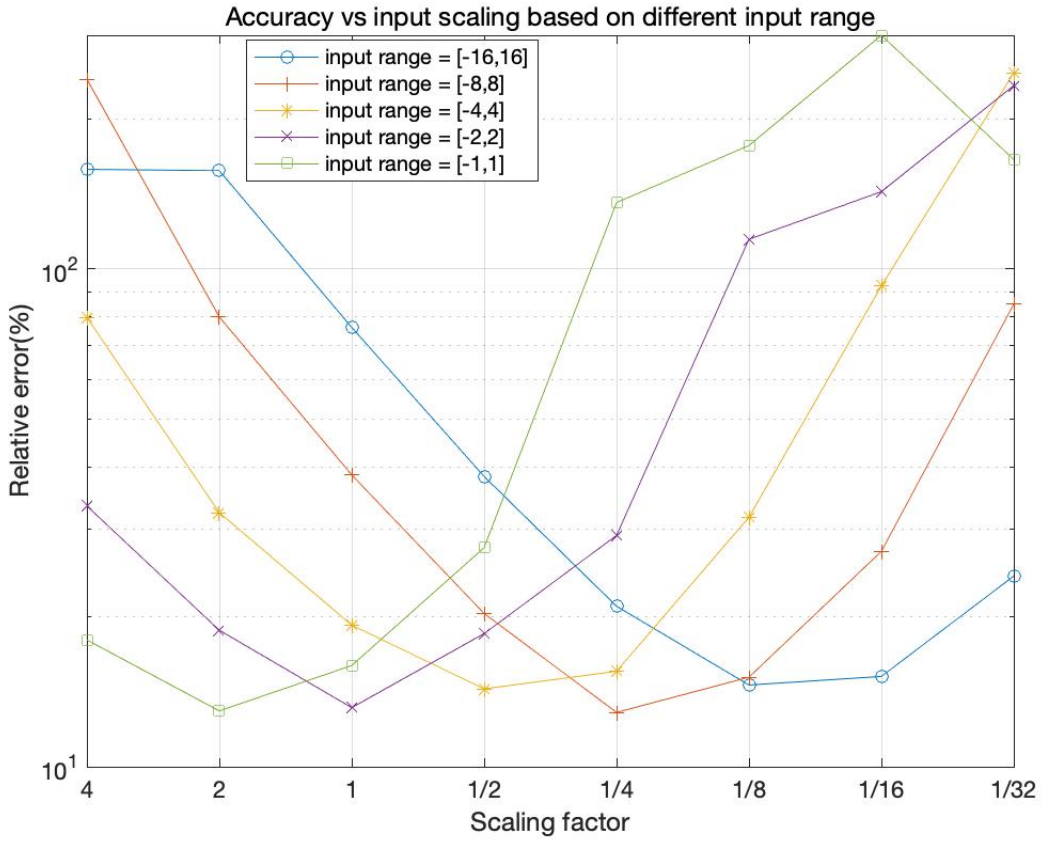

Figure 5.1: Matrices with different data ranges for 16-bit fixed version. Before the calculation, the input is scaled to different ranges. If the scaling factor equals one, the data is not scaled. If the scaling factor equals 1/2, the data is scaled to half of its original value before the calculation starts. The figure shows the relative error with the different scaling factors being used. Lower is better. The legend shows the original data range of each data set. The x-axis shows the scaling factor. The product of these two numbers is the scaled data range before the calculation.

5.3.2 32-bit Fixed Point

This thesis uses 16.16 two's complement for the 32-bit fixed point, which has a larger input range and has a smaller resolution than the 16-bit fixed point. Therefore, another 4 data sets shown in Table 5.2 are used to find the appropriate input scaling.

The results are shown in Figure 5.2. The original data range is shown in the legend. For example, the blue line in the figure indicates the original data range is from [-2048,2048] and when the scaling factor is 1/1024 the relative error reachs its minimum, which means the data is scaled to [-2,2] ($1/1024 \times 2048 = 2$). According to the Figure 5.2, for most of cases, input should be scaled to [-2,2] to reach the best accuracy. Similar to the reciprocal calculation, if the input matrix is ten

Data set number	Matrix size	Data range
0	100x100	[-2048,2048]
1	100x100	[-4096,4096]
2	100x100	[-8192,8192]
3	100x100	[-16384,16384]

Table 5.2: Data set information for 32-bit fixed point

times larger, the inverse is ten times smaller. The calculated output is too small for some large input to be represented by 16-bit fraction word length without scaling. So by scaling the input to a small range, the relative error is decreased. However, if the input is scaled too small, the input loss will be dominant. Therefore, based on the experiment, the data should be scaled to [-2,2] to reach the best accuracy for most of the cases.

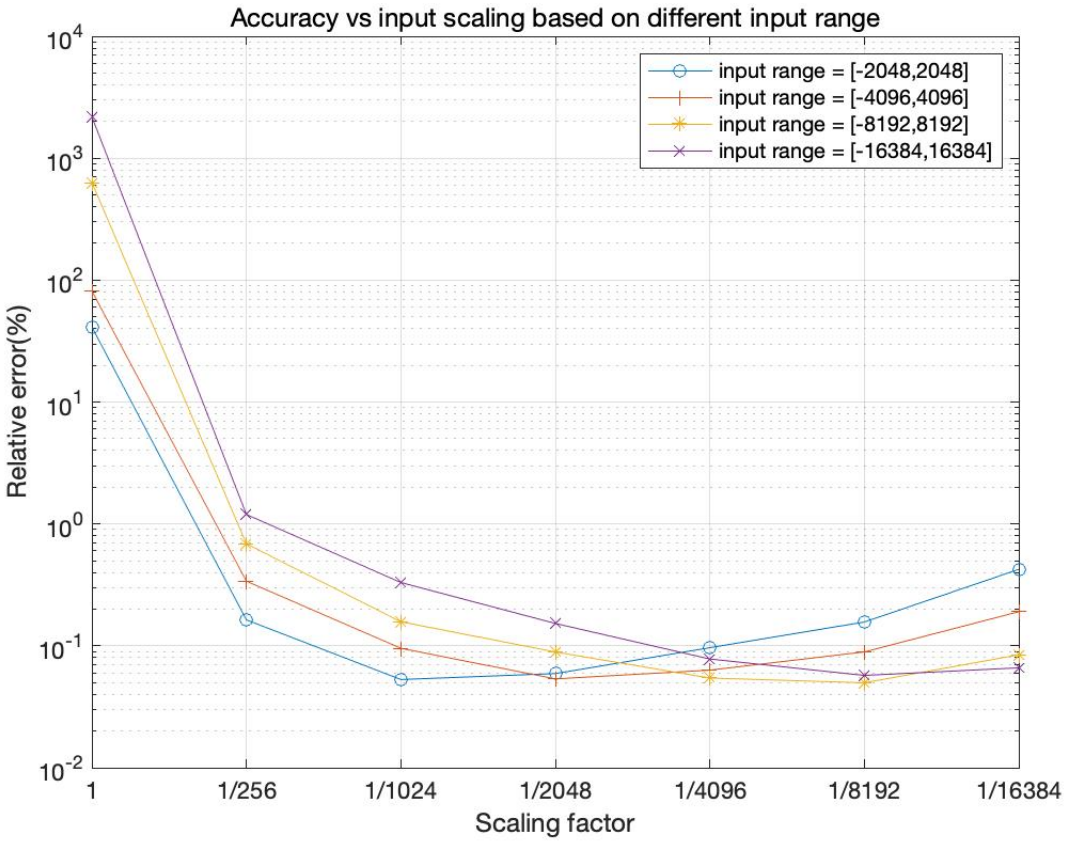

Figure 5.2: Matrices with different data ranges for 32-bit fixed version. Before the calculation, the input is scaled to different ranges. If the scaling factor equals one, the data is not scaled. If the scaling factor equals 1/2, the data is scaled to half of its original value. The figure shows the relative error with the different scaling factors being used. Lower is better. The legend shows the original data range of each data set. The x-axis shows the scaling factor. The product of these two numbers is the scaled data range before the calculation.

Chapter 6

Comparison of Matrix Inversion Methods on AsAP3

As mentioned in Chapter 1, the energy-efficient matrix inversion on multi- and many-core platforms have been a keen research interest. This interest, coupled with the ever-increasing number of processor cores per general processing chip, has led me to focus on using many-core processor arrays for matrix inversion.

Throughput data for the implementations on the many-core platform (AsAP3) are obtained with a cycle-accurate C++ simulator. Power measurements from the 32 nm PD-SOI CMOS fabricated chip are input to the simulator to obtain power data. The precision results are obtained by converting the simulator's output from 16-bit and 32-bit fixed point back to the floating point and compared with the golden reference. The reference is generated using the Matlab built-in $inv()$ function. Two important features are tested and calculated for all matrix inversion implementations: throughput per watt and output accuracy;

Throughput per watt is useful for selecting the most energy-efficient matrix inversion option, which is the bottleneck for implementing scientific applications on many-core or multi-core platforms. Since the area is the same for all the implementations on the many-core platform (AsAP3), throughput per area is not included in this comparison. Throughput per area is presented in the next chapter to compare against the designs on other platforms.

$$Throughput = \frac{Number\ Of\ Inverted\ matrices}{Execution\ Time} \tag{6.1}$$

For each implementation, the throughput is calculated by dividing the total number of inverted

matrices by the execution time. Therefore the unit is matrix inversion per second (MatInv/sec).

The accuracy is useful for selecting the appropriate data type used in different scenarios in real-life problems. The long data type, such as double or single float points, has a higher accuracy but slow computation. The evaluation of accuracy helps find a balance point between the performance and the accuracy, shown in section 6.3.

6.1 Performance Comparison between Implementations on AsAP3

For Power efficient comparison between different implementations on AsAP3, five metrics with different dimensions are used, and the results are shown in Table 6.1. The proposed 16-bit fixed point uses only on-chip memory, which has the best throughput per energy performance. However, due to the limitation of on-chip memory size, it only supports matrices smaller than 320×320. The 32-bit fixed point version and 32-bit float version use external memory. They can invert much larger matrices compare to the 16-bit version but slower in computation. As the matrix size increases, the external memory I/O becomes the bottleneck of the overall program. Therefore, the throughput per energy between the 32-bit fixed point and 32-bit float point versions does not have big differences.

Table 6.1: Throughput per watt for different many-core implementations with various input matrices. The unit is MatInv/sec/W, which is matrix inversion per second per W.

matrix size	**InversionBigMem** 16-bit fixed point	**InversionExternalMem** 32-bit fixed point	**InversionExternalMem** 32-bit float point
100×100	263.16	73.86	65.19
200×200	32.11	10.50	9.41
300×300	9.93	3.25	2.99
400×400	—	1.40	1.28
500×500	—	0.71	0.65

6.2 Error Analysis and Precision

There are two types of error introduced in matrix inversion: quantization error and round off error. Quantization error is introduced when a decimal float number is converted to a binary fixed number. Round off error is introduced when computer rounds off in each step during the calculation of inverse. This section discusses how far the computer answer is from the real answer.

6.2.1 Condition Number

A condition number of a problem measures the solution's sensitivity to small perturbations in the input data [16]. In the matrix inversion problem, the condition number indicates how accurate the computed result is. A matrix A is ill-conditioned if relatively small changes in the input (matrix A) can cause a large change in the output (A^{-1}), which means a small roundoff error can have a drastic effect on the result. However, if the matrix is well-conditioned, then the computed solution is quite accurate. Thus, the accuracy of the solution depends on the condition number of the matrix. An example shown in (6.2) and (6.3) indicates the accuracy is affected by a large condition number. The condition number of matrix A equals 1,623 which is calculated by using built-in function $cond()$ in Matlab. Matrix B is modeled as Matrix A plus a little error. The error can be introduced during the quantization stage when the input data is represented by a fixed-point. As shown in (6.3), $invA$ is the exact value for A^{-1} calculated by using $inv()$ in Matlab and $invB$ is also calculated in Matlab. Since the condition number of A is large, the output changed dramatically (the difference between $invA$ and $invB$) even if a small roundoff error is introduced in the input (the difference between A and B).

$$A = \begin{bmatrix} 4.1 & 2.8 \\ 9.7 & 6.6 \end{bmatrix}, B = \begin{bmatrix} 4.1 & 2.8 \\ 9.671 & 6.608 \end{bmatrix} \tag{6.2}$$

$$invA = \begin{bmatrix} -66 & 28 \\ 97 & 41 \end{bmatrix}, invB = \begin{bmatrix} 472 & -200 \\ -690.7857 & 292.8571 \end{bmatrix} \tag{6.3}$$

The condition number of a matrix is calculated by using Eq. 6.4 where $||\ ||$ is the norm of matrix.

$$cond(A) = ||A|| \times ||A^{-1}|| \tag{6.4}$$

If matrix A is nonsingular, SVD decomposition can be used to calculate condition number as shown below where σ_{max} is the largest value in the diagonal matrix and σ_{min} is the minimum value in the diagonal matrix.

$$cond(A) = \frac{\sigma_{max}}{\sigma_{min}} \tag{6.5}$$

6.2.2 Residual and Accuracy

Let x be the solution of $AX = I$, where A is the input matrix, I is the identity matrix, and X is the calculated version of A^{-1}. Generally, it is hard to verify the error because we cannot get exact result of A^{-1}. So, this thesis uses a complete method which includes left residual (6.6), right residual (6.7) and direct error (6.8) to evaluate the accuracy which is suggested by [17] and [18]. $inv(A)$ is calculated by using Matlab's built-in function $inv()$.

$$left\ residual = AX - I \tag{6.6}$$

$$right\ residual = XA - I \tag{6.7}$$

$$direct\ error = X - inv(A) \tag{6.8}$$

To evaluate the precision, all three methods are used to compared with tolerance for every element in a matrix. If both left and right residual and direct errors are smaller than a tolerance, I will reduce the tolerance until the smallest tolerance.

6.3 Accuracy Comparison between Different Data Types

For Accuracy comparison between different data types, a total of 16 different data sets, whose condition numbers are from 50 to 300 and matrix size are from 50 to 300, is used to evaluate the accuracy. Each data set contains ten different test cases, which are generated randomly by using the equations from Eq. 5.9 to Eq. 5.12. Therefore, 160 matrices in total are used. All test cases are scaled properly for different data types to ensure there is no overflow during the calculation. Left residual (6.6), right residual (6.7)and direct error (6.8) are measured for each test case and compared with tolerance. Tolerance is denoted as $e_k = 2^{-k}$, where $k = 1, 2, ..., 8, 9$. If all of the left residual, right residual, and absolute error for every element in the matrix are below the tolerance, this test case is considered successful. The results are shown in Table 6.2, Table 6.3 and Table 6.4. As the size and condition number increasing, the error also increases. 16-bit fixed point has an

acceptable error only when the matrix size and condition number are small. The 32-bit fixed point can maintain a low error, less than 2^{-7}, for all test cases. The single-precision 32-bit floating point has the best accuracy, which is lower than 2^{-9} for all test cases.

Matrix Size	50×50				100×100				200×200			
Condition #	50	100	200	300	50	100	200	300	50	100	200	300
e_1	10	10	5	2	10	10	1	0	0	0	0	0
e_2	10	4	0	0	0	0	0	0	0	0	0	0
e_3	0	0	0	0	0	0	0	0	0	0	0	0

Table 6.2: The accuracy of using the 16-bit fixed point implementation. Each unique matrix size and condition number contains ten different randomly generated matrices. Ten means all test cases are passed. A test case is considered as passed when both direct error and residual are smaller than the tolerance. Tolerance is denoted as $e_k = 2^{-k}$, where $k = 1, 2, ..., 8, 9$. For example, e_1 means the largest error is not above 2^{-1} and e_3 means the largest error is not above 2^{-3}. 0.5 is a large tolerance but most test cases are failed especially when the condition number and matrix sizes increase. Therefore, it is not usable practically.

The Figure 6.1, Figure 6.2 and Figure 6.3 show the detailed right residual error information based on a worst-case which is a 300×300 matrix with 300 condition number for 16-bit fixed point (i16), 32-bit fixed point (i32), and 32-bit float point (f32). The method use 32-bit float point has perfect accuracy. For the 32-bit fixed point, the residual is still very low. However, due to data size limitation, the 16-bit fixed point cannot keep a good accuracy because the roundoff error and quantization error have a big effect on the calculated results when matrix size and condition number increase.

6.4 Summary

$InversionBigMem\_i16$ uses the 16-bit fixed point has the highest throughput per energy. However, it cannot keep a good accuracy due to the data size limitations. So, the 16-bit fixed point with 8.8 format is not usable practically without some additional hardware.

$InversionExternalMem\_f32$ uses the 32-bit float point that has the best accuracy among three data types. Also, with the usage of external memory, the implementation can take a much larger sized matrix. This implementation is used in the next chapter to be compared with the implementations on other platforms.

Matrix Size	100×100				200×200				300×300			
Condition #	50	100	200	300	50	100	200	300	50	100	200	300
e_1	10	10	10	10	10	10	10	10	10	10	10	10
e_2	10	10	10	10	10	10	10	10	10	10	10	10
e_3	10	10	10	10	10	10	10	10	10	10	10	10
e_4	10	10	10	10	10	10	10	10	10	10	10	10
e_5	10	10	10	10	10	10	10	10	10	10	10	10
e_6	10	10	10	10	10	10	10	10	10	10	10	10
e_7	10	10	10	9	10	10	10	8	9	10	5	1
e_8	10	9	2	1	0	0	0	0	0	0	0	0
e_9	1	0	0	0	0	0	0	0	0	0	0	0

Table 6.3: The accuracy of using the 32-bit fixed point implementation. Each unique matrix size and condition number contains ten different randomly generated matrices. Ten means all test cases are passed. A test case is considered as passed when both direct error and residual are smaller than the tolerance. Tolerance is denoted as $e_k = 2^{-k}$, where $k = 1, 2, ..., 8, 9$. For example, e_1 means the largest error is not above 2^{-1} and e_9 means the largest error is not above 2^{-9}. For all test cases the max error is not greater than 2^{-6}. It can be used if a real application does not need very high precision.

Matrix Size	100×100				200×200				300×300			
Condition #	50	100	200	300	50	100	200	300	50	100	200	300
e_1	10	10	10	10	10	10	10	10	10	10	10	10
e_2	10	10	10	10	10	10	10	10	10	10	10	10
e_3	10	10	10	10	10	10	10	10	10	10	10	10
e_4	10	10	10	10	10	10	10	10	10	10	10	10
e_5	10	10	10	10	10	10	10	10	10	10	10	10
e_6	10	10	10	10	10	10	10	10	10	10	10	10
e_7	10	10	10	10	10	10	10	10	10	10	10	10
e_8	10	10	10	10	10	10	10	10	10	10	10	10
e_9	10	10	10	10	10	10	10	10	10	10	10	10

Table 6.4: The accuracy of using the 32-bit float point implementation. Each unique matrix size and condition number contains ten different randomly generated matrices. Ten means all test cases are passed. A test case is considered as passed when both dierect error and residual are smaller than the tolerance. Tolerance is denoted as $e_k = 2^{-k}$, where $k = 1, 2, ..., 8, 9$. For example, e_1 means the largest error is not above 2^{-1} and e_9 means the largest error is not above 2^{-9}. For all test cases the max error is not greater than 2^{-9}, which is the most accurate one among all three data types.

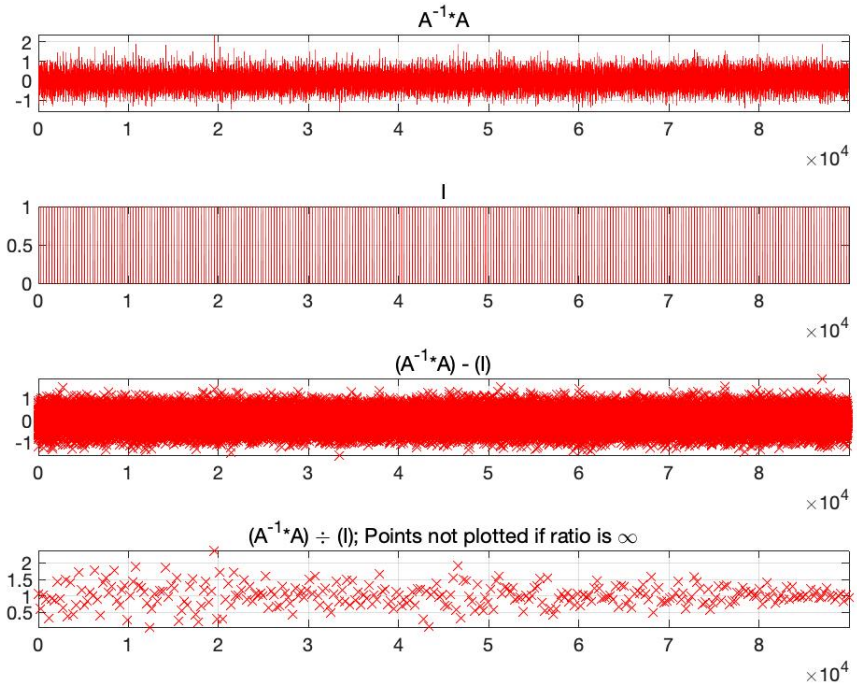

Figure 6.1: The residual of a 300×300 matrix with 300 condition number by using 16-bit fixed point with 8.8 format. It has the largest condition number and matrix size, which is considered as the worst-case of all data sets. The first plot does not show any similar pattern to the identity matrix and both difference (the third plot) and ratio (the fourth plot) is very large. Therefore, the 16-bit fixed point is not usage practically for now.

$InversionExternalMem\_i32$ uses the 32-bit fixed point that combines the advantage of float version and 16-bit version. It has acceptable accuracy for the condition number below 300 and has a better throughput per watt than the floating point.

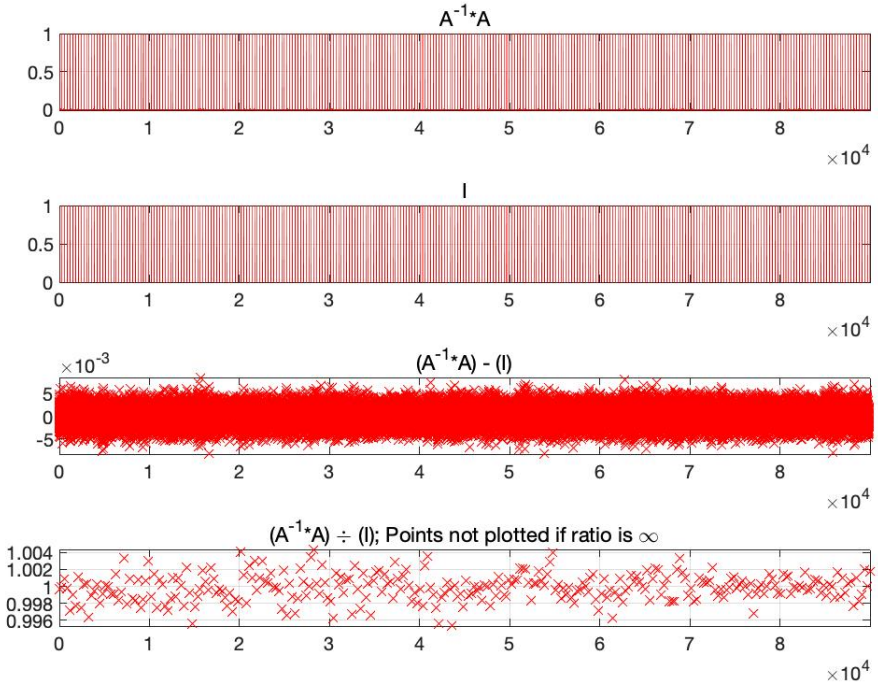

Figure 6.2: The residual of a 300×300 matrix with 300 condition number by using 32-bit fixed point with 16.16 format. It has the largest condition number and matrix size, which is considered as the worst-case of all data sets. The first plot shows the result is very close to the identity matrix and both difference (the third plot) and ratio (the fourth plot) is very small. Therefore, the 32-bit fixed point can keep a good accuracy.

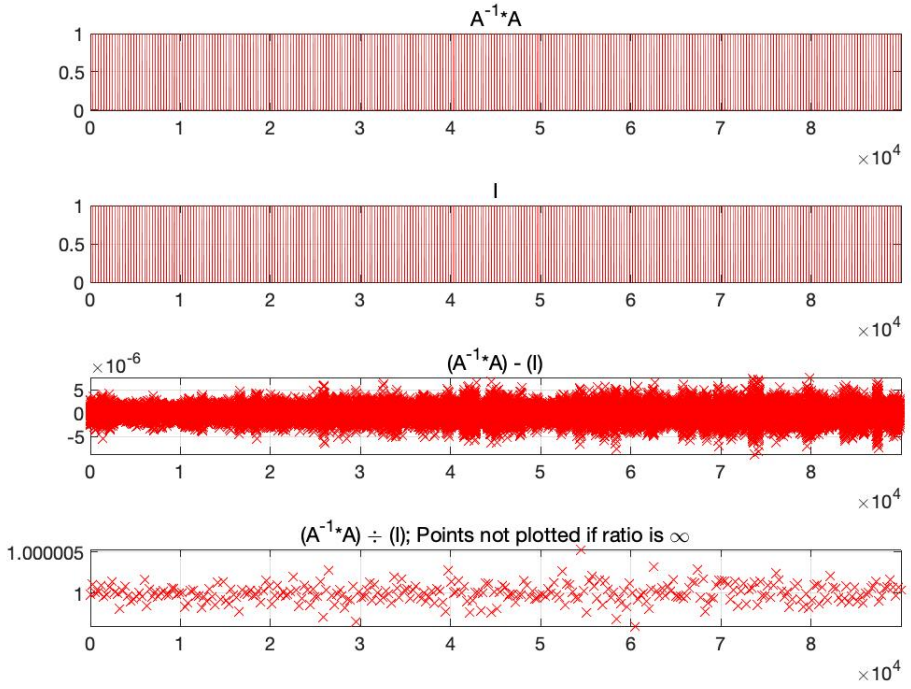

Figure 6.3: The residual of a 300×300 matrix with 300 condition number by using 32-bit float point. It has the largest condition number and matrix size, which is considered as the worst-case of all data sets. The first plot shows the result is very close to the identity matrix and both difference (the third plot) and ratio (the fourth plot) are smaller than 32-bit fixed point. Therefore, the 32-bit float point has a higher accuracy than the 32-bit fixed point.

Chapter 7

Comparison of Matrix Inversion Methods on AsAP3 with General-Purpose Processor and GPU

7.1 Matrix Inversion Data Set

For performance comparison between the implementation on the many-core platform (implementation uses off-chip memory with 32-bit float point data type) and others (general-purpose processor and GPU), five matrices with different dimensions are used (from 100×100 to 500×500). All matrices are generated randomly in single-precision 32-bit IEEE-754 format by using Eq. 7.1 in Matlab. N is the matrix size and each element in the matrix is bounded by $[-range, range]$. Since the accuracy is analyzed in the previous section, the data range in this section is just set to [-1,1] to evaluate the performance.

$$A = 2 \times range \times rand(N, N) - range \tag{7.1}$$

7.2 Matrix Library

To ensure a fair comparison of performance between the method on AsAP3 and other platforms, LAPACK [19] for Windows in C on the general-purpose processor and CUDA cuBLAS [20] on the GPU are used in this thesis. The evaluation precision of all benchmarks is the single-precision

floating-point (32-bit). This thesis compares against the matrix inversion implementations provided by LAPACK open-source library in C and NVIDIA's cuBLAS.

7.3 Measurement and Simulation Methodology

The general-purpose processor used for comparison is the Intel Core i7-9700k, and its specifications are shown in Table 7.1. The GPU used for matrix inversion is the Nvidia GTX 1070, and its specifications are shown in Table 7.1.

Chip	**Technology(nm)**	**TDP(W)**	**Die Area(mm^2)**	**Clock Rate(GHz)**
Intel Core-i7 9700k	14	95	149	4.7
NVIDIA GTX1070-8G	16	150	314	1.5

Table 7.1: Details of general-purpose processor and GPU utilized for matrix inversion comparison.

Throughput data for implementations on the many-core platform is obtained from a cycle-accurate simulator, customized for AsAP3 (KiloCore) [1]. The chip is fabricated by a 32 nm CMOS PD-SOI technology, and this model is the input for the simulator to obtain power data. The chip power measurement is under the working condition at 0.9V and 1.8 GHz.

Throughput is calculated as the reciprocal of the difference of the last output times between two consecutive matrices as shown in Eq. 7.2, where $LastOutputTime_i$ is the last output time of the i^{th} matrix.

$$Throughput = \frac{1}{LastOutputTime_{i+1} - LastOutputTime_i} \tag{7.2}$$

The area is calculated as the sum of the areas of all the processors and on-chip memory modules as in Eq. 7.3, where $nProcs$ and $nMems$ are the number of processors and memory modules in the many-core implementation.

$$Area = nProc \times 0.055 + nMem \times 0.164 \tag{7.3}$$

Throughput per area is calculated as the throughput divided by area and is given by Eq. 7.4

$$ThroughputPerArea = \frac{Throughput}{Area} \tag{7.4}$$

Table 7.2 shows the raw simulation results of matrices with different sizes on the proposed many-core implementation. The many-core implementation has 635 processors and eight on-chip

Matrix Size	Throughput (MatInv/sec)	Energy per Matrix Inversion (mJ/Matrix)	Area (mm^2)
100×100	189.04	15.34	36.24
200×200	27.07	106.25	36.24
300×300	8.20	334.01	36.24
400×400	3.50	779.9	36.24
500×500	1.78	1543.57	36.24

Table 7.2: Multiple matrices simulation results for the many-core implementation. All results are unscaled raw data in 32 nm CMOS.

memory modules. The total area calculated using Eq. 7.3 is 36.24 mm^2. Also, this implementation uses eight off-chip memory modules which have 100 ns latency. When the input sizes increase, the off-chip memory access frequency also increases which lowers the throughput.

The throughput per area (MatInv/sec/mm^2) and matrix inversions per energy (MatInv/J) will be measured and calculated for all matrix inversion implementations on various platforms under different matrix sizes. The general-purpose throughput data is gathered from a built-in implementation in LAPACK, an open-source library for linear algebra. The power consumption is estimated using half of the thermal design power (TDP/2) [21]. The GPU throughput data is collected from the CUDA using the implementation in the cuBlas library, and power consumption is estimated using half of the thermal design power (TDP/2). The energy dissipation on general-purpose and GPU is estimated and calculated by using Eq. 7.5

$$EnergyDissipation = Power \times Execution\ time \tag{7.5}$$

For all implementations, the execution time is measured from the start of reading the input to completing all the calculations. For general-purpose processor-based implementation, this includes the time to read the input matrix but excludes the time to save the output into a file. GPU-based implementations consist of the time to load data from host memory to device memory but do not include the time to write back. For many-core implementations, this includes reading time and the time to output the results to a file.

Due to the different fabrication technologies used, throughput, die area, and energy are scaled to 14 nm values for AsAP3 and GPU implementations. The scaling factors are chosen by

using the characteristics of different technology nodes which are suggested by [22] and [23]. The scaled data is calculated by using the Eq. 7.6. The estimated scaling factor used in this thesis is shown in Table 7.3.

$$ScaledData_{(14nm)} = RawData \times ScalingFactor \tag{7.6}$$

Technology of Raw Data (nm)	Delay	Energy	Area
32	0.410	0.282	0.38
16	0.657	0.805	0.91

Table 7.3: The scaling factor used to convert original raw data to 14nm numbers. For example, the second row indicates the scaling factors used to scale the data from 32 nm to 14 nm.

7.4 Comparison with Other Work

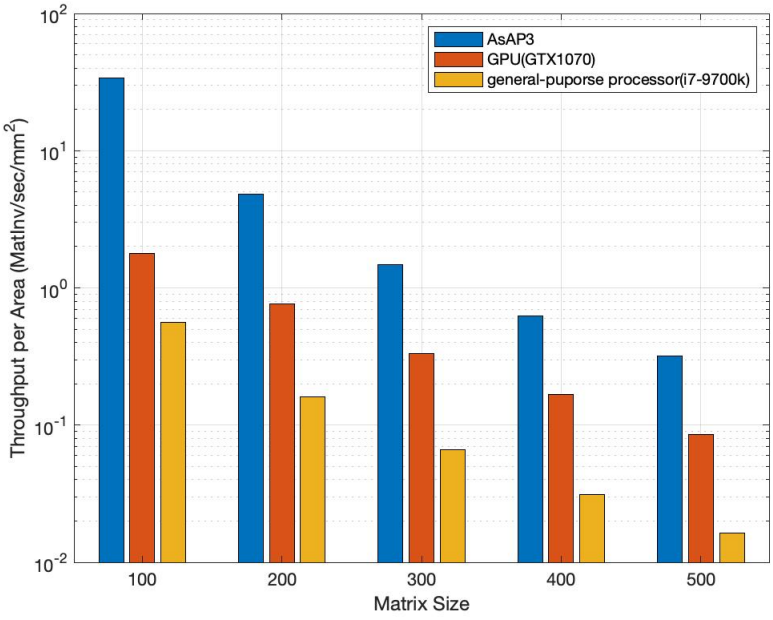

Figure 7.1: Scaled Throughput per Area for different sized matrices on different platforms. Higher is better. Source data is given in Table 7.4.

The proposed many-core matrix inversion implementation is simulated by using different sizes of input matrices. As shown in Table 7.4, the proposed implementation has the highest

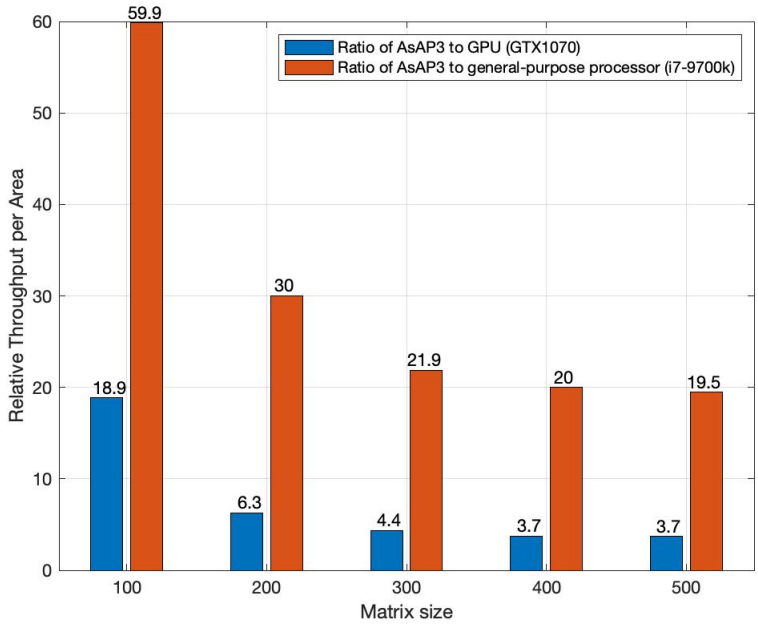

Figure 7.2: Scaled relative Throughput per Area of the implementation on AsAP3 vs the implementations on GPU and general-purpose processor. Source data is given in Table 7.4

throughput per area on all tested matrix sizes. The many-core platform implementation increases throughput per area by a 3.7–19× versus GPU and a 20–60× versus general-purpose processor. The matrix inversions per energy on the proposed implementation is improved by 8.5–41× versus the GPU and 45–131× versus the general-purpose processor, which is shown in Table 7.5.

Since the complexity of inverting a matrix is $O(N^3)$, the throughput decreases dramatically when the input matrix size increases. The many-core implementation uses external off-chip memories whihc has 100 ns latency. When the matrix sizes increase, off-chip memory access frequency also increases in each iteration during the calculation, which is another reason that leads to lower throughput. The Figure 7.1 indicates the throughput per area on different platforms, and the implementation on AsAP3 has a better performance on all input matrix sizes. The throughput per area speed up ratio is shown in Figure 7.2. When the input matrix size becomes larger, the speedup ratio decreases because the on-chip memory is very limited compared to the GPU and general-purpose processors. The frequent data transfer between on-chip memory and external off-chip memory lowers the performance on very large-sized input matrices. However, as the input matrix keeps increasing, GPU and general-purpose processors ran out of the first-level cache. Therefore the speedup ratio remains stable when the input matrix size greater than 300, as shown in Figure 7.2.

The implementation on AsAP3 also offers the highest matrix inversions per energy for all matrix sizes compares to the implementation on the GPU and the general-purpose processor, which is shown in Figure 7.3.

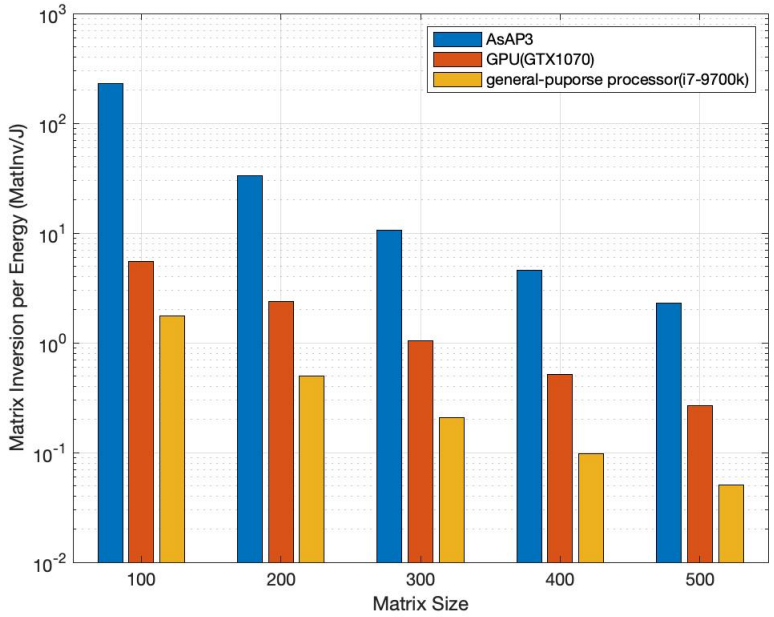

Figure 7.3: Matrix inversions per energy for different sized matrices on different platforms. Higher is better. Source data is given in Table 7.5.

Matrix Size	Platform	Technology (nm)	Scaled Throughput (MatInv/sec)	Area Scaled to 14nm (mm^2)	Scaled Throughput per Area (MatInv/sec/mm^2)	Scaled Relative Throughput per Area
100×100	AsAP3	32	461.06	13.77	33.48	59.86
	GTX1070 CUDA cuBlas	16	507.36	285.74	1.78	3.17
	i7-9700k LAPACK	14	83.33	149	0.56	1
200×200	AsAP3	32	66.03	13.77	4.79	30.00
	GTX1070 CUDA cuBlas	16	217.44	285.74	0.76	4.76
	i7-9700k LAPACK	14	23.81	149	0.16	1
300×300	AsAP3	32	20.01	13.77	1.45	21.87
	GTX1070 CUDA cuBlas	16	95.13	285.74	0.33	5.01
	i7-9700k LAPACK	14	9.90	149	0.07	1
400×400	AsAP3	32	8.55	13.77	0.62	19.99
	GTX1070 CUDA cuBlas	16	47.56	285.74	0.17	5.36
	i7-9700k LAPACK	14	4.63	149	0.03	1
500×500	AsAP3	32	4.35	13.77	0.31	19.45
	GTX1070 CUDA cuBlas	16	24.55	285.74	0.09	5.29
	i7-9700k LAPACK	14	2.42	149	0.02	1

Table 7.4: Comparison of fabrication technology, key throughput and area across 5 matrix sizes for three hardware platforms. Throughput and area data are scaled to 14 nm CMOS. Throughput is measured as matrix inversions per second (MatInv/sec). Raw data is given in Table 7.2.

Matrix Size	Platform	Technology (nm)	Scaled Matrix Inversions per Energy (MatInv/J)	Scaled Relative Matrix Inversions per Energy
100×100	AsAP3	32	231.17	131.77
	GTX1070 CUDA cuBlas	16	5.52	3.15
	i7-9700k LAPACK	14	1.75	1
200×200	AsAP3	32	33.38	66.58
	GTX1070 CUDA cuBlas	16	2.37	4.72
	i7-9700k LAPACK	14	0.50	1
300×300	AsAP3	32	10.62	50.93
	GTX1070 CUDA cuBlas	16	1.04	4.97
	i7-9700k LAPACK	14	0.21	1
400×400	AsAP3	32	4.55	46.65
	GTX1070 CUDA cuBlas	16	0.52	5.31
	i7-9700k LAPACK	14	0.10	1
500×500	AsAP3	32	2.30	45.07
	GTX1070 CUDA cuBlas	16	0.27	5.24
	i7-9700k LAPACK	14	0.05	1

Table 7.5: Matrix inversions per energy for various platforms. To ensure a fair comparison, numbers in column 4 and column 5 are scaled to 14 nm CMOS. The many-core implementation offers the highest matrix inversions per energy on all different matrix sizes. Raw data is given in Table 7.2.

Chapter 8

Thesis Summary and Future Work

8.1 Thesis Summary

This thesis summarizes the current research and challenges in computing the inversion of large matrices. The algorithm's background to invert a matrix and the data type used in the proposed implementations are reviewed. The target platform, a large 2D mesh architecture indicated as AsAP3 (KiloCore), is given.

It continues to demonstrate and explore the matrix implementations on a many-core platform (AsAP3). Two architectures, InversionBigMem and InversionExternalMem, and three different data types are introduced. Also, all unique programs that are loaded to the cores are explained. Accuracy between different data types is evaluated to find the trade-off between performance and accuracy.

To measure against matrix inversion implementations on general-purpose processors and GPU, LAPACK and CUDA are used for the benchmark. The evaluation precision of all benchmarks is single-precision 32-bit IEEE-754 format. Throughput per Area and Matrix Inversions per Energy of all implementations are measured for inverting any random dense matrices with varying sizes. The many-core implementation offers an improvement in Throughput per Area by 3.7–19× versus GPU based design and 20–60× versus the general-purpose processor based design. The many-core implementation also offers the Matrix Inversions per Energy with an improvement of 8.5–41× versus GPU and 45–131× versus general-purpose processor.

8.2 Future Work

The proposed implementation based on a 16-bit version can offer a very high throughput per watt performance compare to the 32-bit version. However, the accuracy of a large matrix is poor by using the 16-bit fixed point. Therefore, a half-precision float point is a good alternative data type used in matrix inversion.

Half-precision or float16 is a relatively new floating-point data type that uses 16-bit, unlike traditional 32-bit single-precision or 64-bit double-precision data types. According to IEEE 754 standard [24], there are 10-bit for the mantissa, 5-bit for the exponent and 1 bit for the sign, which shows in the Figure 8.1.

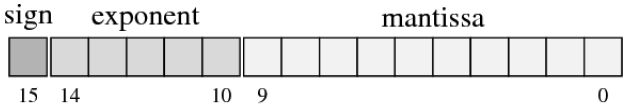

Figure 8.1: IEEE 754 half-precision format

The proposed algorithm has been implemented on Matlab to discover the potential of the 16-bit half-precision float point. The same data sets as mentioned in chapter 6 are used to test how accurate it is. The results are shown in Table 8.1.

Based on the simulation results, the half-precision floating point can keep a relatively high accuracy (less than 2^{-3}) even if the matrix size and condition number are large. Therefore, the next step will be implementing the matrix inversion by using 16-bit half-precision floating point.

Matrix inversion is widely used as an intermediate step of many applications. This thesis analyzes the accuracy of different data types such as 16-bit fixed point, 32-bit fixed-point, 16-bit float point (not implemented) and 32-bit float point. To use this matrix inversion calculator, users need to choose the appropriate data type based on the real applications' accuracy requirement. If there is an overflow during the calculation, the program will print out the error message and stop. Users need to scaled-down the input or change to use other data types manually. For example, switch from 16-bit to 32-bit. Therefore, the next step is to integrate all these data types together and deal with the overflow exception automatically. For example, if the 16-bit fixed point is used and overflow occurs, only use 32-bit for this overflowed element and continue. By doing this, most of the elements are still 16-bit, and only a few elements in the matrix are 32-bit, which can keep

Matrix Size	100×100				200×200				300×300			
Condition #	50	100	200	300	50	100	200	300	50	100	200	300
e_1	10	10	10	10	10	10	10	10	10	10	10	10
e_2	10	10	10	10	10	10	10	10	10	10	10	10
e_3	10	10	10	10	10	10	10	10	10	10	10	10
e_4	10	10	7	2	10	10	7	4	10	10	7	0
e_5	10	10	0	0	10	8	0	0	10	2	0	0
e_6	6	0	0	0	0	0	0	0	0	0	0	0
e_7	0	0	0	0	0	0	0	0	0	0	0	0

Table 8.1: The accuracy of using 16-bit half-precision floating point. Each unique matrix size and condition number contains ten different randomly generated matrices. Ten means all test cases are passed. A test case is considered as passed when both direct error and residual are smaller than the tolerance. Tolerance is denoted as $e_k = 2^{-k}$, where $k = 1, 2, ..., 8, 9$. For example, e_1 means the largest error is not above 2^{-1} and e_7 means the largest error is not above 2^{-7}. For all test cases the max error is not greater than 2^{-3}. It can be used if a real application does not need very high precision.

both high throughput and accuracy.

Glossary

AsAP3 The third generation Asyncronous Array of simple Processors (AsAP) chip. AsAP3 is a fine-grained many-core system with 1000 independently clocked homogeneous programmable processors, also named as Kilocore

BigMem An independent on-chip memory module used on KiloCore contains a 64 kB SRAM and is shared between two neighboring processors

FWL Fraction word length. It is used to show how many bits are used to represented fraction part in fixed point

GALS Globally asynchronous locally synchronous (GALS) is an architecture for designing electronic circuits which addresses the problem of safe and reliable data transfer between independent clock domains.

GJE Gaussian Jordan Elimination, an algorithem that used to invert the matrix.

IEEE-754 A techincal standard for floating point arithmetic and data representation. The standard species a set of formats, operations, rounding rules, flags, and the handling of exceptions.

MIMD In computing, MIMD (multiple instruction, multiple data) is a technique employed to achieve parallelism. Machines using MIMD have a number of processors that function asynchronously and independently. At any time, different processors may be executing different instructions on different pieces of data.

MIMO multiple-input and multiple-output is a method for multiplying the capacity of a radio link using multiple transmission and receiving antennas to exploit multipath propagation

TDP the thermal design power (TDP) is the maximum amount of heat generated by a computer chip or component (often a CPU, GPU or system on a chip) that the cooling system in a computer is designed to dissipate under any workload.

VLSI Very-large-scale integration (VLSI) is the process of creating an integrated circuit (IC) by combining millions of transistors or devices into a single chip.

Bibliography

[1] B. Bohnenstiehl, A. Stillmaker, J. J. Pimentel, T. Andreas, B. Liu, A. T. Tran, E. Adeagbo, and B. M. Baas. Kilocore: A 32-nm 1000-processor computational array. *IEEE Journal of Solid-State Circuits*, 52(4):891–902, 2017.

[2] P. Benner, P. Ezzatti, E. S. Quintana-Ortí, and A. Remón. Matrix inversion on cpu–gpu platforms with applications in control theory. *Concurrency and Computation: Practice and Experience*, 25, 2013.

[3] Girish Sharma, Abhishek Agarwala, and Baidurya Bhattacharya. A fast parallel gauss jordan algorithm for matrix inversion using cuda. *Computers & Structures*, 128:31 – 37, 2013.

[4] C. S., L. V., S. S., and M. J. Design and implementation of a floating point matrix inversion module using model based programming. In *2019 IEEE 16th India Council International Conference (INDICON)*, pages 1–4, 2019.

[5] R. Mahfoudhi, S. Achour, O. Hamdi-Larbi, and Z. Mahjoub. High performance recursive matrix inversion for multicore architectures. In *2017 International Conference on High Performance Computing Simulation (HPCS)*, pages 675–682, 2017.

[6] F. Ries, T. De Marco, and R. Guerrieri. Triangular matrix inversion on heterogeneous multicore systems. *IEEE Transactions on Parallel and Distributed Systems*, 23(1):177–184, 2012.

[7] Bevan Baas, Zhiyi Yu, Michael Meeuwsen, Omar Sattari, Ryan Apperson, Eric Work, Jeremy Webb, Michael Lai, Tinoosh Mohsenin, Dean Truong, and Jason Cheung. AsAP: A fine-grained many-core platform for DSP applications. *IEEE Micro*, 27(2):34–45, March 2007.

[8] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas. Kilocore: A fine-grained 1,000-processor array for task-parallel applications. *IEEE Micro*, 37(2):63–69, 2017.

[9] Z. Yu and B. M. Baas. A low-area multi-link interconnect architecture for gals chip multiprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 18(5):750–762, 2010.

[10] A. T. Tran and B. M. Baas. Achieving high-performance on-chip networks with shared-buffer routers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(6):1391–1403, 2014.

[11] R. W. Apperson, Z. Yu, M. J. Meeuwsen, T. Mohsenin, and B. M. Baas. A scalable dual-clock fifo for data transfers between arbitrary and haltable clock domains. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(10):1125–1134, 2007.

[12] Z. Yu and B. M. Baas. High performance, energy efficiency, and scalability with gals chip multiprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(1):66–79, 2009.

[13] Aaron Stillmaker, Brent Bohnenstiehl, and Bevan Baas. The design of the kilocore chip. In *ACM/IEEE Design Automation Conference*, Austin, TX, Jun. 2017.

[14] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas. Kilocore: A 32 nm 1000-processor array. In *IEEE HotChips Symposium on High-Performance Chips*, August 2016.

[15] Z. Yu and B. Baas. Implementing tile-based chip multiprocessors with gals clocking styles. In *2006 International Conference on Computer Design*, pages 174–179, 2006.

[16] Robert D Skeel. Scaling for numerical stability in gaussian elimination. *Journal of the ACM (JACM)*, 26(3):494–526, 1979.

[17] J. Liu, Y. Liu, X. Liu, and J. Zhou. A reconfigurable processor for matrix inversion computation. In *2018 IEEE International Conference on Integrated Circuits, Technologies and Applications (ICTA)*, pages 55–56, 2018.

[18] P. Salmela, A. Happonen, A. Burian, and J. Takala. Several approaches to fixed-point implementation of matrix inversion. In *International Symposium on Signals, Circuits and Systems, 2005. ISSCS 2005.*, volume 2, pages 497–500 Vol. 2, 2005.

[19] Lapack is a software package provided by univ. of tennessee; univ. of california, berkeley; univ. of colorado denver; and nag ltd.. v3.3. 2011.

[20] Nvdia cublas toolkit document v11.2. 2020.

[21] M. Butler. "bulldozer" a new approach to mult ithreaded compute performance. In *2010 IEEE Hot Chips 22 Symposium (HCS)*, pages 1–17, 2010.

[22] A. Stillmaker and B. Baas. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration, the VLSI Journal*, 58:74–81, 2017.

[23] Aaron Stillmaker, Zhibin Xiao, and Bevan Baas. Toward more accurate scaling estimates of cmos circuits from 180 nm to 22 nm. Technical Report ECE-VCL-2011-4, VLSI Computation Lab, ECE Department, University of California, Davis, December 2011.

[24] Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.