Linear Algebra for Array Signal Processing on a Massively Parallel Dataflow Architecture

Supervisor: Per Söderstam
Abstract

This thesis provides the deliberations about the implementation of Gentleman-Kung systolic array for QR decomposition using Givens Rotations within the context of radar signal processing. The systolic array of Givens Rotations is implemented and analysed using a massively parallel processor array (MPPA), Ambric Am2045. The tools that are dedicated to the MPPA are tested in terms of engineering efficiency. aDesigner, which is built on eclipse environment, is used for programming, simulating and performance analysing. aDesigner has been produced for Ambric chip family. 2 parallel matrix multiplications have been implemented to get familiar with the architecture and tools. Moreover different sized systolic arrays are implemented and compared with each other. For programming, ajava and astruct languages are provided. However floating point numbers are not supported by the provided languages. Thus fixed point arithmetic is used in systolic array implementation of Givens Rotations. Stable and precise numerical results are obtained as outputs of the algorithms. However the analysis results are not reliable because of the performance analysis tools.

Keywords: Am2000 family, Ambric register, aDesigner, radar, antenna arrays, beamforming, QRD, Gentleman-Kung systolic array, Givens Rotations, MPPA, massively parallel processor array, fixed point
Contents

Abstract ................................................................................................................................. 2

Contents ............................................................................................................................. 3

Acknowledgements ........................................................................................................... 5

1 Introduction .................................................................................................................. 5

2 Radars ........................................................................................................................... 6

2.1 Structure ..................................................................................................................... 6

2.1.1 Antenna ................................................................................................................... 6

2.1.2 Transmitter .............................................................................................................. 6

2.1.3 A/D Converter ........................................................................................................ 6

2.1.4 Signal Processing Unit ............................................................................................ 6

2.2 Phased Array Antennas .............................................................................................. 7

2.2.1 Beamforming ........................................................................................................... 8

3 The Architecture of Ambric Processor Array Family ..................................................... 11

3.1 Introduction ................................................................................................................ 11

3.2 Ambric Compute Unit, RAM Unit, Brics .................................................................... 12

3.3 Ambric Registers and Channels ................................................................................ 12

3.3.1 Registers ................................................................................................................ 12

3.3.2 Channels ................................................................................................................ 13

3.3.3 Hardware Objects & Clocking .............................................................................. 14

3.4 Processor Architecture .............................................................................................. 14

3.4.1 SR and SRD Processors ......................................................................................... 14

3.5 Memory Architecture .............................................................................................. 16

3.6 Programming Model .................................................................................................. 16

3.6.1 Structural Object Programming Model ................................................................. 17
4 Implementations ................................................................. 23

4.1 Matrix Multiplications ......................................................... 23

4.1.1 Matrix Multiplication with Single Node ........................ 23

4.1.2 Matrix Multiplication with Multipliers, Adder and Splitter .. 26

4.1.3 Matrix Multiplication with Mesh Algorithm ....................... 27

4.1.4 Comparison of the Algorithms ....................................... 28

4.2 QR Decomposition .......................................................... 30

4.2.1 Givens Rotations ......................................................... 31

5 Evaluation and Discussions .................................................. 41

6 Conclusions ........................................................................... 42

References ............................................................................... 44

Appendix .................................................................................. 46
Acknowledgements

This is my first thesis work. So I was not familiar with such a long project and with writing of the report. But I believe that, it was a great experiment for me and I’m glad that I had a project in the area that I am interested.

Because of the other courses, the project was going slow at the beginning but after finishing the courses I was able to focus on the thesis. So now it is over.

I would like to thank to my supervisor Per Söderstam for his great help, especially when I got stuck somewhere in the project. I would like to thank to Prof. Bertil Svensson because of giving me this opportunity to work on this project. I would like to thank also to Zain-ul-Abdin, Majid Ali Chaudhry, Muhammad Qasim Mughal and Ewa Janczewska for their help.

1 Introduction

The demand on the antennas increases by the time passes. The capabilities and variety of radars also increases by time. In some radars [15], more than one antenna is used. Thus, antenna arrays appear. This arrays can be linear or 2 dimensional. By using this antennas, beamforming can be performed which has a few reasons to be performed. By using beamforming the radar can focus on an angle, so it can ignore the other jammers and ground effect. It can be used also for tracking a target while going on scanning. In data communication area beamforming can be used to increase bandwidth. For instance, a transmitter can use different lobes for the receivers in different places. However, to be able to check a particular angle, some weights must be applied to the signals, that come from the antennas. These weights can be computed by using some methods.

This thesis mentions about a method called Minimum Variance Distortionless Response [7] which is used to find the weights of the antennas for a particular angle in an antenna array. In this method, matrix inversion, which is a complex operation, is used. To avoid this matrix inversion, QR decomposition (QRD) [10][19] can be used. QRD operation decomposes a matrix A into a unitary matrix Q and an upper triangular matrix R. There are several different forms of QR decomposition. One of them is Givens Rotations [9][14]. In givens rotations a set of matrices are applied to the data matrix A from right. In each step one of the values of A is zeroed. The Q matrix is obtained by multiplying the applied matrices.

The main aim of this thesis is to test a new MIMD architecture by performing QRD and performing QRD by implementing a Gentleman-Kung systolic array for Givens Rotations. The architecture is called Ambric architecture. The Ambric chip contains around 336 32-bit processors. Additionally, the tools that are produced for Ambric processor array will be tested. The test includes these points: performance analysis, engineering efficiency and potential of the architecture (scalability, compatibility... ). At the beginning, a few matrix multiplication algorithms will be implemented on the architecture. Then systolic array of Gentleman and Kung will be implemented. Different sizes of the systolic arrays will be compared with each other.
2 Radars

Radar (RAdio Detection And Ranging)[15] is a system of detecting, locating and tracking targets which are capable of reflecting high frequency radio waves (microwaves). Generally these waves are in the wavelength range from a fraction of a centimeter to tens of centimeters. Radar sends the high frequency signal and catches the reflection signal from the target and then extracts the desired information from the reflected signal.

2.1 Structure

Simple radar consists of a transmitter, a receiver, an antenna, an A/D converter, a signal processing unit and a display unit. In each radar, these parts can differ in terms of complexity. A block diagram can be seen in figure 1.

2.1.1 Antenna

The antenna is used to reflect the signal, like radiating the signal into the space or collecting all incoming signals to a point which can be the receiver.

2.1.2 Transmitter

High frequency transmitter creates a signal with pulse repetition frequency which is dependent on mode of operation and type of radar.

2.1.3 A/D Converter

A/D converter simply converts the received analog signal into digital signal to get it ready for any extraction. Before conversion, filtering can be done. It is being performed to get maximum signal to noise ratio. The signal is being sampled and converted into digital format and sent to the signal processing unit which is the Ambric chip in our case. The received signal is the reflection from the target.

2.1.4 Signal Processing Unit

In processing unit, the targets are being found by using all effects like jammers, ground reflection and receiver noise. The phase of the received signal is being determined by using
the phase of the transmitter as reference but it works only for an interval. All data that will be used together has to come in this interval so called ‘coherent processing interval’. The distance of target is being determined by checking the amplitude of range samples. The direction is determined by using the angle of antenna. Finally the velocity is being found by using the doppler shift. The velocity of target is also related to the velocity of the radar. To find the location, direction and velocity of the target, different methods can be used. For further details in antennas one shall read [15].

2.2 Phased Array Antennas

An antenna array consists of a number of active antennas which are connected to a common source or load, to produce a directive radiation. Phased array antennas are samples of electrically steered antennas. Instead of using a mechanical turn table which is really slow when compared to electronic devices, arrays of antennas are being used for directive radiation. An antenna array can have hundreds of small antennas and these small antennas are put together as linear or 2 dimensional arrays. By adjusting the phase, frequency or the amplitude of the transmitted signal in each antenna element, an interference pattern can be created. It becomes possible to send a signal to a particular direction and look for possible targets at that direction so called active beamforming [6][7][8][13][14]. However the cost is a high number of expensive microwave electronics and control mechanisms.

![Figure 2 - a) Antenna array with a main lobe and a spotlight b) Antenna array with a wide main lobe (k-channels, n-samples, w = vector of weights)](image)
2.2.1 Beamforming

Beamforming is a signal processing technique that is used in arrays of sensors (such as radars) to have a directional signal transmission and reception. By using the beamforming technique, spatial domain division can be reached which results faster search for targets and multiple mode operation such as tracking a target while scanning for another. (Spatial domain division is useful for also data communication area. Many users can connect to a center by using the same frequency when beamforming is used. For each user a lobe can be reserved.) It can be used for also ignoring the signals from some particular directions such as ground which is a big problem.

To check a particular direction, weights are being assigned to each antenna. The weights are determined by the direction of the chosen lobe. The signals from each antenna are being multiplied by this weights and summed up together to get a main signal from the lobe direction This operation has the same effect in case of the transmitted signal. The reason of using weights is as follows: each antenna element gets the reflected signal at a different time because of the difference in the distances that the signal travels to reach each antenna. One set of weight is needed for each direction. With N channels and K directions we get;

\[ Y(n) = \begin{bmatrix} y_1(n) \\ \vdots \\ y_K(n) \end{bmatrix} = W(t)x(n) = \quad (1) \]

\[ w = \begin{bmatrix} w_1 \\ \vdots \\ w_N \end{bmatrix}, \quad W = \begin{bmatrix} w_{11} & \cdots & w_{1N} \\ \vdots & \ddots & \vdots \\ w_{K1} & \cdots & w_{KN} \end{bmatrix} \quad (2) \]

Where \(Y(t)\) is the K-sized column vector and includes the total signal for each direction at time \(t\). \(y(t)\) is the weighted signal for each direction at time \(t\), also called beamformer response. \(w\) is the vector of weights and \(W\) is the applied weight matrix. Row \(k\) holds the weights for direction \(k\) and \(x(t)\) is the data set collected from antenna channels at time \(t\). \(w^T\) refers to transpose of \(w\). In general method, Hermitian matrix is used but in our case we do not have complex numbers. Thus we do not need to use Hermitian matrix.

For a given direction, an optimum set of weights for a particular direction \(\theta\) can be found by using the following formula, known as Minimum Variance Distortionless Response (MVDR) [7][14].

\[ w(\theta) = \frac{M^{-1}s(\theta)}{s(\theta)^TM^{-1}s(\theta)} \quad (3) \]

\(w(\theta)\) becomes the weight vector for direction \(\theta\).

\[ M = \frac{1}{m} \sum_{n=0}^{m-1} x(n)x^T(n) \quad (4) \]

\(M\) is the estimate of the cross covariance matrix of \(m\) samples from \(K\) channels.
\[ s(\theta) = \begin{bmatrix} 1 \\ e^{j(2\pi/\delta)D \sin(\theta)} \\ e^{j(2\pi/\delta)2D \sin(\theta)} \\ \vdots \\ e^{j(2\pi/\delta)(N-1)D \sin(\theta)} \end{bmatrix} \] (5)

\( s(\theta) \) is the \( N \) by 1 main beam steering vector related to the steering direction \( \theta \) where \( D \) is the distance between two antennas in the array and \( \delta \) is the wavelength.

The MVDR method, also referred to as the Capon’s Approach, minimizes the output power of antenna array by implementing a single linear constraint that maintains unit gain in the look direction. In common usage of MVDR, the covariance matrix of antenna array’s output is approximated by the covariance matrix of data samples. In our implementation the same method is used. However this method may cause undesirable high side lobes.

In the 3rd equation, there exists a matrix inversion operation which has \( O(n^3) \) complexity. At the same time, finding the correlation matrix and then inverting it needs longer words. That is a significant fact while using fixed point numbers. In order to avoid matrix inversion and solve the larger word length problem, QR decomposition (QRD) can be used on the sample matrix \( X \).

The QRD of a matrix is the decomposition of the matrix into an upper triangular matrix \( R \) and an orthogonal matrix \( Q \). QR decomposition of a real square matrix \( X \) is as below;

\[ X = QR \] (6)

or if \( X \) is not square it can be shown as:

\[ X_{mxN} = Q_{mxm} \begin{bmatrix} R_{N\times N} \\ 0_{(m-N)\times N} \end{bmatrix} \] (7)

\( X \) has the size \( mxN \) where \( m>N \), \( Q \) is orthogonal (unitary) which means \( Q^TQ = I \), and \( R \) is an upper triangular matrix (also called right triangular matrix).

If we assume \( X \) to be the consecutive vector sample matrix from the antenna channels, \( M \) can be shown as:

\[ M = \frac{1}{m} X^T X \] (8)

\[ X(n) = \begin{bmatrix} x^T(1) \\ \vdots \\ x^T(m) \end{bmatrix} \] (9)

By combining equation 7 (decomposed \( X \)) and equation 8, the cross covariance matrix \( M \) can be rewritten as:
\[ M = \frac{1}{m} X^T X = \frac{1}{m} \left( Q \begin{bmatrix} R_1 \\ 0 \end{bmatrix} \right)^T \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = \frac{1}{m} \left[ \begin{bmatrix} R_1 \\ 0 \end{bmatrix} \right]^T Q^T Q \begin{bmatrix} R_1 \\ 0 \end{bmatrix} \]

\[ \Rightarrow M = \frac{1}{m} R^T R \quad (10) \]

By inverting \( M \) we get the equation below:

\[ M^{-1} = m R^{-1} R^{-T} \quad (11) \]

By collecting the quotient in equation 4 (MVDR expression for the weights) into the constant \( \mu \), observing that \( m \)'s cancel each other and combining with equation 11, equation 3 (MVDR expression for weights) can be rewritten as following to extract the weights by solving the equation:

\[ w(\theta) = \frac{M^{-1}s(\theta)}{s(\theta)M^{-1}s(\theta)} = \frac{M^{-1}}{s(\theta)^T R^{-1} R^{-T} s(\theta)} s(\theta) \quad (12) \]

\[ \mu = \frac{1}{s(\theta)^T R^{-1} R^{-T} s(\theta)} \quad (13) \]

\[ \Rightarrow w = \mu R^{-1} R^{-T} s \quad (14) \]

By multiplying both sides with \( R^T R \) the following equation is obtained:

\[ \Rightarrow R^T R w = \mu s \quad (15) \]

Above equation can be solved by using backward and forward substitution.

If we assume that constant \( u = Rw \) then we obtain:

\[ R^T u = \mu s \quad (16) \]

Constant \( u \) can be computed related to the computation of \( \mu \) from equation 13. If we define \( z \) as below:

\[ z = R^{-T} s \quad \Leftrightarrow \quad R^T z = s \quad (17) \]

then \( \mu \) becomes:

\[ \mu = \frac{1}{z^T z} = \frac{1}{|z|^2} \quad (18) \]
By combining equ 16 and equ 18 we get the following equation:

\[ R^T u = \frac{1}{|z|^2} s \]  
\[ \Rightarrow u = R^{-T} \frac{1}{|z|^2} s \]

from equ 17,

\[ R^{-T} = z s^{-1} \]

\[ \Rightarrow u = \frac{zs^{-1} s}{|z|^2} = \frac{z}{|z|^2} \]  

As seen above z is computed by using \( R^T \) and s. Constant u has been computed by using z.

So the numerical problem (large [doubled] word length) of cross correlation matrix has been avoided and there is no need for matrix inversion anymore. These operations have been substituted by QR decomposition and two triangular systems. QR decomposition has \( O(mN^2) \) complexity and there are several methods to do it like Givens Rotations, Gram Schmidt and Householder reflections.

To find the angular distribution of the energy on the antennas, Singular Value Decomposition (SVD) [14] can be applied on the cross covariance matrix M, though there are several different ways to do it. However we will not go into further about SVD.

3 The Architecture of Ambric Processor Array Family

3.1 Introduction

The chips with multi processors always challenged with interconnecting hundreds of processors and memories with flexibility, high performance, low cost and low power. Moreover, challenged with how to program and validate a complex and irregular application on those processors efficiently which is the main issue. With conventional multiprocessing techniques, keeping the processors busy, communicating with one another and synchronization is not easy and can be failure-prone.

Ambric chip (Am2045) has around 180 million silicon transistors, over 300 32-bit processors. The memory is distributed to the processors with constant amount. The main issue of Ambric chip is Ambric register that is used through channels. The chip is asynchronous inside but the processors or the objects synchronize and communicate with one another through these configurable Ambric channels. In structural object programming, objects are strictly encapsulated software programs. They run concurrently on asynchronous array of processors and memories. The structure of these channels, (the synchronization between objects) is very useful for systolic array. (When an object sends a data to the other one, the data waits in the end register till the receiver object reads it and also the object can stall until a data comes.) We will implement and discuss it in further chapters.
There are a few structures on Ambric chip such as compute unit (CU), ram unit (RU) and brics.

### 3.2 Ambric Compute Unit, RAM Unit, Brics

Compute unit consists of 2 SRD and 2 SR processors. Each compute unit has a ram unit as a pair which consists of 4 RAM banks. Each of the rams has 2KB of memory. The RAM banks are reached through the RU engines. There is a configurable network between RU engines and RAM banks. Thus the RU engines connect to the RAM banks dynamically through the configurable network. There are channels also between compute units. The interconnection between two CPUs in CU is dynamic under instruction control. When 2 compute units and 2 ram units come together, they create a bric. In total it makes 8 CPUs and 21KB of RAM. Brics have their own circuit-switched interconnection network with configurable switches. This network always runs at the highest clock rate and connects compute units through clock crossing registers. There are 3 kinds of brics, general purpose brics, special purpose brics and interface brics. There are 45 brics on the AM2000 family Ambric chip.

![Figure 3 - Structure of a bric (From [5])](image)

### 3.3 Ambric Registers and Channels

This section deals with Ambric channels and registers which are the heart of the Ambric chip.

#### 3.3.1 Registers

Architecture of the massively parallel processor array (MPPA) is built on the Ambric registers. The capabilities of the architecture and the implementations depend on these registers.
Instead of ordinary registers, there are only Ambric registers throughout the Ambric system. They are different than the ordinary edge triggered registers but they still have some similarities. Ambric registers are also clocked registers and they have data in and data out. In addition to data in and data out there are accept and valid signals as control signals to implement a hardware protocol for local forward and backward pressure. This protocol makes the system self-synchronized and asynchronous.

The operations in Ambric registers are self initiated. When the register is ready to get an input it asserts it’s accept signal upstream. On the other hand when the register has an output to send, it asserts its valid signal downstream. Thus, when two registers, which are connected to each other, see both signals are true then they understand that transfer of the data can be done or the transfer of the previous data has already been done. As the result the registers become self-synchronized by using two signals. Since these dynamics are self contained, the synchronization becomes hidden.

Each register can hold two words. This happens in case of output stalls and requirement of incoming data. The register holds both the stalled output and the incoming data. Thus in a register chain with N registers, Nx2 data words can be held.

![Figure 3 - Ambric Registers (From Am2000 Family Architecture Reference [3])](image)

### 3.3.2 Channels

Ambric channels consist of Ambric registers. Channel is created by putting together few Ambric registers. The channels are scalable and the length depends on the number of registers in the channel. The channels are fully encapsulated and used for data and control token transfers between objects. They are ordered as first in first out and connected point to point. One other property of these channels which can be called as a drawback is being unidirectional. During implementations this property will be mentioned more.

Since each register has capacity of holding two data words, the channels can be used as small FIFOs as mentioned in previous section. Bigger FIFOs can be implemented in RAMs. These FIFOs can be used in channels transparently, to control the data flow or implement storage in the application when it is necessary. FIFOs often replace random access memories due to being self synchronized.
Ambric channels are used throughout the chip, inside the processors, memories and interconnects.

### 3.3.3 Hardware Objects & Clocking

A hardware object in Ambric can be combinational logic, state machine or RAM. An object is interconnected by Ambric registers. Internal synchronization is acquired by using `accept` and `valid` signals. Composite objects and complete applications are obtained by connecting objects. These objects are connected by Ambric channels, by sending the data; the synchronization between objects is acquired. Since the objects run independently and synchronized only through the channel interfaces, this system is called globally asynchronous and locally synchronous (GALS).

The synchronization protocol enables clock crossing registers. A clock crossing register works with different, but related, clocks on input and output. Thus, the objects that are connected to the ends of this register can run with their own speed. Objects can adjust their clock rates dynamically during an application. For instance when an object feels that it is stalling then it reduces the clock rate in order not to consume much power. So each object runs at the lowest clock rate possible.

The processors are also hardware objects. They are interconnected by Ambric channels and called leaf objects in programming model. They synchronize with each other by using transfer events. Since they are connected with Ambric registers they communicate as registers. If a processor wants to send a message to another, first it checks whether the receiver processor (`accept` signal) is ready to get it. If it is ready, then the sender processor sends the message otherwise it stalls until the receiver gets ready. The same protocol is valid for a processor which wants to get a message. This protocol is useful for systolic array implementation, since the processor stalls until it gets the input, and will be mentioned in next chapters.

### 3.4 Processor Architecture

There are two types of processors in MPPA, SRD and SR. These processors have some local instruction memory and they get the instructions through the channels. Processors execute the instructions streamed from local memory, by a program counter by using random access channels. Ambric processors do not have interrupt because there is no need since each processor is dedicated to own task.

#### 3.4.1 SR and SRD Processors

SR processors are 32 bit streaming RISC processors. SRD processors are also 32 bit streaming RISC processors but in addition to this they have DSP extension. So when compared, SRD has more capabilities than SR. Thus SR processors are mainly used for small tasks, when SRD is not necessary, like forking. SRD is more like extension of SR. Both processors can be programmed. The local memories are parity checked. Below a comparison table between SRD and SR processors can be seen:
<table>
<thead>
<tr>
<th><strong>SRD Processor</strong></th>
<th><strong>SR Processor</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>32 bits wide instruction</td>
<td>16 bits wide instruction</td>
</tr>
<tr>
<td>Accepts inputs from 2 channels and feeds an output to a channel each cycle</td>
<td>Accepts 1 input and feeds 1 output each cycle</td>
</tr>
<tr>
<td>Three ALUs (two in series and a parallel) including multiply and barrel shift</td>
<td>One ALU</td>
</tr>
<tr>
<td>20 general purpose registers + previous result register &amp; 64 bit accumulator</td>
<td>8 general purpose registers &amp; 32 bit accumulator</td>
</tr>
<tr>
<td>256 (32bits) words - local memory + 2K more instructions from the RAM unit next to it</td>
<td>64 (32bits) words - local memory</td>
</tr>
<tr>
<td>Streaming and random access to the RAM unit through a dedicated RAM unit read/write engine</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 4 - Architecture of SR processors (From [5])*
3.5 Memory Architecture

In the architecture of Ambric, like any other objects, memories are encapsulated and can be read and written by using channels. There are access engines between processors and memories. Access engines work in 2 forms. In first form access engines set some region of memories as FIFOs, which is the most common way of memory usage, and connect them to the channels. In the second form, access engines enable random access read and write. The written data is sent over channel like the read data. Requests can be: read or write the data.

The memory is distributed on the chip as ram units. In addition to this each processor has a small internal memory to keep instruction data. In case of requirement, an SRD processor can use an external memory, which is in RU, to keep the instruction data.

For more detailed information one can read [3].

3.6 Programming Model

To reach the practical programming model, components shall be familiar, productive, easy to use and the model should be scalable to any size.
To obtain the practical programming model, Ambric uses block diagram method because of its familiarity to parallelism. In Ambric, block diagrams are the objects and lines are the channels with defined behaviour. Thus, the whole block diagram becomes the structure of objects and channels.

By having hierarchy with interfaces and strictly encapsulated objects, design reuse becomes an important issue. So scalability becomes available. The objects can be reused even by different designs, because they do not share anything with any other which they can harm. Instead of using shared memory for communication, these objects communicate with each other through the channels which also arrange the synchronization between objects. Synchronization is done during runtime, not during compilation time.

Figure 6 - The hierarchy between design, structure and implementation parts of the programming model

The design may consist of several objects which are built by using astruct language. The connection between the objects builds the structure of the whole design. These objects can be either composite or leaf objects. Each leaf object is binded with a java file which is coded by using ajava. These objects can use different java files depending on the binding.

The memory is distributed with limited amount which is a constraint. However, it is not something new for the developers.

3.6.1 Structural Object Programming Model

Objects are programmed by conventional sequential code in structural object programming model of Ambric. (The compiler [aDesigner] is built on eclipse and uses ajava and astruct languages.) Smallest programmed object is called leaf which can be either a processor or a memory. Leaves can be connected to each other by channels, so comes out composite objects which can include leaves and/or other composite objects. Objects run independently with their own speed without having any effect on the others. When an object feels a stall or that it is idle, it reduces the clock frequency by itself in order not to consume much power.
On the other side, processors, memories and the channel hardware synchronize themselves transparently, dynamically and locally even they do not run with the same speed.

Channels provide hardware level interface. So connecting the objects becomes easy since they use only the channels for interaction. Also assembling the objects into composite objects becomes easy.

![Composite Objects](image1)

**Figure 7 - Structural Object Programming Model (From [5])**

During programming, conventional techniques can be used. However, the processors miss some operations like division and modulus. In addition to this, there is no floating point, and no array implementation except in constructors, which are two other constraints.

One can think that Ambric has the same model of computation (MoC) as Communicating Sequential Processes (CSP). However that way of thinking is not correct. MoC of Ambric is inspired by CSP. (CSP is a formal language to describe interaction patterns in concurrent systems [16].) However Ambric still has differences with CSP. In CSP components are sequential processes and they run concurrently. The objects in Ambric correspond to the components in the CSP. In CSP message passing is done synchronously which means there is no buffer between processors and they wait until a synchronization point to do the transfer. However in Ambric objects send messages to each other asynchronously and channels arrange the synchronization and the registers have small buffers. Additionally, Ambric MoC has similarities with Kahn Process Networks (KPNs) [17] and it can be defined as a process network with channels which are bounded FIFOs. In the other hand in KPNs the channels are unbounded FIFOs which means that you can always push data through the channels. In KPNs, get operation blocks until an input is available like the Ambric registers stall until there is a valid input. Another common property is that the objects (processes) can not check if there is an input on the channel. But sometimes this property becomes a constraint.
3.6.2 Programming and Analysis Tools & Languages

All implementations are done on a platform called aDesigner. aDesigner has been produced for Ambric. It has been built on Eclipse environment. Thus the programming language is built on java and called ajava. The syntax is the same. However, ajava has some limitations when compared to java. For instance all types of arithmetic operations are not supported in ajava. There is also a structure language called astruct, used to implement the structure of objects and so of the application. Creation of objects and channel connections between these objects are done in structure files. In addition to java, assembly can also be used. However, for the beginning java is much easier and we use java in our implementations. The structure files and the program files are binded to each other by the programmer while implementing the structure. Thus each object knows which program to run.

The program that will be run by the processor is written in ajava. Since java is a common programming language, programming the object is not a hard work, except some drawbacks of ajava. This program is sent to all objects that are binded to that java file. With astruct language, objects, numbers and names of input and output channels, properties (variables) of the objects, triggers, tags and the binded java files can be defined. Thus the structure of the object is built using astruct language and the task that will be processed is defined using ajava. These two languages complete each other. In our implementations we use neither tags nor triggers. Thus we do not go into further details about them. By connecting the objects with each other, composite objects are obtained. This operation is done with astruct language and while doing this operation, no binding with java file is used since there is no java code for composite objects. Defining the design of the application is also done by using astruct language. In design file, a sample of the highest level composite object is defined. If there are any input-output channels, they are connected to the proper inputs and outputs.

A structure file sample (which defines the channels, variables and java file of an object) can be seen below:

```plaintext
package single; //name of package that includes this object
interface calculator {
    inbound in;     //input channel declaration
    outbound out;  //output channel declaration

    /* the properties below, are used during object declarations for
dynamism purposes*/
    property int A_row;
    property int B_row;
    property int B_col;
}

//binding the object with the java file
binding Jcalculator implements calculator{
    implementation "Calculator.java"; //The java file that will be
run by this object
}
```

In the interface definition, i/o channels and properties of the object are defined. Properties are used to parameterize the object. The values of the properties are passed to the Java
constructors at compile time. They are static for the life of the object. In the binding
definition, the structure (object) is binded to a java file. It means that, the object will run that
java file. The object runs the run function of the java file as default. However this can be
changed by using triggers. We do not use triggers thus we will not go into further details
about them.

Another structure file sample (which defines a composite object) can be seen below:

```
package matrixMult2;

import matrixMult2.Adder;
import matrixMult2.Multiplier;
import matrixMult2.Splitter;

interface MatrixObj {
    inbound in;
    outbound out;
}

binding JMatrixObj implements MatrixObj{
    Splitter splitter;
    Multiplier mult1;
    Multiplier mult2;
    Adder adder;
    
    channel ch0 = {in, splitter.in},
    ch1 = {splitter.out_1, mult1.in},
    ch2 = {splitter.out_2, mult2.in},
    ch3 = {mult1.out, adder.in_1},
    ch4 = {mult2.out, adder.in_2},
    ch5 = {adder.out, out};
}
```

Splitter, Adder, Multiplier are the names of the structures and splitter, mult1, mult2 and
adder are the objects that are included in the composite structure. This composite structure
has an input and an output channel which are defined in the interface part. In the binding
definition of composite objects, the object does not get binded to any java file. Instead,
objects and the channel between objects are defined.

A design file sample (still a structure file).

```
package matrixMult2;
import matrixMult2.MatrixObj;
import astruct.pcie.Vio;

interface Root{}

binding CRoot implements Root{
    Vio vio = {numSinks = 1, numSources = 1};
    MatrixObj matrixObj;
    
    channel ch0 = {vio.out[0], matrixObj.in},
```
In the above design, Vio library defines inputs and outputs to the files in simulator. The matrixObj object, which is composite, is connected to these inputs and outputs. The top level interface is defined in the design file. This top level interface is the application itself.

aDesigner has two kinds of simulator: Java Virtual Machine (JVM)[4] and Instruction Set Simulator (ISS)[4]. JVM is more flexible than ISS. Unfortunately, performance analysis can only be done while using ISS. When JVM is selected as the simulator, all debugging features of java can be used and there is no limitation of java subset which is used to build designs for the hardware. In addition to the simulators, aDesigner creates machine code for hardware chip. So it can program the hardware device as well.

In aDesigner, as mentioned in processors chapter, there exist addition, subtraction, multiplication and barrel shift operations. There is no division operation and array definitions can be used only in constructor. These are two serious drawbacks. Another drawback is lack of floating points. There is only integer in aDesigner which forced us to use fixed points in our implementations. Fixed point arithmetic brings more operations and takes longer time than ordinary operations. A simple multiplication can take longer time while using fixed point arithmetic. The reason is converting the numbers into a Q format. Thus shifting and some more operations are done on the numbers.

There is a graphical window in aDesigner which shows the graphical structure of the application, with objects and the channel connections, after running the design. Though sometimes becomes useless because of the difficulty of changing the graphical layout. But still helps to see the connections between objects.

![Figure 8 - A screenshot of graphical layout of a design, that has been created by aDesigner (splitter, mult2, mult1 and adder are leaf objects)](image)

**Performance Analysis Tools in aDesigner and Analysis Method:**
To analyze the performance, aDesigner has an attachment on the simulator as an analyzer. But this analyzer is dedicated more to analyze code efficiency than analyzing the application run time. It allows the programmer to analyze only single processors by giving some cycle counts as total cycle count, busy cycle count and stall cycle count. Thus, neither composite objects nor the applications can be analyzed with this analyzer. In order to get the run time for entire application, we had to combine the cycle counts of each processor when they run in serial. So we needed to find out when they run in parallel and when they run in serial. In addition to the cycle counts, the analyzer gives also the datas in the registers like inputs, outputs and program counter with the first and last cycle numbers of the data’s duration time. (Like between 45th and 97th cycles there is 0x10000 in input1.) That is the main source we use for performance analysis in parallel implementations. The same analysis method is used for all implementations except single node matrix multiplication.

The analyzer ignores the channel stalls which means that it does not count the cycles when the processor waits for an input or when it waits to send an output. The length of a cycle (in terms of seconds) can change from a processor to another. Because the processors can run with different frequencies as mentioned before. So when a processors counts two cycles, the other one can count only one cycle or four cycles for the same time, depends on the frequencies of the processors. When there is any negative value in any register, the performance analyzer shows it as 0xffffffff. This becomes a serious obstacle for our analyzing method.

Because of the drawbacks and constraints, the performance analyzer is not reliable.

To get the performance analysis of a processor, the programmer needs to define the processor and a time interval (program counter’s start and end addresses). We use this method to get the performance analysis. In our case the time interval is from the beginning (pc address = zero) till the end of the process. There is another method. In this second method, the programmer defines a cycle count (t) and gets the performance analysis of defined processors each at each t cycles. However, it is not used in this thesis.
4 Implementations

4.1 Matrix Multiplications

To analyze the performance of a system, matrix functions are powerful and important tools. Thus to test the implementations in structural object programming and aDesigner, for the beginning 3 different types of matrix multiplications are used. The first one is matrix multiplication in serial.

4.1.1 Matrix Multiplication with Single Node

In this implementation there is only one object used. Thus, it is the easiest implementation for matrix multiplication. At first, the object reads the input matrices and saves them into arrays. Then multiplies them in a loop with an ordinary way and writes each result just after obtaining, to a file called capture file in aDesigner.

Assume A and B are 2X2 matrices as below.

\[
A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad B = \begin{bmatrix} 6 & 7 \\ 8 & 9 \end{bmatrix} \quad \Rightarrow \begin{bmatrix} A_{11}x B_{11} + A_{12}x B_{21} \\ A_{21}x B_{11} + A_{22}x B_{21} \end{bmatrix} \begin{bmatrix} A_{11}x B_{12} + A_{12}x B_{22} \\ A_{21}x B_{12} + A_{22}x B_{22} \end{bmatrix}
\]
The Multiplication Algorithm:

```java
for (int k = 0; k < A_row; k++)
    for (int i = 0; i < B_col; i++) {
        result = 0;
        for (int j = 0; j < B_row; j++)
            result = result + A[k][j] * B[j][i];
        out.writeInt(result);
    }
```

In each iteration a value from matrix A is multiplied by a value from matrix B. In the second loop, a value of the result matrix is obtained and written to the capture file in each iteration. Since three for loops are used, the complexity of the algorithm is $O(n^3)$.

Analysis of the algorithm:

The setup time of the program is 30 cycles for any size of matrices. In this case setup means getting the instruction codes and running the declarations. Then the first input is read and object goes on reading new inputs.

For multiplication of two $2x2$ matrices, which means there are 8 input datas, the last input is read at 243rd cycle. It means that the object gets ready to multiply the matrices and enters the loop that is given above. The first output is written at 382th cycle and the object writes the other results till 728th cycle. So the run time of the object becomes 728 cycles includes 30 cycles of setup time.

For multiplication of $2x4$ and $4x2$ matrices the number of inputs becomes two times of the previous implementation thus the number of the multiplications also becomes doubled. The first input is read at 30th cycle as well. The last input is read at 449th cycle and so object starts to multiply the matrices. The first output is written at 676th cycle and the last one is written at 1286th cycle. As the result the run time of the object becomes 1286 cycles includes 30 cycle of setup time.

For multiplication of two $4x4$ matrices, which means there are 32 input datas (16 from each matrix), the last output is written at 4207. cycle. So the run time is 4207 cycles.

For multiplication of two $8x8$ matrices, the run time becomes 27080 cycles.

Run time increase related to the sizes of matrices can be seen in the figure below:
Figure 9 - Increase of the cycle counts related to the matrix size (matrix size corresponds the size of both matrices that are multiplied with each other)

Since this algorithm has complexity $O(n^3)$ then the expected values can be calculated as below: (assume that $k$ is a constant multiplier)

\[
\begin{align*}
\text{For } 2\times 2; & \quad O(2^3) = 8k \\
\text{For } 4\times 4; & \quad O(4^3) = 64k \\
\text{For } 8\times 8; & \quad O(8^3) = 512k
\end{align*}
\]

\text{If we assume that; } 8k = 726, \quad \text{then; } 64k = 5808 \text{ and } 512k = 46464

As seen above the cycle count for $4\times 4$ matrices is expected to be 5808 and for $8\times 8$ matrices it is expected to be 46464. Because of this, we can say that the increase of the cycle counts is below the expected value. This shows that the algorithm runs better when the matrix size is big and the reasons are probably the setup times.
4.1.2 Matrix Multiplication with Multipliers, Adder and Splitter

![Diagram of Matrix Multiplication]

**Figure 10** - Matrix Multiplication with 2 multipliers, 1 adder and 1 splitter (Each shape corresponds to a different structure. Multipliers use the same structure and same program.)

Assume that A and B are the same matrices that are used in previous implementation. In this algorithm, the splitter reads the entire matrices from the input file which is called Stimulus file in aDesigner. Then sends the first value of A and the corresponding value of B to multiplier 1, then sends the second value of A and corresponding value of B to multiplier 2. When the multiplier gets both of the values it multiplies these values and send the result to the adder. After receiving values from both multipliers, adder sums them up and outputs as a value of the resulting matrix.

\[
A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times B = \begin{bmatrix} 6 & 7 \\ 8 & 9 \end{bmatrix} = \begin{bmatrix} A_{11}xB_{11} + A_{12}xB_{21} & A_{11}xB_{12} + A_{12}xB_{22} \\ A_{21}xB_{11} + A_{22}xB_{21} & A_{21}xB_{12} + A_{22} + B_{22} \end{bmatrix} \\
\Rightarrow \begin{bmatrix} 1x6 + 2x8 & 1x7 + 2x9 \\ 3x6 + 4x8 & 3x7 + 4x9 \end{bmatrix} = \begin{bmatrix} 22 & 25 \\ 40 & 57 \end{bmatrix}
\]

In the above multiplication, \(A_{11}xB_{11}\) is done by multiplier 1 and \(A_{12}xB_{21}\) is done by multiplier 2. Later, the results are added together in adder. Multiplier 1 keeps \(A_{11}\) for the next corresponding value \((B_{12})\) from matrix B for the next multiplication\((A_{11}xB_{12})\). Multiplier 2 also keeps the value from A until it multiplies that value with every corresponding value from B.

**Analysis:**

For the multiplication of above matrices (A and B), run time of splitter becomes 390 cycles (without setup time, we ignore it) because the splitter reads the entire matrices, saves them and sends proper values to each multiplier. After getting two values, the multipliers start the multiplication. So for a while they run in parallel with splitter. When splitter finishes, last multiplier runs for 7 cycles more and sends the last data to the adder. In addition to this adder also runs in parallel with the rest. After multipliers finish, adder runs for 2 cycles more and it also finishes. So the total run time becomes as below:
4.1.3 Matrix Multiplication with Mesh Algorithm

In this implementation shifter, splitter(s), adder(s) and multipliers are used. The matrices are divided into sub-matrices. Sub-matrices are sent to multipliers to be multiplied. After multiplying the sub-matrices, the multipliers shift the sub-matrices to the neighbour multipliers (sub-matrices of A horizontally and sub-matrices of B vertically) and multiply again until all sub-matrices are computed. The steps of the algorithm [18] can be seen below: (Assume that A and B are square matrices with sizes that can be divided by n.)

Step 1: Initialize matrix A by shifting i-th row i steps left (where 0 <= i < n)
   Initialize matrix B by shifting j-th column j steps up (where 0 <= j < n)
Step 2: Splitt sub matrices to the multiplier objects
Step 3: Multiply sub matrices of A and B and keep the result in C
Step 4: Shift matrix A one step left
   Shift matrix B one step up
Step 5: Multiply sub matrices of A and B and add to C
Step 6: Repeat step 4 and step 5 n-1 times

(The sub-matrices can also be just one value. n is the number of rows and columns of the multiplier objects.) A simple design of this algorithm with 4 multipliers can be seen below:

![Diagram of matrix multiplication with mesh algorithm](image)

**Figure 12** - Mesh algorithm implementation in aDesigner with a shifter, three splitters a combiner and four multipliers

**Analysis:**
The main drawback of this algorithm is number of channels. As can be seen from the figure above, there are many channels used to connect the objects, especially multipliers. Each multiplier has 6 channels connected. The reason of the high number of channels, that connects the multipliers, is unidirection property of the channels. Another drawback is length of setup time, because of shifting the matrices.

When we simulate this algorithm with 2x2 matrices, shifter’s run time becomes 439 cycle (without setup time which is 22 cycle, since the setup is done once we ignore setup time.) The splitter just takes the input and forwards it to the proper output and each operation takes one cycle, since the ALU in SRD can read and write in one cycle. So first and second steps, each adds only 1 cycle to the total time since they just read and write in one cycle. They run in parallel with each other and shifter. In total each of them runs 8 cycles but first 7 cycles in parallel. The total cycle count for splitting becomes 9 cycles. The multipliers directly start to run when they get two inputs (one from matrix A and the other from matrix B). Run time of a multiplier becomes 21 cycles. (Setup time is 3 cycles but we assume that in this 3 cycles multipliers run parallel with shifter and splitter, so we can ignore it.) The multipliers send the results to the combiner to write the results in an order to the output (capture) file. Run time of combiner is 4 cycles because it just reads the result and directly writes it to output. However, the combiner also runs in parallel with the multipliers thus combiner adds only 1 cycle (as a delay) to the total run time. So the total count of cycles becomes:

<table>
<thead>
<tr>
<th>Cycle Count</th>
<th>Shifter</th>
<th>439 cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>First step splitter</td>
<td>8 cycle</td>
<td></td>
</tr>
<tr>
<td>Second step splitters</td>
<td>8 cycle</td>
<td></td>
</tr>
<tr>
<td>Multipliers</td>
<td>21 cycle</td>
<td></td>
</tr>
<tr>
<td>Combiner</td>
<td>4 cycle</td>
<td></td>
</tr>
</tbody>
</table>

![Figure - 13](image_url) Total run time of the algorithm (462 cycle) with run time of each object (for multiplication of 2x2 matrices)

The time consumed while shifting the matrices is 92.8% of total run time.

4.1.4 Comparison of the Algorithms

It is easy to notice that the single node algorithm has the simplest implementation. In addition to this, mesh algorithm has the most complex implementation, especially in terms of objects and channels. If shifting task can be done in parallel then the run time of this algorithm can increase significantly since shifting consumes almost 93% of run time. If we compare total cycle counts, second algorithm seems to be the best one however, for bigger sizes mesh algorithm can show better performance. Because it is fully parallel in case of
multipliers. Thus, mesh algorithm is better then the other one in case of parallelism but it can still be improved (shifting task). In cycle count figures, it can be seen that mesh algorithm has the lowest cycle count for the multipliers. After initializing, the mesh algorithm shows a better performance. But it uses more multipliers (4).

Mesh algorithm uses 1 shifter, 3 splitter, 4 multipliers, 1 combiner.
The second algorithm uses 1 splitter, 2 multipliers, 1 adder.

Total cycle counts of both algorithms can be seen together in below figure.

Figure 14 - Run times of both algorithms for multiplication of 2x2 matrices

Speed up is calculated as below:

\[ \text{Speed up} (S) = \frac{\text{serial run time} (T_s)}{\text{parallel run time} (T_p)} \]

Efficiency is calculated as below:

\[ \text{Efficiency} (E) = \frac{\text{Speed up}}{\text{Number of processors used}} \]

For the second algorithm, the speedup is:

\[ S = \frac{726}{399} = 1.82 \]

Two multipliers are used in the second algorithm. Thus a speed up of 2 can be expected. However, there is also communication time. Thus 1.82 as a speedup is an expected value.

The efficiency of the second algorithm:
With big sized matrices the efficiency can get greater. The main reason of the efficiency being below 1 is the communication time.

For the mesh algorithm, the speed up is:

$$S = \frac{726}{462} = 1.57$$

Four multipliers are used in the mesh algorithm but the speed up is still lower than the second algorithm. The main reason is the shifting task. But also communication takes longer time. As mentioned before, when the matrix sizes increase, this algorithm probably shows a better performance than the second algorithm.

The efficiency of the mesh algorithm:

$$E = \frac{1.57}{9} = 0.174$$

The reason of low efficiency is again the communication time. Additionally, comes the shifting time in this algorithm.

### 4.2 QR Decomposition

Since the reasons of using QRD were given in the beamforming chapter, it is vital to make a summary of it to remind.

The QRD of a matrix is the decomposition of the matrix into an upper triangular (R) and an orthogonal matrix (Q). QR decomposition of a real square matrix $X$ is as below;

$$X = QR$$

or if $X$ is not square it can be shown as $(m>N)$:

$$X_{mxN} = Q_{mxm} \begin{bmatrix} R_{N\times N} \\ 0_{(m-N)\times N} \end{bmatrix}$$

However in our implementation we just ignore the zeros that are below the upper triangular R matrix. Because while computing the weights, the part with zeros disappears.

There are several methods to compute QRD like Givens Rotations, Householder and the Modified Gram-Schmidt decomposition. We will implement Givens Rotations with systolic array.
4.2.1 Givens Rotations

In this method, a number of unitary rotation matrices, \( G \), are applied to the data matrix \( A \) from right side. In each step one of the values in matrix \( A \) is turned into zero. The transpose of rotation matrices can be applied from left side. Generally the matrices have complex values. However, complex numbers are not supported in java, thus, we do not use them.

The unitary matrix \( G \) (Givens rotation matrix) is the identity matrix with four substitutions. The rotation matrix for zeroing the value of \( A \) which stays at \((i,k)\) needs the following substitutions:

\[
G(k,k) = c \\
G(k,i) = s \\
G(i,k) = -s \\
G(i,i) = c
\]

Thus the rotation matrix becomes:

\[
G(i, k) = \begin{bmatrix}
1 & \ldots & 0 & \ldots & 0 & \ldots & 0 \\
\vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\
0 & \ldots & c & \ldots & s & \ldots & 0 \\
\vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\
0 & \ldots & -s & \ldots & c & \ldots & 0 \\
\vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\
0 & \ldots & 0 & \vdots & 0 & \ldots & 1
\end{bmatrix}
\]

and when the rotation matrix is applied, the following equation is obtained:

\[
A G_{(i,k)} = G^T_{(i,k)} A = A'
\] (16)

Since the task of the rotation matrix is to zero the chosen value of matrix \( A \), the following equations have to be verified:

\[
c \cdot a_{k,k} + s \cdot a_{i,k} = 0
\] (17)

\[
c^2 + s^2 = 1
\] (18)

If we go further, \( c \) and \( s \) can be shown as:

\[
C = \frac{a_{k,k}}{\sqrt{a_{k,k}^2 + a_{l,k}^2}}
\] (19)

\[
S = \frac{a_{l,k}}{\sqrt{a_{k,k}^2 + a_{l,k}^2}}
\] (20)

The multiplication of the rotation matrices gives the \( Q \) matrix.

When it comes to finding the \( r \) value (in the systolic array):

\[
\begin{bmatrix}
C & S \\
-S & C
\end{bmatrix}
\begin{bmatrix}
a \\
b
\end{bmatrix}
= \begin{bmatrix}
r \\
0
\end{bmatrix}
\]

\[(c \cdot a + s \cdot b)^2 = r^2\]
\[ \Rightarrow c^2a^2 + s^2b^2 + 2casb = r^2 \]

\[ (-s * a + c * b)^2 = 0^2 \]

\[ \Rightarrow s^2a^2 + c^2b^2 - 2sacb = 0 \]

If we sum up the equations:

\[ \Rightarrow c^2a^2 + s^2b^2 + 2casb + s^2a^2 + c^2b^2 - 2sacb = r^2 \]

\[ a^2 \left( c^2 + s^2 \right) + b^2 \left( s^2 + c^2 \right) = r^2 \]

(remember equ. 18)

\[ r^2 = a^2 + b^2 \]

Thus c and s becomes (by using equ.19 and equ.20):

\[ c = \frac{a}{r} \]

\[ s = \frac{b}{r} \]

### 4.2.1.1 Systolic Array Implementation of Givens Rotations

Systolic arrays are implemented in hardware for specific tasks. They are often optimal in terms of communication and computation. Thus all parts in the array can be kept active. These arrays are implemented on special purpose hardwares and they are dedicated for the task. This makes the array very powerful and fast for the task in hand. In addition to this, it does not have any flexibility. Thus it can not be used for different sized applications. However, in our implementation, the size of the systolic array can be changed, which results in flexibility. The size of the array depends on an argument in the implementation. Thus the implementation becomes a dynamic systolic array implementation and the size can be changed to any number. Because of the architecture, systolic array suits very well on Ambric. No synchronization is needed. While implementing the systolic array on conventional processors, the programmer needs to send a bunch of zeros to the array as initial values of matrix. In Ambric case, there is no need for these initial values, because the processor stalls until it gets all corresponding datas and then starts to compute. The systolic array implementations can be seen below:
Figure 15 - Conventional systolic array and Ambric systolic array
The systolic array implementation that we use in this thesis is as below:

![Systolic Array Diagram](image)

**Figure 16 -** Systolic array implementation with Ambric architecture for matrices with four columns. **a)** Boundary cell, **b)** Inner cell, **c)** Splitter, **d)** Joiner

In our implementation, data comes in through IN channel to the first splitter, row by row like: X(1,1), X(1,2), X(1,3), X(1,4), X(2,1), X(2,2)…. First splitter checks out whether the data shall be sent to the array or to the neighbour splitter. All splitters do the same, except the last one. It sends all inputs directly to the array. By this way, all cells that are in the first row of the array get the input at the same time. Then the cells begin to compute c, s, r and x_out values.

Boundary cells process the following operations:

```c
if(x_in == 0){
    c = 1;
    s = 0;
}
else{
    int a = fixMult(r, r, 16, 16, 16);
    int b = fixMult(x_in, x_in, 16, 16, 16);
    r_tmp = fixSqrt(a+b, 16, 16);
    c = fixDiv(r, r_tmp, 16, 16, 16);
    s = fixDiv(x_in, r_tmp, 16, 16, 16);
    r = r_tmp;
}
```

Since there is no floating point numbers in Ambric, we needed to implement fixMult, fixSqrt and fixDiv functions to use fixed points. We use Q16.16 fixed point format in our
implementations. When we compare our results to the results from MATLAB 7.0, maximum precision difference is around ±0.1, however this difference mainly depends on the resulting value. fixMult returns the multiplication of two numbers with wanted Q format, fixSqrt returns the square root of the given number with wanted Q format and fixDiv returns the division of two numbers with wanted Q format. Thus in traditional way the operations would be:

```c
if(x_in == 0){
    c = 1;
    s = 0;
}
else{
    int a = r*r;
    int b = x_in*x_in;
    r_tmp = sqrt(a+b);
    c = r/r_tmp;
    s = x_in/r_tmp;
    r = r_tmp;
}
```

Inner cells process the following operations:

```c
x_out = fixMult(c, x_in,16,16,16) - fixMult(s, r,16,16,16);
```

```c
r = fixMult(s, x_in,16,16,16) + fixMult(c, r,16,16,16);
```

In traditional way:

```c
x_out = c*x_in - s*r;
```

```c
r = s*x_in + c*r;
```

Initial value for r is zero in both kinds of cells. Boundary cells run 2 fixed multiplication, 1 fixed square root, 1 addition and 2 division operations. Inner cells run 4 fixed multiplication and 2 addition operations. In the later stages it will be seen that, in worst case, boundary cells do approximately 8 times more computation than inner cells. It happens due to the complexity of division and square root functions with fixed point arithmetics.

Joiners are used to collect r values and write them in an output file. All cells send the r value at the proper time (after computing the entire matrix) to the neighbour cell until it reaches the joiners. Each cell sends at first own r value and then the other r values that come in from neighbour cell. In the joiners, the method is similar. Each joiner sends at first the r values that come from the array (r_in2 channel). Then it sends the r values that come from neighbour joiner (r_in1 channel). Thus the r values are sent row by row, as the input matrix.

The mathematics behind systolic algorithm is zeroing a vector of an already decomposed matrix.
Analysis:

In the systolic array implementation different kinds of cells do different computations. Sometimes this causes different cells to run with different frequencies to keep up working together with the other cells. Thus the cells send the data to the outputs approximately at the same time. All inner cells do the same computation, so their run times are approximately the same. However, boundary cells have two conditions and these conditions change the run time. \( \text{if} (x_{\text{in}} == 0) \) If this statement (in boundary cell) becomes true than the run time becomes really short because then boundary cell does not calculate anything. But if that statement is not true than the cell computes 2 fixed multiplications, 1 fixed square root, 1 addition and 2 division operations. The latter consumes much time. Thus the cell may need to increase the frequency. If the cell is running at the highest frequency, than the other cells stall while waiting for data from boundary cells. As mentioned before, when the processors feel that they are in a stall, they decrease the frequency in order not to consume much power.

For QRD of a 2x2 matrix if we assume the splitters run frequency is \( N \) then the inner cells run frequency becomes \( 4N \), joiners run frequency becomes \( 2N \) and the boundary cells run frequency becomes \( 32N \). We obtain these results by checking the cycle counts, between two data transfers, in the cells which take part in the transfer. We will use the boundary cells’ frequency as reference (probably 300 Mhz). We will ignore the setup time for all cells since it is not a significant issue. The systolic array used for QRD of 2x2 matrix can be seen below

![Figure 17 - Systolic array for matrices with 2 columns (number of rows can be 2 or higher)](image)

The splitters run for 2240 cycles (actually the run time is 70 cycles but since we use frequency of boundary cells then it becomes 2240 (70x32) cycles.) First row of the array runs 754 cycles in parallel with the splitters and after splitters stop, it goes on running 424 cycles more. Since the second row is doing the same computations, theoretically it has to run during the same cycle count as first row. When we want to check it practically, the performance analyzer does not help us because of the negative values of \( x_{\text{in}} \) that comes into boundary cell. (As mentioned before, when the value in a register is negative, the performance analyzer shows it as 0x7fffffff.) Thus we use theoretical assumptions. Then we assume that second row runs 754 cycles in parallel with the first row and then runs 424 cycles more. Joiners run 400 cycles in parallel with the array then they run 592 more cycles. The entire run time can be seen in the figure below:
**Figure 18** - Total runtime of systolic array for 2x2 matrix (frequency of boundary cells is used as reference)

**For QRD of a 4x4 matrix** splitters are the slowest objects as in the previous analysis. If we assume that their frequency is $3N$ then, frequency of inner cells becomes $6N$, frequency of joiners becomes $4N$ and frequency of boundary cells becomes $48N$. We use the frequency of boundary cells as reference. The systolic array implementation for 4x4 matrix is as below:

![Systolic Array Diagram](image)

**Figure 19** - Systolic array for matrices with 4 columns (number of rows can be 4 or higher)

The splitters run for 5808 cycles. When the first row gets the inputs, it begins to run and runs for 4392 cycles in parallel with splitters then runs 2376 cycles more. Since theoretically each row is doing the same thing then we assume that they run during the same amount of cycles.
as first row. When we try to check it practically, we again have problems like negative data in registers. However, we were able to check one more row (3rd row) and it gave the same result as first row (4392 cycles in parallel with the row above and then 2376 more cycles). The only difference between rows is the number of ‘r’s those are sent to the joiners. Sending each r value makes a few cycles of difference and in pure systolic array, r values are not passed on to the neighbours. Thus we ignore the difference that is caused by ‘r’s. The joiners run for 588 cycles in parallel with the array then run 564 cycles more and so the run time reaches the end. The entire runtime can be seen in the figure below:

![Figure 20 - Total runtime of systolic array for 4x4 matrix (frequency of boundary cells is used as reference)](image)

**Figure 20 -** Total runtime of systolic array for 4x4 matrix (frequency of boundary cells is used as reference)

For **QRD of a 8x8 matrix**, splitters are again the slowest objects and the frequency rates are all same as the previous analysis (qr of a 4x4 matrix). We again assume that the frequency of the splitters is 3N, the frequency of the joiners is 4N, the frequency of the inner cells is 6N and the frequency of the boundary cells is 48N. We use the frequency of the boundary cells as well. The systolic array implementation for 8x8 matrix is as below:
We begin again with the splitters. The splitters run for 43584 cycles. The first row joins the splitters and run 10152 cycles in parallel and then runs 2600 cycles more. We assume that the other rows also run during the same amount of cycles. We ignore the differences caused by the number of ‘r’s again. The joiners run 1164 cycles in parallel with the array and then they run 684 cycles more so the runtime reaches the end. The entire runtime can be seen in the figure below:

**Figure 21** - Systolic array for matrices with 8 columns (number of rows can be 8 or higher)
If we compare the results of qrd of 2x2, 4x4, 8x8 matrices, we see that the results of 4x4 and 8x8 can be related to each other. In 4x4 array, each cell in the array (boundary and inner cells) runs the codes for 4 times and the result (R matrix) comes out. This operation goes on for 6768 cycles. In 8x8 array, each in the array runs the codes for 8 times (2 times of the cells in the 4x4 array) and the result (R matrix) comes out. This operation is 2 times of the operation in 4x4 array and runs for 12572 cycles which is approximately 2 times of 6768. 2x2 array gives totally different results and even the frequency rates are different, however it can be because of having only one inner cell. The cycle count that each row adds to the total cycle count is 2376 in 4x4 array and 2600 in 8x8 array. They are approximately equal. One of the reasons of the small difference can be the ignored r values. Thus we can say that, run time of the systolic array (without splitters and joiners) increases linearly with the size of the matrix. Total cycle counts with corresponding matrix sizes, can be seen below:
5 Evaluation and Discussions

When the programmer wants to create a dynamic implementation (like dynamic systolic array), he may need to use division operation. However, division operation is not supported. Thus, he may need to implement own division operation or use shifting right which just allows to divide by 2 for each step. (We used shifting as division, while implementing dynamic systolic array.) If the programmer may need modulus operation, he needs to implement it too. Additionally, all array implementations must be done in constructors (in aJava). There is only integer in Ambric, so if the programmer needs any floating point number, he has to use fixed point numbers. Thus he needs to implement the operations by using fixed point arithmetics. On the other side, aStruct and aJava languages are not difficult to get used to. Since there are guiding pdf files, they get easier to use. However, forgetting about a small thing may cause big problems such as using `out.write(value)` instead of `out.writeInt(value)` which may take two days to realize in aJava. The codes can be debugged in aDesigner. The debugger works well however sometimes (like debugging for along time) it can stop and quit debugging and give strange errors.

The performance analyzer is not very useful to analyze the entire application’s run time. In addition to this, different designs can not be compared reliably because of variable cycle length. As mentioned before, the analyzer is dedicated to analyze the code efficiency. The measurements, that we did, are not perfect. Because they depend on some theoretical assumptions and the numbers we get from the analyzer are not reliable since the analyzer ignores the channel stalls. Because of the problem with negative values in registers, the
programmer may run into trouble while analyzing. Analyzing an application, which runs for a long time, can take long time. aDesigner works well itself but we can not say the same thing for the performance analysis tools.

The programming language (ajava) is not difficult to use since it is similar to java. The structure language is easy too. There is no need for a special effort to get used to the languages. However, there are small things like you have to use brackets whenever you use a for loop in astruct. Creating objects and writing codes for the objects are not big deal. Object reuse seems to be quite useful but we did not use this method.

For matrix multiplication we tried mesh algorithm which is totally parallel. However because of unidirection property of the channels, we needed to define 21 channels for multiplication of only 2x2 matrices. This number does not increase linearly but slower with the size of matrices but it is still too high. In addition to this, cells need to be synchronized and channels arrange it transparently.

As mentioned before, systolic array suits well on the architecture of the Ambric. It is totally parallel and the cells step all together. However because of using division and square root operations, the boundary cells slow down the entire array. The performance comparison of the systolic array can be done only locally. It can not be compared with different designs but with different sized systolic arrays with same design. Because we do not have any runtime value as seconds but cycles which are local for the Ambric. One can try to convert the cycles to seconds, but there is no clear knowledge about the reference (boundary cell) frequency. It may be the highest frequency which is 300 Mhz.

The communication in the array, does not take much time when it is compared to the computation time. Because each cell communicates only with the cells in the neighbourhood. Thus each communication step takes a few cycles when the computation step takes a few thousands cycles.

6 Conclusions

The main purpose of this thesis was to implement a systolic array of Givens Rotations on Ambric processor array and analyze the performance. At the beginning, three matrix multiplication algorithms have been implemented and analyzed to get familiar with the architecture, languages and aDesigner. In matrix multiplication algorithms, only integers are used. Additionally, no division operation is used. Only the operations that are supported by aDesigner are used. Thus no additional implementation is needed. Second algorithm (the one using only 2 multipliers) seems to be working better however the mesh algorithm would show better performance with bigger sized matrices.

In the second step, static systolic array is implemented. Fixed point arithmetics library have been created. The library includes multiplication, division and square root operations.

In the third step, dynamic systolic array is implemented. So the size of the array is made to be dependent on an argument.

The thesis shows that the architecture of the MPPA is suitable for systolic array of Givens Rotations. The systolic array gives stable and precise numerical values. In MATLAB 7.0, the
result values have 4 digits of fraction when in our implementation they may have more. The pure implementation shows a linear increase of runtime with the increasing matrix size which is expected.

In future works, one can use a cell as a bigger portion of the array, thus more antenna channels can be used. The cell can compute more than one $r$ value. Also different QRD methods can be implemented on Ambric processors, like Householders or Gram-Schmidt. However, probably, the programmer will not be able to compare the performances of different methods. He should take in account that the performance analyzer part of aDesigner is not very useful. Additionally the fixed point arithmetics library can be used for any kind of application.
References

[1] aDesigner Reference
[2] aDesigner Users Guide
Appendix

Codes

The ajava and astruct codes, that are used while implementing systolic array of Givens Rotations, are given below:

**Astruct codes of boundary cells**

```java
interface boundary {
    inbound b_in;
    outbound b_out;

    property int row; //number of rows of the data matrix
    property int sub_col;
    property boolean last;
}

binding Jbound implements boundary{
    implementation "Boundary.java";
}
```

**Ajava codes of boundary cells**

```java
public class Boundary {
    Math amath = new Math();
    private int sub_col;
    private int row;
    private int[] r;
    private int[] x_out;
    private boolean last;
    private int counter;
    private int tmp;

    public Boundary(int row, int sub_col, boolean last){
        this.row = row;
        this.sub_col = sub_col;
        this.last = last;
        counter = 0;
        tmp = (sub_col*(sub_col+1))>>1;
        r = new int[tmp];
        for(int i=0; i<tmp; i++)
            r[i]=0;
        x_out = new int[(sub_col*(sub_col-1))>>1];
    }
}
```
public void run(InputStream<Integer> b_in,
OutputStream<Integer> b_out){
    int x_in, c, s;
    int count=0;
    int count2=0;
    int count3=0;
    counter++;

    for(int j=0; j<sub_col; j++){
        x_in = (j == 0) ? b_in.readInt() : x_out[count++];

        if(x_in == 0){
            c = 65536;
            s = 0;
        }
        else{
            int a = fixMult(r[count2],r[count2],16,16,16);
            int b = fixMult(x_in,x_in,16,16,16);
            int r_tmp;
            //to avoid sqrt if a=0 (also helps with the precision)
            if(a==0)
                r_tmp = x_in<0 ? -x_in :  x_in;
            else
                r_tmp = fixSqrt(a+b,16,16);

            c = fixDiv(r[count2], r_tmp, 16, 16, 16);
            s = fixDiv(x_in, r_tmp, 16, 16, 16);
            r[count2] = r_tmp;
        }
        count2++;
    }

    //inner cell
    for(int i=j+1; i<sub_col; i++){
        x_in = (j == 0) ? b_in.readInt() : x_out[count++];

        x_out[count3++] = fixMult(c, x_in,16,16,16) -
                        fixMult(s, r[count2],16,16,16);
        r[count2] = fixMult(s, x_in,16,16,16) +
                    fixMult(c, r[count2],16,16,16);
        count2++;
    }

    if(row!=2 && !last){
        b_out.writeInt(c);
        b_out.writeInt(s);
    }
}

if (counter == row)
    for(int i=0; i<tmp; i++)
        if(r[i]<0)
            b_out.writeInt(-r[i]);
        else
            b_out.writeInt(r[i]);
}
Astruct codes of inner cells:

```java
interface inner {
    inbound from_left;
    inbound from_up;
    outbound to_right;
    outbound to_down;

    property int x;  //row-coordinate
    property int y;  //column-coordinate
    property int row; //number of rows of the data matrix
    property int col; //number of columns
    property int sub_col;
}

binding Jinner implements inner{
    implementation "Inner.java";
}
```

Ajava codes of inner cells:

```java
public class Inner {
    Math amath = new Math();
    private int sub_col;
    private int col;
    private int row;
    private int[] r;
    private int[] x_out;
    private int x;  //x coordinate
    private int y;  //y coordinate
    private int counter;

    public Inner(int col, int row, int sub_col, int x, int y){
        this.col = col;
        this.row = row;
        this.sub_col = sub_col;
        this.x = x;
        this.y = y;
        int tmp = sub_col*sub_col;
        r = new int[tmp];
        x_out = new int[sub_col];
        counter = 0;
        for(int i = 0; i<tmp; i++)
            r[i]=0;
    }
}
```
public void run(InputStream<Integer> from_left,
           InputStream<Integer> from_up, OutputStream<Integer> to_right,
           OutputStream<Integer> to_down){

        int x_in, c, s;
        counter++;
        c = from_left.readInt();
        s = from_left.readInt();

        if(x<(col-1)){ //if the cell is in the last column then
dont send c&s
            to_right.writeInt(c);
            to_right.writeInt(s);
        }

        //calculate for the first row of the processor
        for(int i=0; i<sub_col; i++){
            x_in = from_up.readInt();

            x_out[i] = fixMult(c, x_in,16,16,16) - fixMult(s, 
                             r[i],16,16,16);
            r[i] = fixMult(s, x_in,16,16,16) + fixMult(c, 
                             r[i],16,16,16);
        }

        //the other rows:
        for(int j = 1; j<sub_col; j++){
            //read c-s for each row
            c = from_left.readInt();
            s = from_left.readInt();

            if(x<(col-1)){ //if the cell is in the last
column then dont send c&s
                to_right.writeInt(c);
                to_right.writeInt(s);
            }

            //each element in the row
            for(int i = 0; i<sub_col; i++){
                x_in = x_out[i];
                int temp = j*sub_col+i;
                x_out[i] = fixMult(c, x_in,16,16,16) -
                             fixMult(s, r[temp],16,16,16);
                r[temp] = fixMult(s, x_in,16,16,16) +
                             fixMult(c, r[temp],16,16,16);
            }
        }

        for(int i=0; i<sub_col; i++)
            to_down.writeInt(x_out[i]);

        if(counter == row){
            int tmp1 = (sub_col*(sub_col+1))>>1;
            tmp1 = (x-y-1)*(sub_col*sub_col)+tmp1;
            if(sub_col>1)
for (int i = 0; i < tmp1; i++)
    to_right.writeInt(from_left.readInt());
// first send the other r's
// then send own r
for (int i = 0; i < (sub_col*sub_col); i++)
    if (r[i] < 0)
        to_right.writeInt(-r[i]);
    else
        to_right.writeInt(r[i]);

    counter = 0;
}
}

Java codes of fixed point operations:

Division:

    public int fixDiv(int N, int D, int qn, int qd, int q){
        // adjust argument fix
        int d = qn - q;
        N = d < 0 ? N << -d : N >> d;
        d = qd - q;
        D = d < 0 ? D << -d : D >> d;

        // adjust for signs
        int sign = 1;
        sign = N < 0 ? sign * -1 : sign;
        sign = D < 0 ? sign * -1 : sign;
        N = N < 0 ? -N : N;
        D = D < 0 ? -D : D;

        // check for valid input
        if (D == 0){
            return (sign > 0) ? Integer.MAX_VALUE:
                sign > 0) ? Integer.MAX_VALUE:
                    Integer.MIN_VALUE;
        }

        D >>= 1;  // to have enough space for denominator to be
greater than numerator
        N >>= 1;
    // condition arguments
        int m = 0;
        while (D <= N){
            D <<= 1;
            m += 1;
        }
    // do division
        int P = N;
        int Q = 0;
        for (int i = 31; i >= 0; i--){
            P = ((P << 1) - D);
            if (P >= 0){
Q = Q | (0x1 << i);
}

else{
    P += D;
}
}

// adjust to fix
Q = (Q >> (32 - q)) << m;
// adjust for signed integer operands
Q = Q & (0x7fffffff >> (32 - q - m - (Q < 0 ? 1: 0)));

return (int) (sign * Q);
}

Square Root:
/**
 * This is the Frieden algorithm from Crenshaw.
 * @param a Operand.
 * @param qx Fixpoint of operand.
 * @param q Desired fixpoint of result.
 * @return Fixpoint aligned square root of operand a.
 */
public int fixSqrt(int a, int qx, int q){
    int rem = 0;
    int root = 0;
    for(int i = 0; i < 16; i++){
        root <<= 1;
        rem = (rem << 2) + ((a >> 30) & 0x3);
        // last AND term to compensate for signed int
        a <<= 2;
        root += 1;
        if(root <= rem){
            rem -= root;
            root += 1;
        }
        else{
            root -= 1;
        }
    }
    root = root >> 1;
    int d = (qx >> 1) - q;
    root = d < 0 ? root << -d: root >> d;

    return root;
}

Multiplication:
/**
 * This is the Frieden algorithm from Crenshaw.
 * @param a Operand.
 * @param qx Fixpoint of operand.
 * @param q Desired fixpoint of result.
 * @return Fixpoint aligned square root of operand a.
 */
if(dq >= 0){
    return ((hi << (32 - dq)) | lo >>> dq);
} else{
    return lo << -dq;
}

Abstract codes of the joiner cells

interface joiner {
    inbound from_down, from_left;
    outbound j_out;
    property int loop1;
    property int loop2;
}

binding Jjoiner implements joiner{
    implementation "Joiner.java";
    attribute CompilerOptions(targetSR = true) on Jjoiner;
}

Abstract codes of the splitter cells:

interface splitter {
inbound in;
outbound to_left, to_down;

property int id;
property int sub_col;
}

binding Jsplitter implements splitter{
    implementation "Splitter.java";
    attribute CompilerOptions(targetSR = true) on Jsplitter;
}

Java codes of the splitter cells:

public class Splitter {

    private int id;
    private int counter;
    private int sub_col;

    public Splitter(int id, int sub_col){
        this.id = id;
        this.sub_col = sub_col;
        counter = 0;
    }

    public void run(InputStream<Integer> in, OutputStream<Integer> to_left, OutputStream<Integer> to_down){
        if(counter == id){
            for(int i=0; i<sub_col; i++)
                to_down.writeInt(in.readInt());
            counter = 0;
        }
        else{
            for(int i=0; i<sub_col; i++)
                to_left.writeInt(in.readInt());
            counter++;
        }
    }
}

Codes of systolic array composite object:

import givens_rotations.boundary;
import givens_rotations.inner;
import givens_rotations.splitter;
import givens_rotations.joiner;

interface systolic_array {

    inbound in;
    outbound out;

    property int row;
}
property int col;
property int sub_col;
}

binding Jsystolic_array implements systolic_array{

void generate(){
    int counter, i, j;
    int M = (col*(col-1))>>1; //number of inner cells = m*(m-1)/2

    boundary bound_cell[col];
    inner inner_cell[M];
    splitter Splitter[col];
    joiner Joiner[col];

    counter = 0;
    for(i=0; i<col-1; i++){ // because of boundary cells,
        for(j=i; j<col-1; j++){
            inner_cell[counter].name = "inner"+counter;
            inner_cell[counter].row = row;
            inner_cell[counter].col = col;
            //number of rows
            inner_cell[counter].sub_col = sub_col;
            inner_cell[counter].x = j+1;
            inner_cell[counter].y = i;
            counter = counter+1;
        }
    }

    //job of joiners is done here to be able to use SR processors
    int tmp3 = sub_col*sub_col;
    int tmp4 = (sub_col*(sub_col+1))>>1;
    //properties of boundary cells
    for(i=0; i<col; i++){ //id's are given to the splitter and joiner cells
        int tmp2 = tmp3;
        int tmp1 = tmp4;
        bound_cell[i].row = row;
        bound_cell[i].sub_col = sub_col;
        bound_cell[i].last = false;
        bound_cell[i].name = "boundary"+i;
        Splitter[i].id = i;
        Joiner[i].name = "Joiner"+i;
        Splitter[i].name = "Splitter"+i;
        Splitter[i].sub_col = sub_col;
        //loop counts for joiners
        tmp2 = i*tmp2 + tmp1;
        Joiner[i].loop1 = tmp2; //own rows r value number
        tmp1 = i*sub_col;
        tmp1 = tmp1*(tmp1+1)>>1;
    }
}

Joiner[i].loop2 = tmp1; //number of the rest {r
values}
}

bound_cell[col-1].last = true;

//HORIZONTAL CHANNEL CONNECTIONS
counter = 0;
for(i=0; i<(col-1); i++){
    //channels from boundaries to inners
    channel ch1 = (bound_cell[i].b_out,
inner_cell[counter].from_left);
    for(j=i; j<(col-2); j++){
        channel ch = (inner_cell[counter].to_right,
inner_cell[counter+1].from_left);
        counter=counter+1;
    }
    //channels from inners to joiners
    channel ch2 = (inner_cell[counter].to_right,
Joiner[col-i-1].from_left);
    counter= counter+1;
    //channels between splitters
    channel ch3 = (Splitter[i+1].to_left,
Splitter[i].in);
}
channel b2j = (bound_cell[col-1].b_out,
Joiner[0].from_left);
channel in2splitter = {in, Splitter[col-1].in};

attribute Unused() on Splitter[0].to_left;

//VERTICAL CHANNEL CONNECTIONS
//the connections between splitters and cells
channel sp2b = {Splitter[0].to_down, bound_cell[0].b_in};
for(i=1; i<col; i++){
    channel sp2c = {Splitter[i].to_down, inner_cell[i-1].from_up};
}
//the connections between upper inner cell and boundary
cell
counter = 0;
for(i=1; i<col; i++){
    channel ch_i2b = {inner_cell[counter].to_down,
bound_cell[i].b_in};
    counter = counter+(col-i);
}
//the vertical connections between inner cells
counter=1;
for(i=0; i<(col-2); i++){
    for(j=i; j<(col-2); j++){
        channel ch_i2i = {inner_cell[counter].to_down,
inner_cell[counter+(col-2-i)].from_up};
        counter = counter+1;
    }
    counter = counter+1;
}
//the connections between joiners
attribute Unused() on Joiner[0].from_down;

for(i=0; i<col-1; i++){
channel j2j = {Joiner[i].j_out, Joiner[i+1].from_down};

channel j2j = {Joiner[col-1].j_out, out};

Codes of design file:

import givens_rotations.systolic_array;
import astruct.pcie.Vio;

interface Root{}

binding JRoot implements Root{
    Vio vio = {numSources = 1, numSinks = 1};
    systolic_array array = {row=400, col=10, sub_col=20};
    //multiplication of sub_col and col must be equal to row
    channel ch0 = {vio.out[0], array.in},
        ch1 = {array.out, vio.in[0]};

}

design partial_array {
    Root root;
}