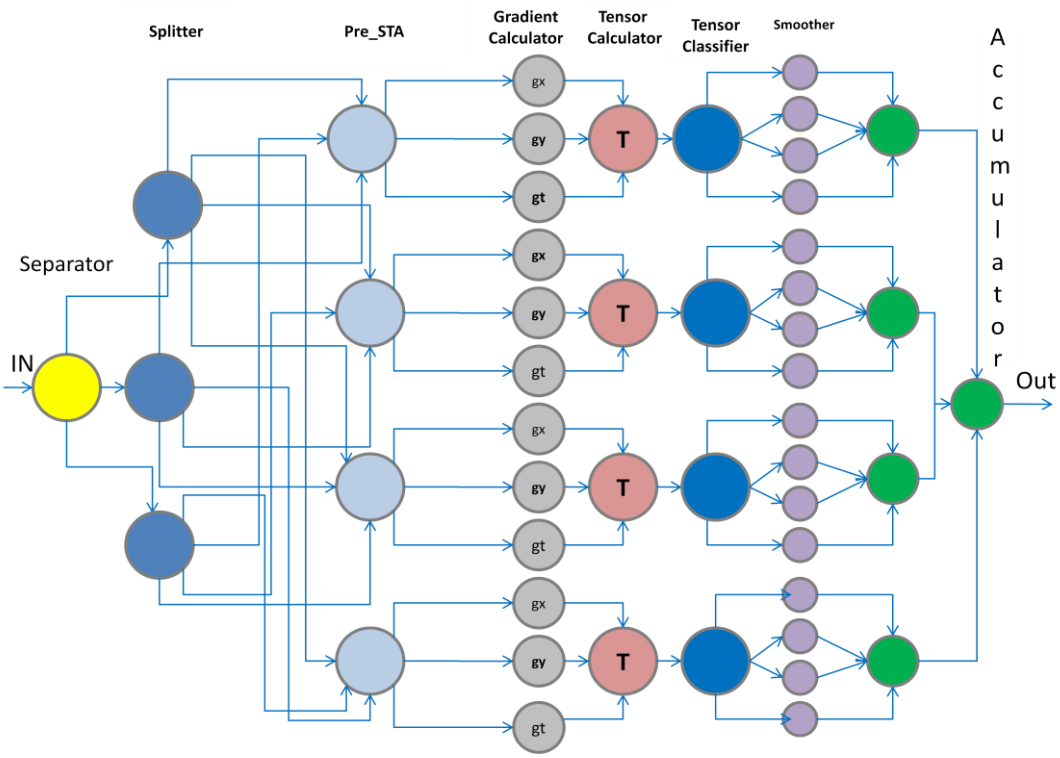
Technical report, IDE-1201, February 2012

Parallelization of the Estimation Algorithm of the 3D Structure Tensor

Master's Thesis in Embedded and Intelligent Systems

Ashraful Alam
asrala08@student.hh.se



School of Information Science, Computer and Electrical Engineering
Halmstad University

Parallelization of the Estimation Algorithm of the 3D Structure Tensor

Ashraful Alam
asrala08@student.hh.se

Project Report IDE-1201

Master's thesis in Embedded and Intelligent Systems, 30 credits

Supervisors:

Prof. Bertil Svensson
Dr. Zain-ul-Abdin
Dr. Stefan Karlsson

Examiner:

Prof. Tony Larsson

School of Information Science, Computer and Electrical Engineering
Halmstad University
Box 823, S-301 18 Halmstad, Sweden

February, 2012

Preface

This project is the concluding part of Master's in Embedded and Intelligent Systems with specialization in Embedded Systems at Halmstad University, Sweden. Firstly, I would like to thank Professor Bertil Svensson and Dr. Kenneth Nilsson for providing this project work and providing the materials about algorithm. Also, I would like to express my gratitude to my supervisors Dr. Stefan Karlsson for providing concrete ideas to understand the algorithm, and Dr. Zain ul Abdin for his nice comments and feedback about Ambric programming and design of the data flow diagram as well as supervision throughout the entire project.

Finally, I would like to thank my friends, who shared their ideas and my family for their support and faith in me throughout my whole life.

Ashraful Alam
Halmstad University, Sweden.

Abstract

This thesis work provides the implementation of 3D structure tensor on a Massively Parallel Processor Array (MPPA), Ambric 2045.

The 3D structure tensor algorithm is often used in image processing applications to compute the optical flow or to detect local 3D structures and their directions. The 3D structure tensor algorithm (3D-STA) consists of three main parts: gradient, tensor and smoothing. This algorithm is computationally expensive due to many multiplications and additions which are required to calculate the gradient (edge), the tensor and to smooth every pixel of the image. This is why this algorithm is very slow to run on a single processor. Therefore, it is important to make it parallel for high performance computation.

This thesis provides two parallel implementations of 3D-STA; namely coarse-grained parallelism and fine-grained parallelism. Ambric has 336 processors. Only 49 processors are used in coarse-grained implementation and 165 processors are used in fine-grained implementation. The performance of the two implementations is measured using a video stream input, consisting of a sequence of images of size 20x256x256. The performance of the coarse-grained parallelism implementation is 25 frames per second (fps) and the one of the fine-grained parallelism implementation is 100 fps. Thus the fine-grained version is four time faster than the coarse-grained one.

Additionally, the results are compared with the result of the Matlab implementation, running on Intel(R) Core 2 duo @2.10 GHz processor and also compared with another parallel optical flow implementation, in terms of speed and efficiency. The coarse-grained implementation is 58 times faster than the Matlab implementation and it achieves approximately half of the performance of the other parallel optical flow implementation. On the other hand, the fine-grained implementation is 230 times faster than the Matlab implementation and more than twice as (100/43) fast as the other parallel optical flow implementation.

These performance results are satisfactory and the results that our parallel implementations can be considered for real-time applications.

Table of Contents

1. Introduction 1
1.1 Motivation 1
1.2 Thesis Contribution 1
1.3 Outline of the thesis 2
2. Background 3
2.1 3D Sobel Operator 3
2.2 Structure tensor 4
2.3 3D Structure tensor algorithms 5
2.4 Data flow diagram of the Algorithm 6
2.4.1 Gradient 6
2.4.2 Tensor 9
2.4.3 Smoothing 10
3. Related work 13
4. Ambric Architecture 15
4.1 Introduction 15
4.2 Structural Object Programming Model 15
4.3 Ambric channel synchronization and communication 16
4.4 Chip Architecture 16
4.4.1 Ambric Compute Unit 17
4.4.2 RAM Unit 18
4.5 Brics and Interconnections 19
4.6 Processor Architecture 20
4.6.1 SR processor 20
4.6.2 SRD processor 21
4.7 Ambric Registers and Channels 23
4.7.1 Registers 23
4.7.2 Channels 23
4.8 Hardware Objects & Clocking 24
4.9 Performance measurement Tools 25
5. Implementation on Ambric hardware 27
5.1 Introduction 27
5.2 Design approaches for implementation 27
5.2.1 Coarse-grained implementation 28
5.2.2 Fine-grained implementation 29
5.3 Three stages of 3D-STA 31

5.3.1 The Gradient Calculation 31
5.3.2 Tensor block 33
5.3.3 Smoothing the tensor values 34
6. Result Analysis and Comparison 37
6.1 Result Analysis 37
6.2 Result Comparison 39
6.2.1 Comparison of the result for two design approaches and Matlab 39
6.2.2 Comparison of the results for two design approaches and Optical flow Implementation by Brigham Young University [4] 40
7. Conclusion and future work 43
7.1 Conclusions 43
7.2 Performance 43
7.3 Parallelism 44
7.4 Optimization 44
7.5 Future work 44
Bibliography 46

List of Figures:

Figure 2.1: 3D sobel operator 3
Figure 2.2: 3D sobel operator (t, y and x directions) 4
Figure 2.3: The 3D structure tensor algorithm takes a sequence of images as an input, processes it in several computational steps and finally outputs the structure tensor. 6
Figure 2.4: Gradient calculation in different direction (t, y and x) 7
Figure 2.5: Input Image 8
Figure 2.6: Output Image (Gradient) 8
Figure 2.7: Gradient calculation in x, y and t directions [15] 9
Figure 2.8: Tensor elements, corresponding one pixel nine tensor values 10
Figure 2.9: Smoothing using 8x8 square filter 10
Figure 2.10: Smoothed image becomes 32x32 using 8 by 8 filters 11
Figure 4.1: Structure Object Programming model 16
Figure 4.2: Ambric channel synchronization 16
Figure 4.3: Ambric chip 17
Figure 4.4: Compute Unit 18
Figure 4.5: Ram Unit 18
Figure 4.6: One Complete bric in the center, surrounded by parts of eight adjacent brics [3] 19
Figure 4.7: Processor Architecture [1] 20
Figure 4.8: Architecture of SR processors [3] 21
Figure 4.9: The SRD processor is a 32-bit streaming RISC processor with DSP extensions for math-intensive processing [3] 22
Figure 4.10: Ambric register 23
Figure 4.11: Hardware object with channel interfaces 24
Figure 4.12: Design of performance measurement in Ambric 25
Figure 5.1: Image divided for Ambric's particular processor 28
Figure 5.2: Data flow diagram of Coarse-grained implementation 29
Figure 5.3: Data flow diagram of Fine-grained implementation 30
Figure 5.4: Design of the Gradient calculation block on Ambric hardware 32
Figure 5.5: Part of codes to calculate gradient (gx) in the x direction 33
Figure 5.6: Design of the Tensor calculation block in Ambric 34
Figure 5.7: Part of tensor calculation program 34
Figure 5.8: Smoothing design block in Ambric implementation 35
Figure 5.9: Part of code for smoothing tensor 36

List of Tables:

Table 4.1: Comparison between SRD and SR processors 22
Table 6.1: Total number of processors used in coarse-grained implementation……..38
Table 6.2: Total number of processors used in fine-grained implementation 38
Table 6. 3: Performance results of coarse-grained implementation and fine-grained implementation 38
Table 6. 4: Execution time for gradient, tensor and smoothing..... 39
Table 6. 5: Result comparison among coarse-grained and fine-grained parallelism implementation and Matlab 40
Table 6. 6: Result comparison among coarse-grained, fine-grained and optical flow implementation by Brigham Young University [4] 41

Chapter 1

Introduction

1.1 Motivation

Many image processing applications use the 3D structure tensor algorithm (3D-STA), for example to compute the optical flow or to detect local 3D structures and their directions. Applications include autonomous systems, robotics and CT scans. Practically, this algorithm is difficult to apply in robotic vision because of its tremendously high computation demands caused by the frame rate requirements in real time applications. In most cases, traditional general purpose processors executing conventional sequential programs cannot compute 3D structure tensor in real time. So, the 3D-STA algorithm and many other image processing algorithms demand the use of high performance computing techniques (i.e., parallelization) to meet real-time requirements. Therefore, multi-core architectures preferably programmed in, dataflow languages, need to be used in order to meet the high performance demands. Different blocks of this algorithm are suitable to be implemented on multi-core architectures. Each block can be divided into smaller parts and each part can be mapped on a processor in pipelined fashion. Finally, the 3D structure tensor can be implemented using many interconnected cores which run in parallel.

This thesis provides 3D-STA implementation on Ambric 2045, which is called massively parallel processor array (MPPA) architecture. There are two design (coarse-grained and fine-grained) approaches are developed for this algorithm and implemented on the Ambric 2045. Then we have evaluated the results using the perfHarness library. The Ambric MPPA contains 336 programmable processors on a single die. These processors are general purpose and are programmed using Java or assembly language. The Ambric MPPA is interesting because it combines structural design with general-purpose programming. Programming the Ambric is easier; because of we write sequential programs for each of the processors.

In this project, we will briefly discuss the Ambric device and the implementation of 3D-STA on the Ambric MPPA. We will also compare the performance result of the Ambric implementation of 3D-STA against an implementation of optical flow on Ambric.

1.2 Thesis Contribution

The goal of this thesis work is to study 3D structure tensor algorithm and implement it on a specific multi core platform, called the Ambric Massively Parallel Processor Array (MPPA), using the languages aStruct and aJava (subset of Java). The goal has been achieved by first analyzing the algorithm and identifying its three computational stages, i.e., gradient calculation implemented using a 3D sobel operator, outer products (tensor) calculation, and

smoothing the tensor components using 8x8 square filters. We have presented two design approaches (coarse-grained and fine-grained) of implementations based on the granularity of parallelism. For coarse-grained and fine-grained implementation, we have split the image into four parts and sixteen parts respectively. Hence, the image is distributed over the several processors efficiently to calculate 3D-STA. Even each direction of gradient calculation is done using each of the Ambric processor. Then, we measured the performance of the two design approaches in Ambric hardware by using performance Harness Library which is provided by the Ambric department of Nethra Imaging. Furthermore, each stage of the algorithm execution time are measured using performance analysis tool which is attached aDesigner on the simulator.

Finally, the performance results are compared with the Matlab simulation and also compared with another optical flow implementation on Ambric MPPA [4].

1.3 Outline of the thesis

This thesis is organized as follows

Chapter 2 Background: this chapter describes the 3D structure tensor algorithm (3D-STA), including the 3D sobel operator for calculating edges of the images, the tensor calculation, and finally averaging over the components of the tensor values for removing the noise of the images.
Chapter 3 Related work: this chapter shortly discusses other optical flow implementations on Ambric and on FPGA.
Chapter 4 Ambric Architecture: this chapter introduces the Ambric Architecture and its performance analysis tools.
Chapter 5 Implementation: this chapter explains the problem to implement the 3D-STA algorithm, how to divide it into parts and different strategies for parallelization.
Chapter 6 Result analysis: this chapter presents the result analysis of the project and compares the results to related work.
Chapter 7 Conclusion and Future work: the final conclusions about the performance and efficiency of the Ambric architecture are given as well as several future works, which aim to enhance the 3D-STA.

Chapter 2

Background

This chapter gives a detailed explanation of the 3D sobel operator, 3D-STA and its three computational stages. The three stages are gradient calculation, outer products (tensor) calculation, and smoothing the tensor components corresponding to Sections 2.4.1, 2.4.2, and 2.4.3 respectively.

2.1 3D Sobel Operator

The Sobel operator is an edge detection operator. The 3D sobel operator kernel size is 3x3x3. The kernel is convolved over the image to calculate the edges of the image [9]. The 3D sobel edge detector is operating on three adjacent slices (a subset of the image solid). The results of edge detection are put in the central slice. Operators are shown in the Figure 2.1. It consists of 3 slices of matrix. In front, all the values are positive (i.e. [1, 2, 1], [2, 4, 2], [1, 2, 1]) , in the middle piece all the values are zeros (i.e. [0, 0, 0], [0, 0, 0], [0, 0, 0]) and in the last portion all the values are negative (i.e. [-1,-2,-1], [-2,-4,-2], [-1,-2,-1]). To calculate gradient on x, y and t direction, details of the sobel operator are shown in Figure 2.2 [6].

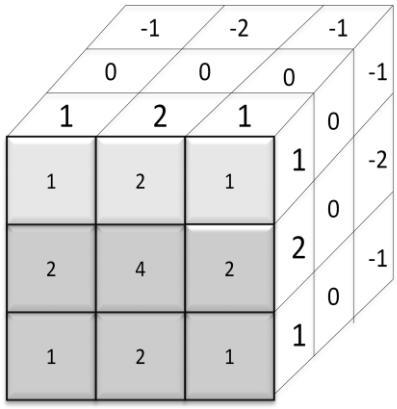


Figure 2.1: 3D sobel operator

Previous frame	Middle Frame	Afterward Frame
-1 -2 -1	0 0 0	1 2 1
-2 -4 -2	0 0 0	2 4 2
-1 -2 -1	0 0 0	1 2 1
t-1	t	t+1

Gradient on T Direction

1	2	1
0	0	0
-1	-2	-1

y-1

2	4	2
0	0	0
-2	-4	-2

y

1	2	1
0	0	0
-1	-2	-1

y+1

Gradient on Y Direction

-1	0	1
-2	0	2
-1	0	1

x-1

-2	0	2
-4	0	4
-2	0	2

x

-1	0	1
-2	0	2
-1	0	1

x+1

Gradient on X Direction

Figure 2.2: 3D sobel operator (t, y and x directions)

Advantages of the 3D sobel operator are given below-

- It provides not only the most 'correct' gradients but also the filtering properties of the kernel are very effective in reducing noise and aliasing artifacts.
- It is relatively easy to implement in hardware form, most obviously by a pipeline approach
- It is less sensitive to isolated high intensity point variations

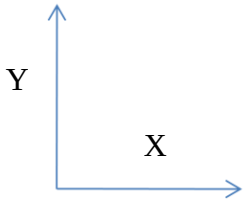
2.2 Structure tensor

Structure tensor play a fundamental role in today's image processing and computer vision, as it allows orientation estimation and image structure analysis such as corner detection, texture analysis, diffusion filtering and optical flow estimation [17].

2D structure tensor:

2D structure tensor (S) can be defined for a function f of two variables p=(x, y), where x, y spatial components. It is the 2×2 matrix as the product of the image derivatives by

$$S = \begin{bmatrix} \langle f_x f_x \rangle & \langle f_x f_y \rangle \\ \langle f_y f_x \rangle & \langle f_y f_y \rangle \end{bmatrix}$$

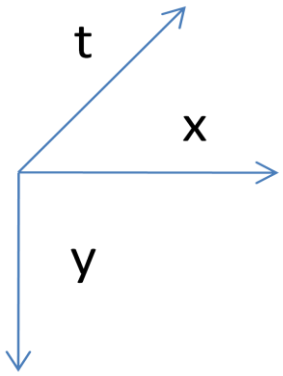


Where, f_x and f_y are the first order image intensity derivatives and <> represent the smoothing operation.

3D structure tensor:

The 3D structure tensor (S) can be defined for a function *f* of three variables *p*=(*x*, *y*, *t*), where x, y are spatial components and t is the temporal component. It is the 3×3 matrix as the product of the image derivatives by

$$S = \begin{bmatrix} \langle f_x f_x \rangle & \langle f_x f_y \rangle & \langle f_x f_z \rangle \\ \langle f_y f_x \rangle & \langle f_y f_y \rangle & \langle f_y f_z \rangle \\ \langle f_z f_x \rangle & \langle f_z f_y \rangle & \langle f_z f_z \rangle \end{bmatrix}$$



Where, the symbol <> represents the smoothing operation

2.3 3D Structure tensor algorithms

3D structure tensor algorithm (3D-STA) takes a sequence of images f(x) as an input and it can act as a volume of data where x, y and t are the 3D coordinates, x and y are the spatial components, and t is the temporal component [4,7,11]. The algorithm has several computational stages, shown in Figure 2.3.

The steps of the algorithm are given below-

1. Taking a sequence of images as an input
2. Calculating the gradient in 3D direction x, y and t using 3D sobel operator (mask size 3x3x3)
3. Calculating the tensor for every pixel in the images using gradient
4. Finally, smoothing each component of the tensor, to get the 3D structure tensor

2.4 Data flow diagram of the Algorithm

Figure 2.3 depicts the dataflow of this algorithm [4, 7]; the video input is represented as a sequence of images. For instance f is 3D image. Several stages of the algorithm are briefly described in the following subsections.

Input: sequence of images (for example: 20 images)

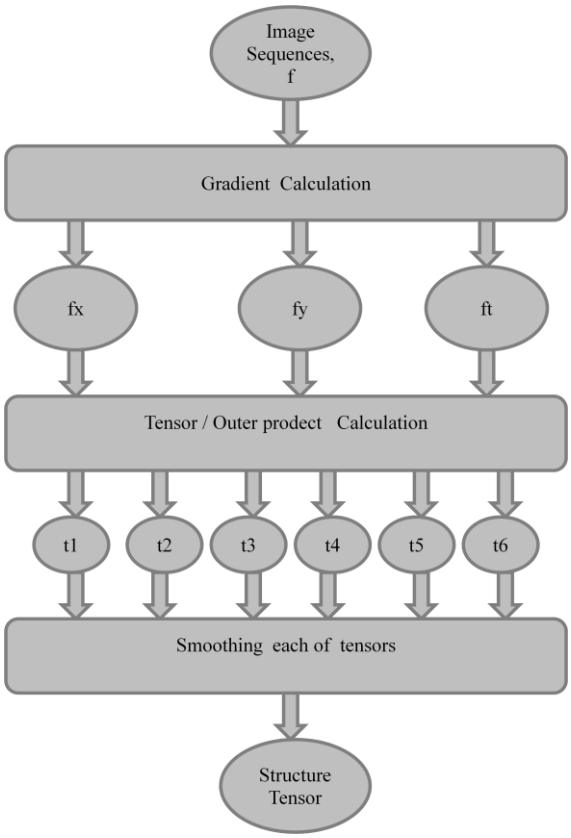


Figure 2.3: The 3D structure tensor algorithm takes a sequence of images as an input, processes it in several computational steps and finally outputs the structure tensor.

2.4.1 Gradient

In image processing the gradient of the image is one of the fundamental building blocks. It is a vector. In previous decades, many gradient operators were developed to calculate the gradient of the image such as Sobel operator, Roberts operator, Prewitt operator, Canny edge detector operator etc. The extraction of the gradient information from 3D objects plays an important role for many applications including object reconstruction and 3D medical imaging. The gradient information is used to identify strong features in the image, for instance edges or the local orientation of the curves and the surfaces. In image processing the extraction of the 3D gradient information is generally performed by filtering the input 3D data with filters that are typically implemented using 3×3×3 masks or higher ones [16]. In

this thesis we used 3D edge detector sobel operator (mask size 3x3x3) to calculate gradient (derivative) image sequence called gx, gy and gt for 3D images in three directions x, y and t, in Figure 2.4.

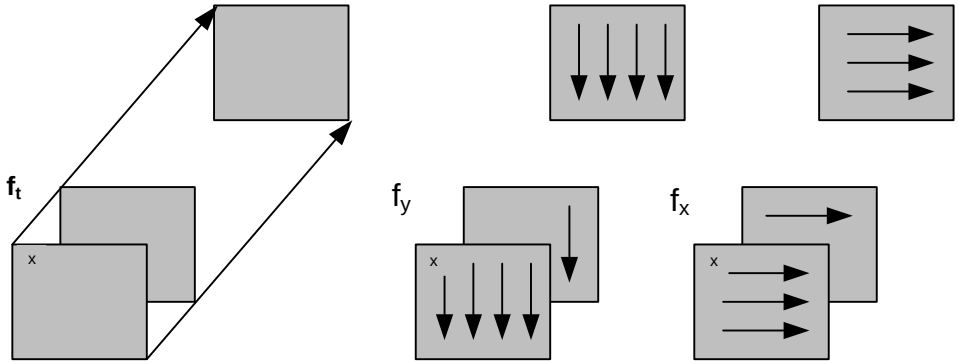
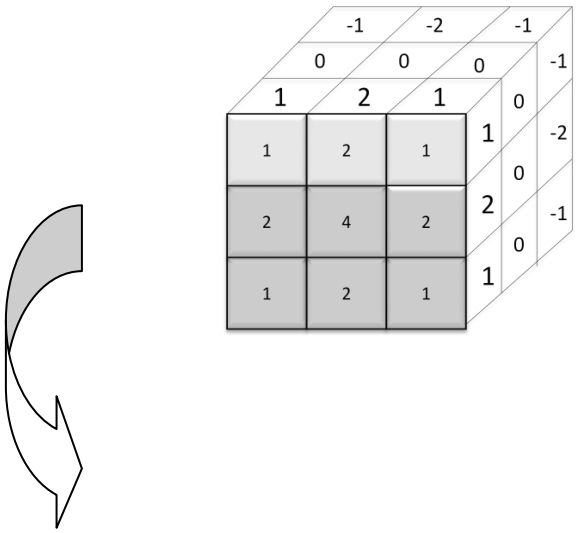


Figure 2.4: Gradient calculation in different direction (t, y and x)

The image will be convolved with sobel operator mask of size 3x3x3 and the center position will get new value in three directions (x, y, and t). This operator uses three different kernels to do the convolution for three frames along x, y and t directions to calculate the gradient in three directions. The operation will start at the three frames of input image in first, second and third rows convolved with sobel operators (using eq. 2, 3, 4) until end of the rows of input images, as shown in Figure 2.5. Then we will get gradients in x, y and t directions, as shown in Figure 2.6. After that, the procedure will be started again to calculate gradient second row, third row and 4th row of input images convolved with 3D sobel operators and will get three gradient values in each directions using eq-2, 3, and 4. Similarly, the next three rows will be convolved and it will continue until last row of the input images. Notice that the outer boundary of the image can't be computed by 3x3 masks. Outer boundaries of the images are usually blacked out but it can be calculated partially. In our work we used zero padding of the images boundaries to calculated borders of the images. For more information about gradient, reader can refer to [8, 15].



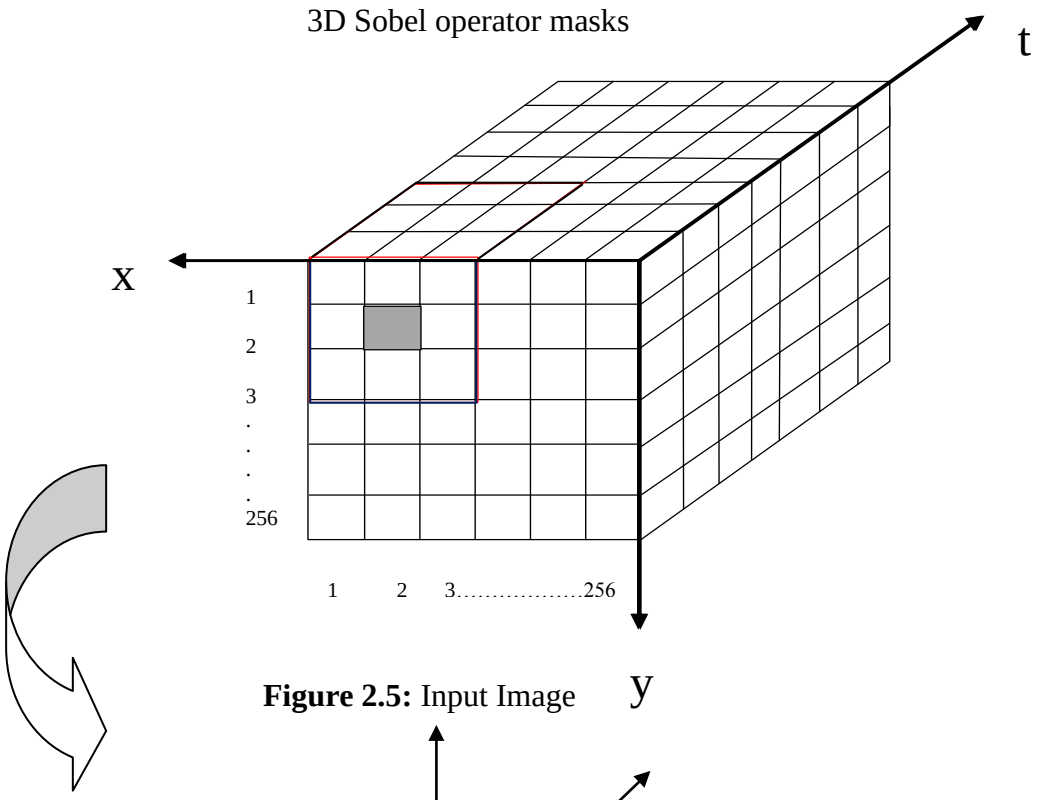


Figure 2.5: Input Image

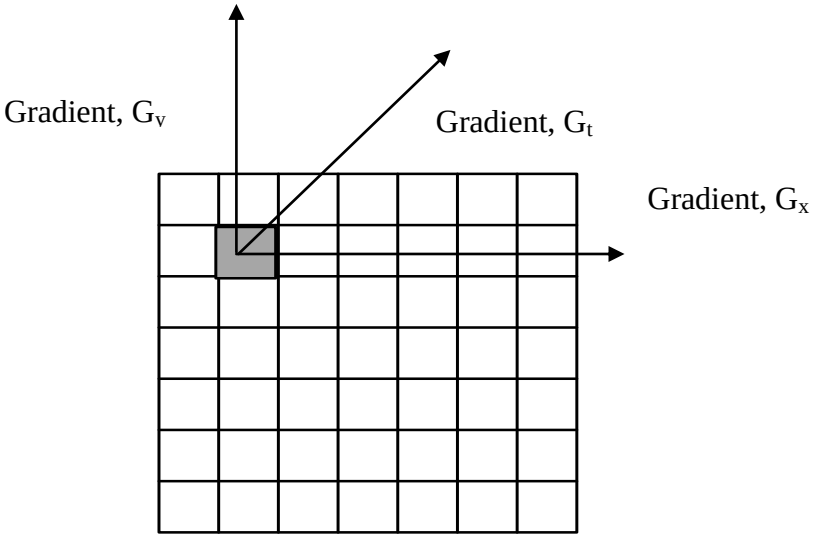


Figure 2.6: Output Image (Gradient)

$G_x(y,x,z)=$

```
Images(y+1,x+1,z+1)+2*Images(y,x+1,z+1)+Images(y-1,x+1,z+1)
+2*Images(y+1,x+1,z)+4*Images(y,x+1,z)+ 2*Images(y-1,x+1,z)
+Images(y+1,x+1,z-1)+2*Images(y,x+1,z-1)+Images(y-1,x+1,z-1)
-(Images(y+1,x-1,z+1)+2*Images(y,x-1,z+1)+Images(y-1,x-1,z+1)
+2*Images(y+1,x-1,z)+4*Images(y,x-1,z)+ 2*Images(y-1,x-1,z)
+Images(y+1,x-1,z-1)+2*Images(y,x-1,z-1)+Images(y-1,x-1,z-1));-----eq.1
```

$G_y(y,x,z)=$

```
Images(y+1,x+1,z+1)+2*Images(y+1,x+1,z)+Images(y+1,x+1,z-1)
+2*Images(y+1,x,z+1)+4*Images(y+1,x,z)+ 2*Images(y+1,x,z-1)
+Images(y+1,x-1,z+1)+2*Images(y+1,x-1,z)+Images(y+1,x-1,z-1)
-(Images(y-1,x+1,z+1)+2*Images(y-1,x+1,z)+Images(y-1,x+1,z-1)
+2*Images(y-1,x,z+1)+4*Images(y-1,x,z)+2*Images(y-1,x,z-1)
+Images(y-1,x-1,z+1)+2*Images(y-1,x-1,z)+Images(y-1,x-1,z-1));-----eq.2
```

$G_t(y,x,z)=$

```
Images(y+1,x+1,z+1)+2*Images(y,x+1,z+1)+Images(y-1,x+1,z+1)
+2*Images(y+1,x,z+1)+4*Images(y,x,z+1)+2*Images(y-1,x,z+1)
+Images(y+1,x-1,z+1)+2*Images(y,x-1,z+1)+Images(y-1,x-1,z+1)
-(Images(y+1,x+1,z-1)+2*Images(y,x+1,z-1)+Images(y-1,x+1,z-1)
+2*Images(y+1,x,z-1)+4*Images(y,x,z-1)+ 2*Images(y-1,x,z-1)
+Images(y+1,x-1,z-1)+2*Images(y,x-1,z-1)+Images(y-1,x-1,z-1));----eq.3
```

Figure 2.7: Gradient calculation in x, y and t directions [15]

2.4.2 Tensor

The structure tensor can be represented by symmetric 3×3 matrices which contain the structure information and intensity of the pixels. It has the 3D volumetric images and 3D spatio-temporal images respectively. This is the second stage of the algorithm and it is also called outer product calculation. The tensor calculation is done using the following equation which has been computed at each pixel of the target images. In this stage three gradient values are received as a row vector and transpose the row vector, get column vector. Then, we get tensor by multiplying row and column vector (see eq.4) [8]. In tensor calculation, we get nine values among them six values are independent and three values are repeated values. For detailed information reader can refer to [8, 15]

$$\mathsf{T} = (\nabla f)^T(\nabla f) = \begin{bmatrix}\partial_x f\\ \partial_y f\\ \partial_t f\end{bmatrix}\begin{bmatrix}\partial_x f & \partial_y f & \partial_t f\end{bmatrix} = \begin{bmatrix}\partial_x f\partial_x f & \partial_x f\partial_y f & \partial_x f\partial_t f\\ \partial_y f\partial_x f & \partial_y f\partial_y f & \partial_y f\partial_t f\\ \partial_t f\partial_x f & \partial_t f\partial_y f & \partial_t f\partial_t f\end{bmatrix} \quad \ldots\ldots\text{eq.4}$$

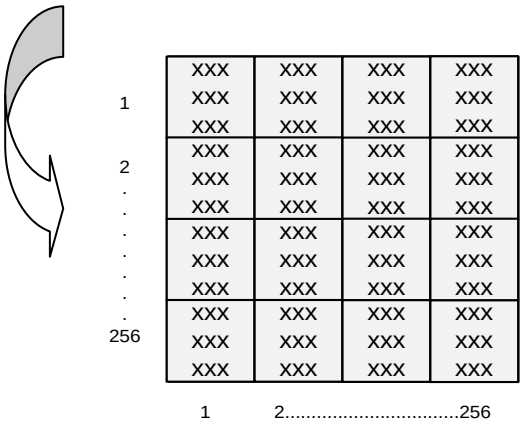


Figure 2.8: Tensor elements, corresponding one pixel nine tensor values

2.4.3 Smoothing

This is the final stage of the algorithm for smoothing each of the tensor value. We can easily obtain the structure tensor (S) from equation-4. We only need to smooth the components of tensor over the x, y and t axes. There are many algorithms to smooth the tensor values. In our implementation, we have used eight by eight square filters (8x8) to smooth the tensor values. The calculation begins from the first row and column and continues until eighth row and column (In Figure 2.9, red Colored Square). We calculate the summation of all those values. After summing all the values, we have got one smoothed value, S. After that, similar calculation will start from the ninth row and column which continues up to the sixteenth row and column. We added all of the values and then got second smoothed value. Similar processes will be continued until 256th row and column. Finally, we will get smoothed image for 32x32 showed in Figure 2.10. For more detailed information see reference [15].

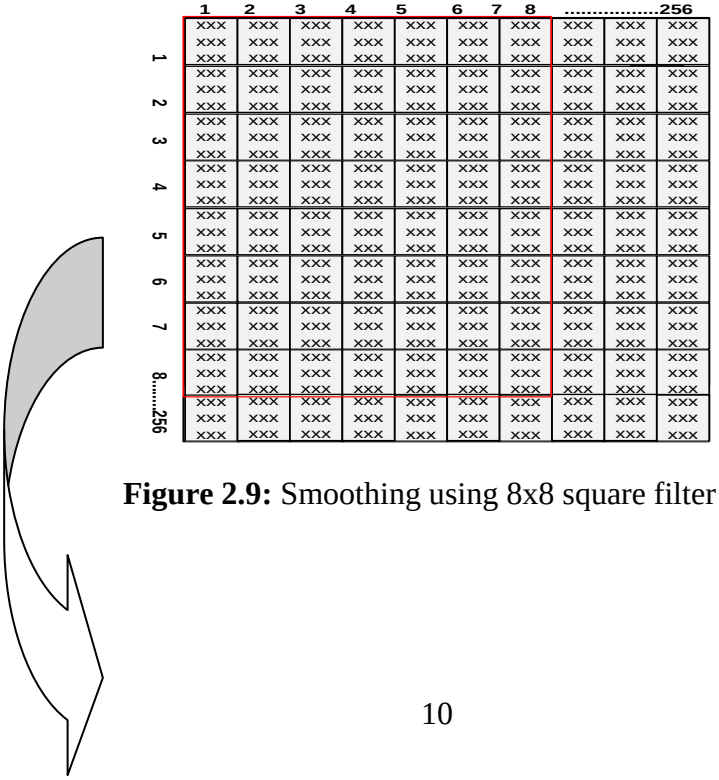


Figure 2.9: Smoothing using 8x8 square filter

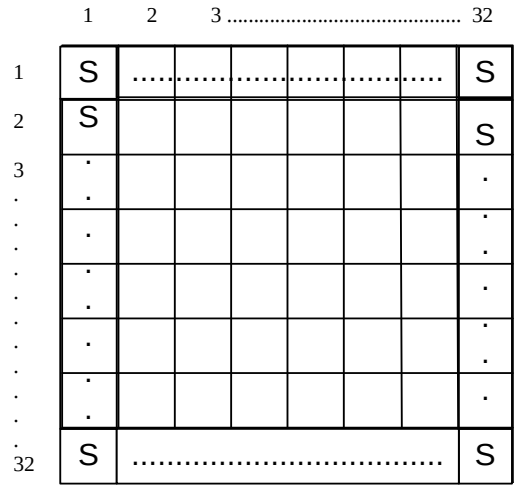


Figure 2.10: Smoothed image becomes 32x32 using 8 by 8 filters

$$S = \begin{bmatrix} \langle f_x f_x \rangle & \langle f_x f_y \rangle & \langle f_x f_z \rangle \\ \langle f_y f_x \rangle & \langle f_y f_y \rangle & \langle f_y f_z \rangle \\ \langle f_z f_x \rangle & \langle f_z f_y \rangle & \langle f_z f_z \rangle \end{bmatrix}$$

Where, the symbol <> represents the smoothing operation.

Chapter 3

Related work

Over the past two decades, tensor based optical flow has been implemented in different parallel processing environment such as FPGA, GPU etc. As far as optical flow implementation on Ambric environment is concerned, only one example has been found [4]. The aim of the optical flow is to compute a velocity vector. That vector indicates the pixels' apparent motion in the x and y directions over the past few image frames.

In [4], the algorithm was implemented as a multi-stage pipeline. The first stage is called Gradient block, also known as the derivative block in ambric design. The derivative block is to accept a sequence of images and then perform three convolutions. One convolution is temporal (the dt calculation) and is computed across five images using five-elements masks. The other two convolutions are spatial in one dimension (dx and dy), operate on only one image at a time and use five-element masks (mask size is: [-1 8 0 -8 1]). The three different derivative calculations compute results using differing numbers of images (only one image requires performing spatial convolutions and five images require a temporal convolution). So, the derivative calculation is done using 1D convolution in the x, y and t direction to produce gradient image sequence gx, gy and gt. The second stage (smoothing), smoothed the gradient values to produce gxs, gys and gts and get better derivative using 3D convolution. The 3D convolution performed using 3x1 convolutions in t direction, 5x1 in x direction and 1x5 in y direction. The third stage, tensor calculation (Outer Product) is point to point multiplication between previously computed and smoothed gxs, gys and gts images. A 2D smoothing step is then performed in each of the six succeeding smooth _ blocks (each does a 3x1 followed by a 1x3 convolution), creates xxs; yys; tts; xys; yts, and xts images. After that calculate the velocity vector for each pixel as:

$$V_x = \frac{(xxs \times yts - xys \times xts)}{(xxs \times yys - xys^2)}$$

$$V_y = \frac{(yys \times xts - xys \times yts)}{(xxs \times yys - xys^2)}$$

Finally, the results of the velocity calculation, V_x and V_y, are spatially smoothed with the kernels 7x1 and 1x7.
The performance result, shows that their implementation can processes approximately 37 frames per second (FPS) on an image stream consisting of 320x240 images. According to this performance result, it can process 43 frames per second consisting of 256x256 images.

In [7, 11, and 13], FPGA implementation is a five-stage computation. The first stage performs 1D convolution in the x, y and t dimensions using five-element masks. The convolutions results are gradient images gx, gy and gt. Gradient Weighting is the second

processing stage, it performs seven-element row and column convolutions in succession on each of *gx*, *gy*, and *gt* to produce weighted gradient images *gx*, *gy*, and *gt*. The third stage (Outer Product), produces five new matrices by multiplying various combinations of the weighted gradient matrices together element-wise (o1 = gx × gx, o2 = gy × gy, o4 = gx × gy, o5 = gx × gt, and o6 = gy × gt). Tensor calculation is the fourth processing stage, and performs a three-element row convolution on each of these matrices to produce five tensor matrices (t1, t2, t3, t5, t6). The final stage is optical flow calculation, which computes the velocity flow fields V_x and V_y using the formula:

$$V_x = \frac{(t_6 t_4 - t_5 t_2)}{(t_1 t_2 - {t_4}^2)}$$

$$V_y = \frac{(t_5 t_4 - t_6 t_1)}{(t_1 t_2 - {t_4}^2)}$$

It can process 640 × 480 images at 64 frames per second.

In our work, which will be described in details in chapter 5, we implemented 3D structure tensor that is used in optical flow. A 3D structure tensor has three computation stages. The first stage of the algorithm is gradient calculation, it takes three images, using 3D sobel operator kernel to produce derivative image sequence called gx, gy and gt. After that, outer product calculation is done point to point multiplication using gradient image sequence gx, gy and gt. Finally, we smoothed the tensor values using 8x8 square filters. In our implementation we used 3D sobel operator for this algorithm to calculate appropriate gradient. The performance result shows that our implementation can process 100 frames per second image size 256x256.

Chapter 4

Ambric Architecture

4.1 Introduction

Ambric is a device which consists of an array of 336 32-bit RISC (Reduced Instruction Set Computing) processors and 336 2-KB memories, which run up to 300 MHz. It is most applicable for embedded systems such as medical, imaging, video and signal processing. The Ambric chip (Am2045) has around 180 million silicon transistors. Ambric chip contains registers that are 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. It is massively parallel distributed memory multiprocessor. It is based on the structural object programming model, where objects are strictly encapsulated software programs. Processors are general purpose and are programmed using Java and aStruct or assembly language. They run concurrently on an asynchronous array of processors and memories.

4.2 Structural Object Programming Model

Ambric chip has hundreds of 32-bit RISC processors and all are programmed by a conventional sequential code. It has also hundreds of distributed memories. Its circuit-switched interconnect network joins CPUs, memories dedicated hardware for each channel and register at each stage. All of the processors are called object and the smallest programmed object is called the leaf object. The leaf objects can be connected to each other by channels. Objects are running concurrently on an array of Ambric processors and memories. Basically, the objects run independently with their own speed without having any side effect on the others because they have no shared memory. Figure 4.1 shows, the structure of object programming model [2].

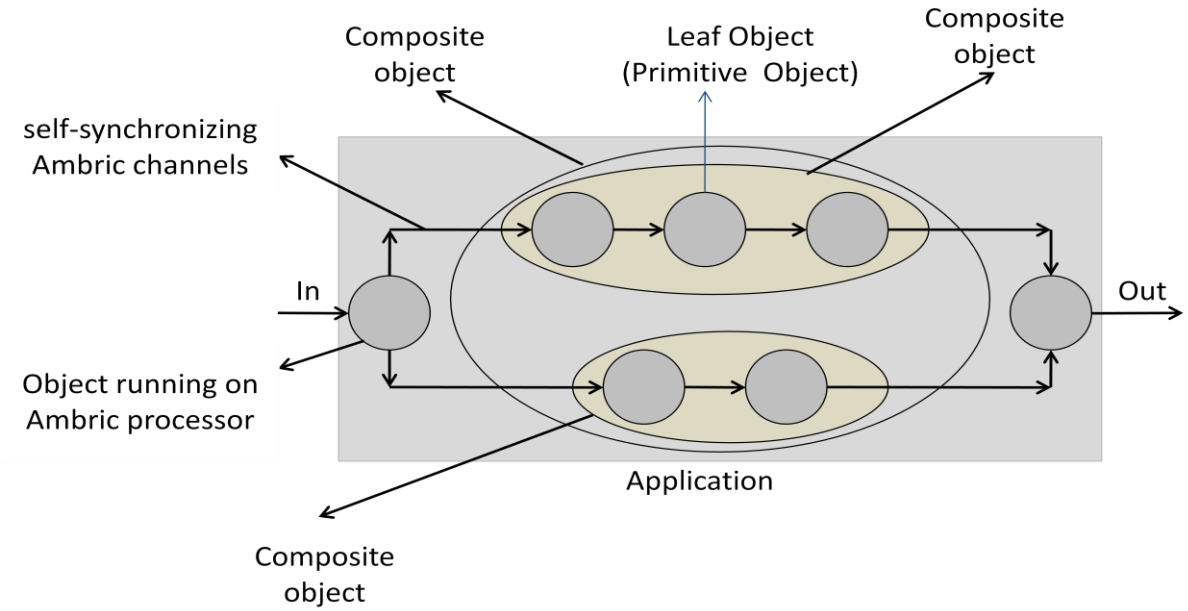


Figure 4.1: Structure Object Programming model

4.3 Ambric channel synchronization and communication

Ambric has direct CPU to CPU communication, it is fast and efficient. Processor to processor communicates (data sending and receiving) on a self-synchronizing Ambric channel. The communication process is a simple read or writes to a processor register. The channel synchronizes its objects at each end, dynamically as needed at run time, not at compile time. Channels give explicit synchronization in addition to communication, when the channel is ready after that the CPU only sends data, otherwise it just stalls. Similarly, when the channel has data then the CPU only receives; else it just stalls, as shown in Figure 4.2. For detailed information [2]

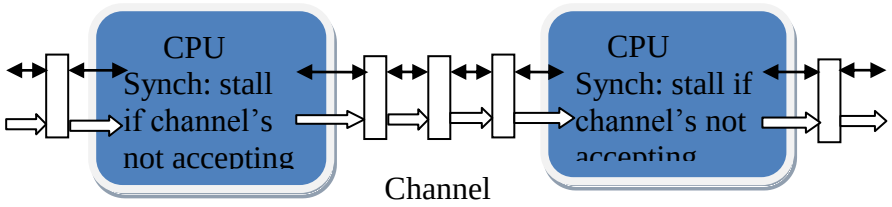


Figure 4.2: Ambric channel synchronization

4.4 Chip Architecture

Am2045 is a standard-cell 130nm ASIC with 117 million transistors. The Ambric chip consists of five-by-nine array of brics, shown in Figure 4.3. The brics contain up to 336 32-bit processors and up to 7.2 Mbits of distributed SRAM that run at up to 300 MHz. The chip has four GPIOs (General Purpose Input and Output) for reading and writing input and output

streams. Two DDR2 SDRAM (double-data rate 2 synchronous dynamic RAM) interfaces to external memories, a Flash interface for the run-time configuration, a PCIe (Peripheral Component Interconnect express) supporting initial chip configuration from the host or PC. The Ambric chip consists of two types of processors, one is Streaming RISC with DSP extension (SRD) and another one is Streaming RISC (SR) [1].

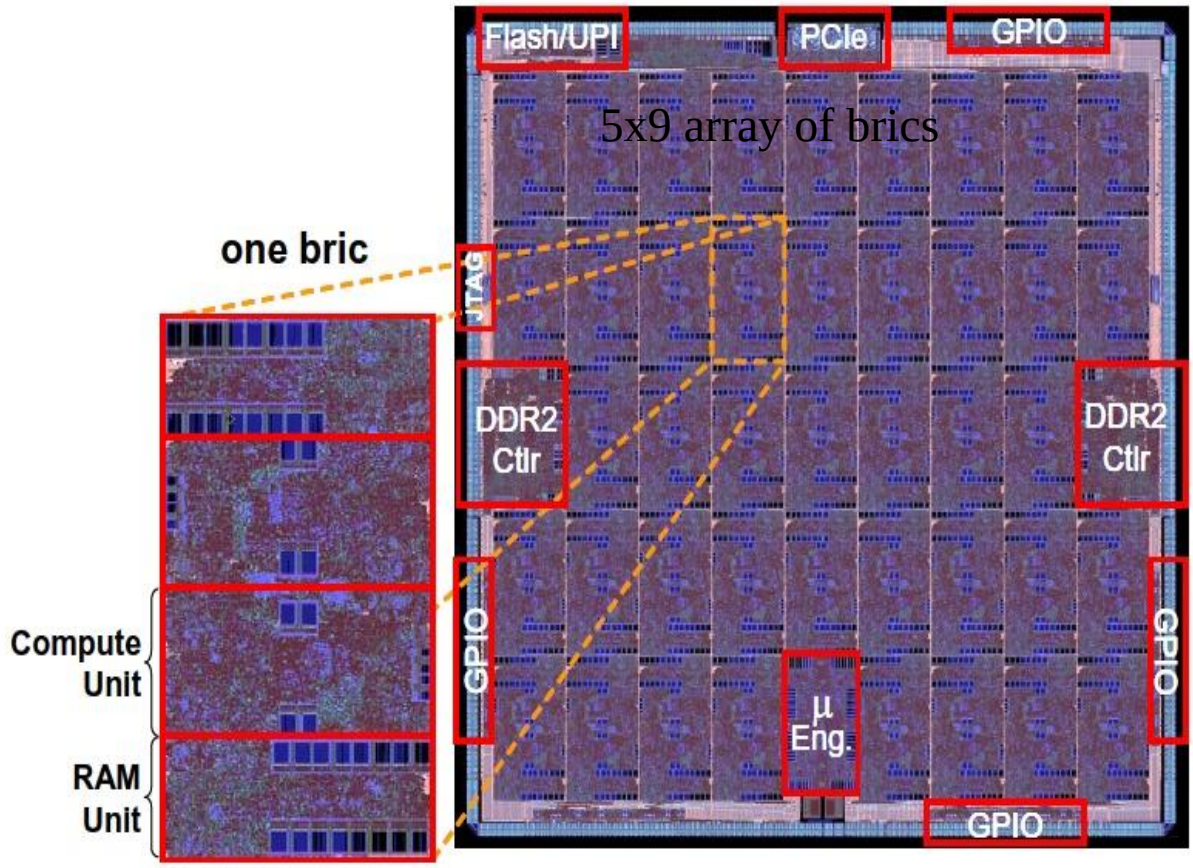


Figure 4.3: Ambric chip

4.4.1 Ambric Compute Unit

A cluster of four Ambric processors (two SRs and two SRDs) are represent a Compute Unit (CU) in ambric, as shown in Figure 4.4. This CU is paired with a RAM Unit (RU) with configurable access engines. CU is interconnected with six input multiplexers and one output crossbar.

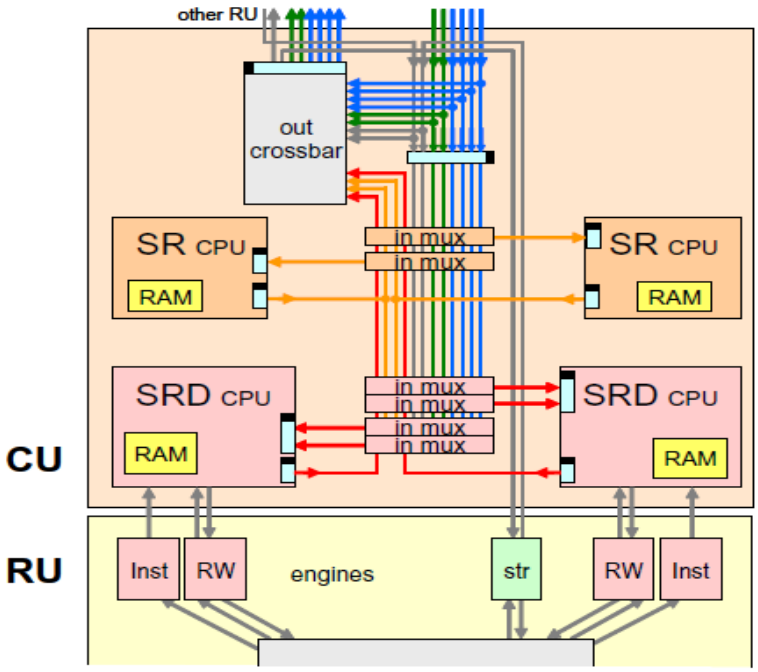


Figure 4.4: Compute Unit

4.4.2 RAM Unit

In ambric, the RAM (Random Access Memory) units are the main on-chip memory facilities. Each RU has four independent single-port RAM banks, as shown in Figure 4.5, each with 512 words. It has six configurable memory access engines that turn RAM regions into objects that stream addresses and data over channels.

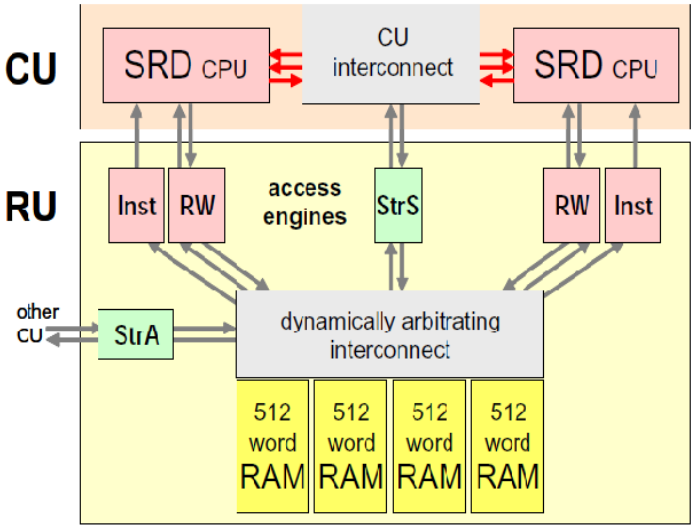


Figure 4.5: Ram Unit

There are two RW for SRD data by random access or FIFOs, two Inst for SRD instructions by random access or FIFOs and two Str for channel-connected FIFOs, or random access over channels using packets. RAM banks are accessed by the engines on demand; through a dynamic, arbitrating channel interconnect. To get one word per cycle FIFO bandwidth from single-port RAMs, the engines can be configured to use two banks, striping even addresses to one and odds to the other. Channel self-synchronization has naturally organized their accesses into full bandwidth for each. Engines can interconnect through RU neighbor channels to form multiple RU FIFOs [1].

4.5 Brics and Interconnections

The block diagram, in Figure 4.6, shows one complete bric in the center, surrounded by parts of eight adjacent brics. Each bric has two CUs and two RUs, totaling 8 processors and 21K bytes of RAM in Figure 4.6. When two compute units and two ram units come together, they create a bric. Brics have their own circuit-switched interconnection network with configurable switches. This network always runs at the highest clock rate and connects to the compute units through clock crossing registers. In total, there are 45 brics on the AM2000 family Ambric chip [1].

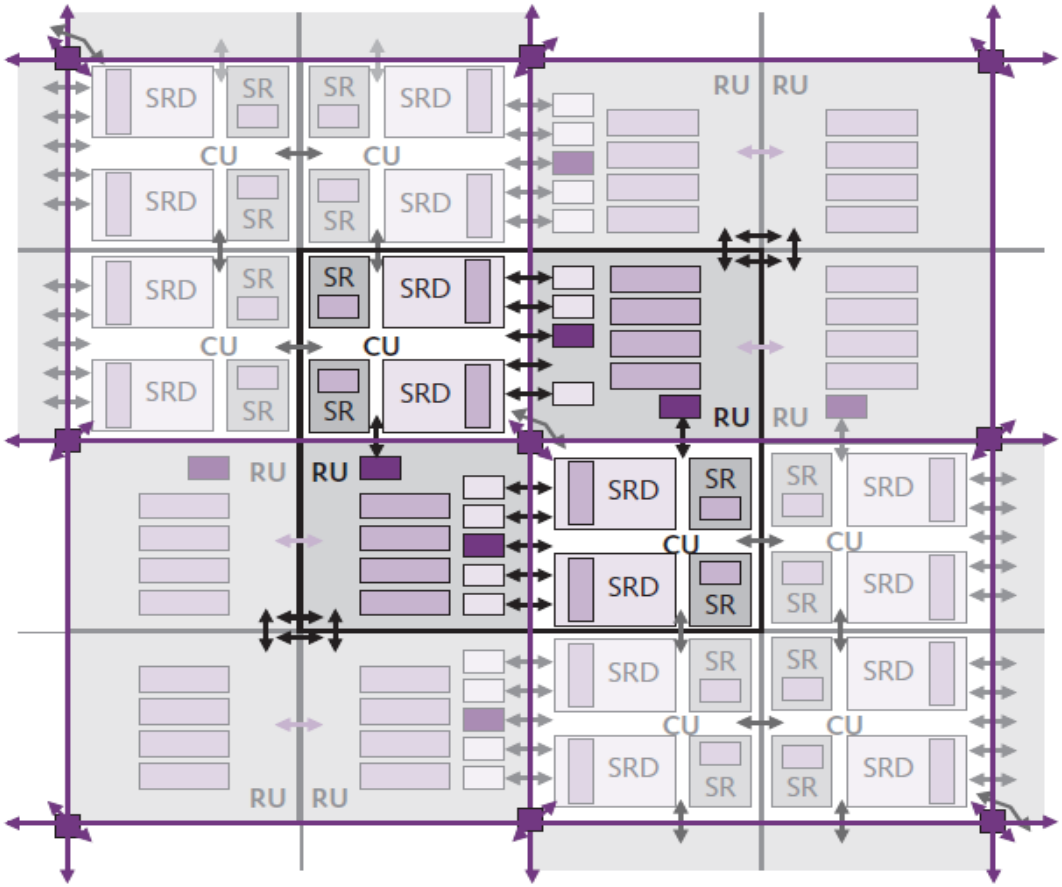


Figure 4.6: One Complete bric in the center, surrounded by parts of eight adjacent brics [3]

4.6 Processor Architecture

Figure 4.7 shows an abstract view of the ambric processor architecture. The purpose of this architecture is to process data and control from channels. In the Ambric chip, there are two types of processors, one is SRD (Streaming RISC with DSP extensions) processor and other one is SR (Streaming RISC) processor. These processors have local RAM memories and they get the instructions through channel. Processors execute instructions streamed from local memory by using random access channels. In Ambric processors there is no interrupts each other since each processor is dedicated to own task.

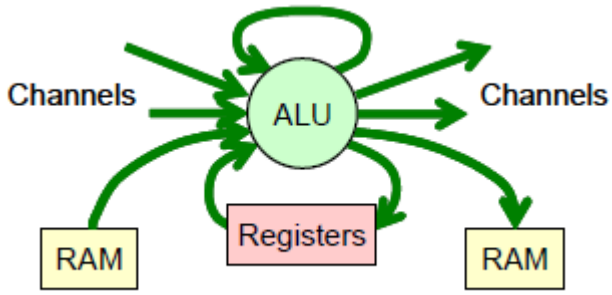


Figure 4.7: Processor Architecture [1]

4.6.1 SR processor

SR is a simple 32-bit streaming RISC CPU, used mainly for small or fast tasks, such as forking and joining channels, generating complex address streams, and other utilities. Figure 4.8, shows the SR processor's block diagram. It has a single-integer ALU with eight general registers, one input channel connected to the CU input crossbar, and one output channel connected to the CU output crossbar. 64 words of parity-checked local memory hold up to 128 16-bit instructions and data. SR's data path is an Ambric channel, with Ambric registers on the ALU inputs and outputs, and its control path is a channel, because the Instruction Register (IR) and Program Counter
(PC) are Ambric registers.

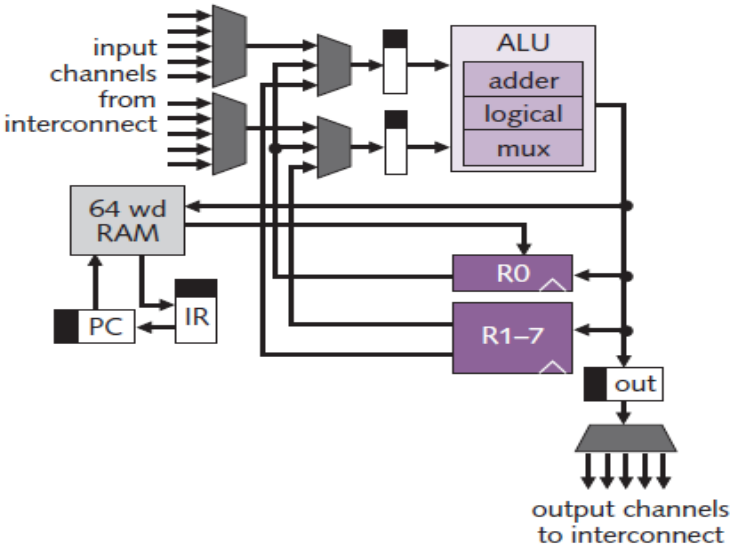


Figure 4.8: Architecture of SR processors [3]

4.6.2 SRD processor

The architectures of SR and SRD processors are very similar. Nevertheless, SRD processor is an expansion of SR processor. It is more powerful than an SR core. It has 32-bit-long instructions and three 32-bit ALUs—two in series, one in parallel—allowing two-way, or even three-way, instruction-level parallelism. Each of the two series ALUs can execute one 32-bit operation, two 16-bit operations, or four 8-bit operations per clock cycle. The third ALU is pipelined for rapidly executing multiply-accumulate (MAC) and sum of absolute differences (SAD) instructions. This ALU can execute one 32-×8-bit operation or two 16-×8-bit operations per cycle. The SRD core has 18 general-purpose 32-bit registers and a 64-bit accumulator, plus 1KB of local memory for caching instructions or data. A block diagram of an SRD processor is shown in Figure 4.9 [1].

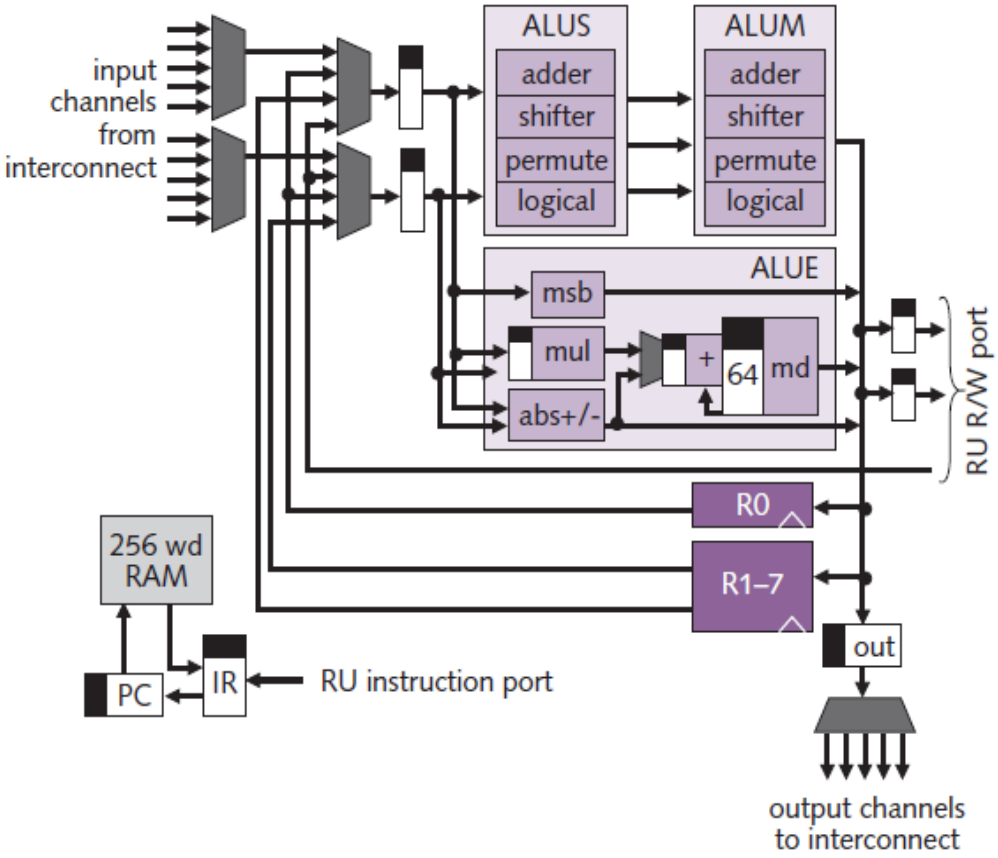


Figure 4.9: The SRD processor is a 32-bit streaming RISC processor with DSP extensions for math-intensive processing [3]

The differences between SR and SRD processors:

SR processor	SRD processor
32 bits CPU streaming RISC with DSP extensions	32 bits CPU streaming RISC
It has one ALU	It has three ALUs, two in series and one is parallel
64 (32 bits) words local RAM	256 word local RAM
Accepts 1 input and feeds 1 output each cycle	Accepts inputs from 2 channels and feeds an output to a channel each cycle
16 bits wide instruction	32 bits wide instruction

Table 4.1: Comparison between SRD and SR processors

4.7 Ambric Registers and Channels

4.7.1 Registers

Ambric registers are the heart of the Ambric hardware that makes these programming models. Basically, Ambric architecture and implementation is based on its capabilities. Commonly used edge-triggered registers are replaced everywhere in the system but it is still a clocked register with data in and data out. There are two control signals, one is ***valid*** and the other one is ***accept***, that apply a hardware protocol for local forward and back-pressure. This protocol makes the systems which are self-synchronizing and asynchronous by using the Ambric register. Every Ambric register is preemptive as a result its operation is self–initiated. Figure 4.10 shows the register input and output procedures, when a register can accept an input, it asserts it is ***accept*** signal upstream; when it has output available, it asserts ***valid*** downstream. After that when they see both valid and accept are true, they separately know the transfer has occurred, devoid of negotiation or acknowledgement. Therefore, this protocol is simple, working using only two signals without intermediate logic. It is not like FIFOs, these dynamics are self-contained, and as a result they become hidden and abstract.

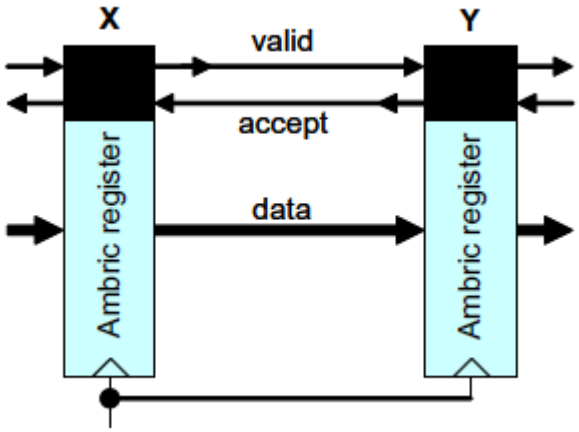


Figure 4.10: Ambric register

4.7.2 Channels

Ambric channels are the important part of the Ambric hardware. A chain of registers like Figure 4.10 is called an Ambric channel (or just a channel). Channels are the fully encapsulated, fully scalable technology for passing control and data between objects called for by the programming model.

A channel is word-wide, unidirectional, point-to-point, and strictly ordered (first-in-first-out). Local synchronization at each stage lets channels dynamically accommodate varying delays and workloads on their own. Changing the length of a channel does not affect functionality,

only latency changes. It is a simple and universal abstraction that combines communication and synchronization.

Ambric channels are used throughout the chip architecture, inside processors and memories as well as throughout the interconnection. The two words of buffering in each stage of a channel act as a small FIFO to smooth out dynamic behavior between objects. Larger FIFOs, implemented in RAMs, can be inserted transparently in channels for larger flow control, but also to implement storage in the application. FIFOs often replace random-access memories in Ambric applications, because they are a streaming, self-synchronizing form of storage. Ambric channels are used throughout the chip, inside the processors, memories and interconnections.

4.8 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. By connecting objects, composite objects and complete applications are obtained. These objects are connected by Ambric channels, by sending data; the synchronization between objects is acquired. Since the objects run independently and they are synchronized only through the channel interfaces, this system is called globally asynchronous and locally synchronous (GALS).

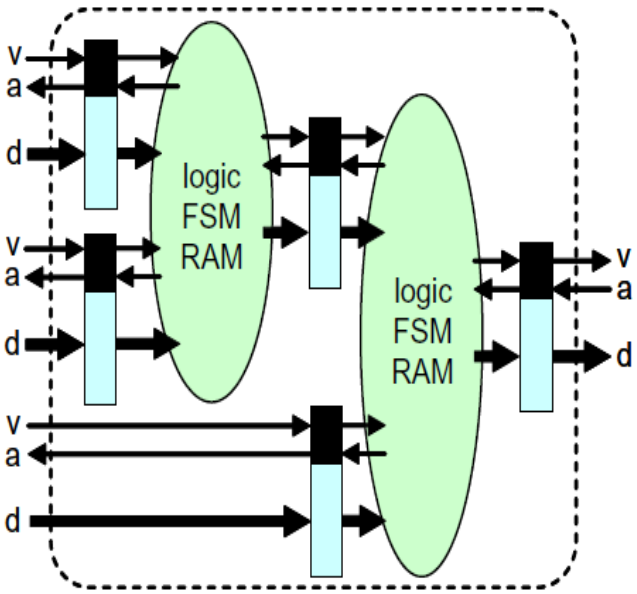


Figure 4.11: Hardware object with channel interfaces

The synchronization protocol enables clock crossing registers. A clock crossing register works with different, but related, clocks on input and output. So 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 if the receiver processor (*accept* signal) is ready to get it. If the receiver processor 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 [1].

4.9 Performance measurement Tools

Performance tool (performance Harness Library) has been provided by the Ambric department of Nethra Imaging, released in July, 2010. It is still under development. By using these tools we found the performance in terms of latency and cycle count per output sample for different design of 3D-STA. There are 2 types of input preloader used in this library. One is FIFO preloadar using for small amount of data and another one called DDR loader used for large number of input data. This perfHarness has four channels, two of them input and two of them output channels. In this test, the DDR Loader Object reads the input samples and stores in DDR memory from input file. Then the PerfHarness object streams the input samples from the DDR memory and calculates the performance. After that in the next stage the full program will be loaded in to the dut Object. After completing algorithm's execution then again the output will be sent to the PerfHarness Object where it finds the latency and cycle count per output sample as shown in Figure 4.12

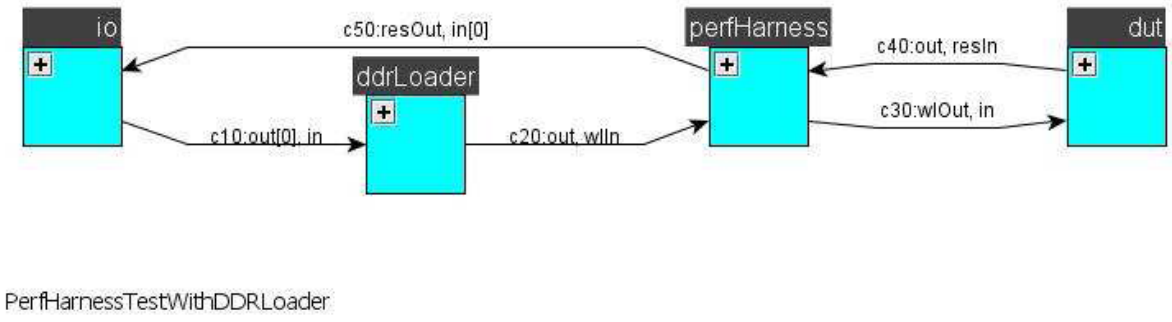


Figure 4.12: Design of performance measurement in Ambric

The PerfHarness object has 4 channels, 2 inputs and 2 outputs. One input and one output channel is connected to the system which is under measurement. The second input channel is used for receiving the input data. The second output channel is used to write out the chosen measurement results, in Figure 4.12.

The performance analysis (PerfHarness) library is developed to analyze designs. This performance analysis works, the input words and output words are divided into pieces of workloads. For example: the input of 1000 words can be divided into 10 pieces of workloads of 100 words. For more information see [14].

Chapter 5

Implementation on Ambric hardware

5.1 Introduction

This thesis work is an implementation of 3D-STA on the high performance Ambric Massively Parallel Processor Array (MPPA). In this chapter we describe implementation strategies of this algorithm which is described in chapter 2. Two design strategies are implemented on the Ambric hardware. In the first design technique, each image is split into four parts. In the second design, each image is divided into sixteen parts. We have demonstrated and analyzed the performance of these two approaches using 20 images of size 256x256. Since we are using 3D soble operator we have to deal with the apron problem when splitting the image. Apron pixels are the extra pixels that need to be added in order to calculate the gradient in the border of the image. To prevent apron problem after the image is split, the splitting parts has to share one extra column and row to calculate gradient of their boundary pixels.

5.2 Design approaches for implementation

Sobel operator need apron pixels, usually zero-padding, around the image in order to convolve pixels along the edge of the image. The implementation of this algorithm on Ambric's core is done by splitting the image into small parts, and then assigns these small parts to different processors. The splitting of the image will produce ***additional edges*** which require additional apron pixels. These small parts of the image need to share one extra column and row as apron pixels along the ***additional edges***, otherwise the computation will fail to calculate boundary pixels of the small part of the image, as shown in Figure 5.1 [9].

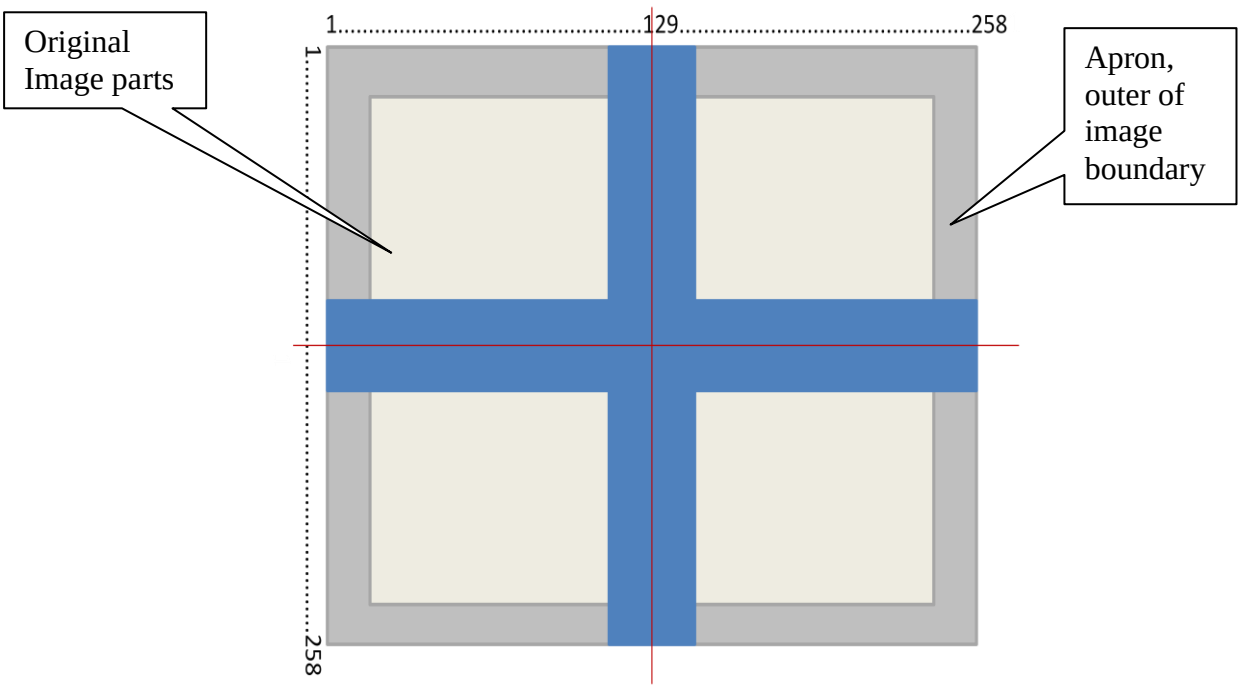


Figure 5.1: Image divided for Ambric's particular processor

Figure 5.1 depicts apron pixels along the ***additional edges***, in this design the image is divided into four parts. The gray color represents the apron of the image, the blue color represents shared border of the image and the white color is the original image. The image size is 256x256 with apron 258x258. When we divide the image into four parts then each part becomes a 129x129 and for calculating the border of the image's part added one more pixel from the other part of the image's portion. So, each divided part becomes a size130x130.

To prepare an input data we rearrange the image into a set of blocks of size 3x3. Then, three blocks from three consecutive images will be mixed to create a block of size 3x3x3, which is equal to the kernel size of 3D soble operator. The reason for mixing the data of three consecutive images is because of ambric PerfHarness library (to measure the performance) support only one input file and one output file. But in the implementation, in both approaches, we have introduced a processor named *separator* to separate the mixed data of the three images. Now we brief the two implementation approaches based on the granularity of parallelism.

5.2.1 Coarse-grained implementation

In this approach, first the input data is separated into three different images, and then each image is split into four parts using three processors called *splitter*. After splitting, four *pre-STA* processors collect the same part from the three images to prepared data for pipelined 3D-STA implementation. In total, this approach uses 17 SR and 32 SRD

processors. The SR processor is used to distribute, to collect data, while the SRD processors are used to compute the three stages of 3D-STA. Half of the SRD processors are used for smoothing, 12 for gradient calculation and 4 for tensor calculation. Figure 5.2 shows the mapping of this approach on ambric platform. The circles represent processors and arrows represent the flow of data.

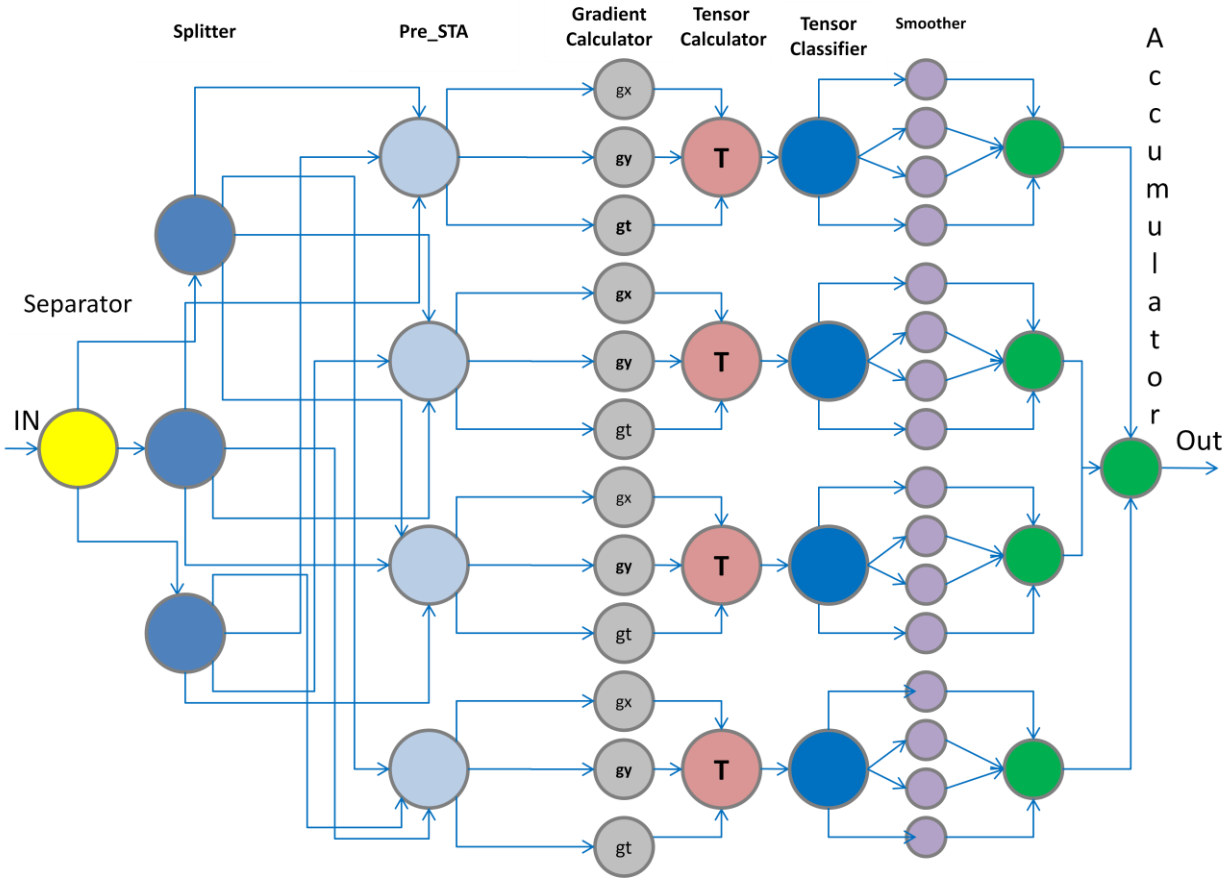


Figure 5.2: Data flow diagram of Coarse-grained implementation

So in this design, a total of forty nine processors are used out of 336 processors to calculate all of the parts of algorithm.

5.2.2 Fine-grained implementation

The procedure of this approach is the same as the coarse-grained implementation a greater number of processors. Here, the image is split into sixteen parts by two splitting stages (*splitter_1* and *splitter_2*). These sixteen parts of the image will be arranged for the three stages of 3D-STA by using 16 *pre_STA* processors. This approach uses 69 SR and 96 SRD processors. In this design all of the SR processors are also used to distribute and to collect data while SRD processors are mainly used for calculating all parts of the algorithm. Forty-eight SRD processors are used for gradient, sixteen for tensor calculation and thirty-

two for the smoothing operation. Figure 5.3 shows the mapping of this approach on the ambric platform.

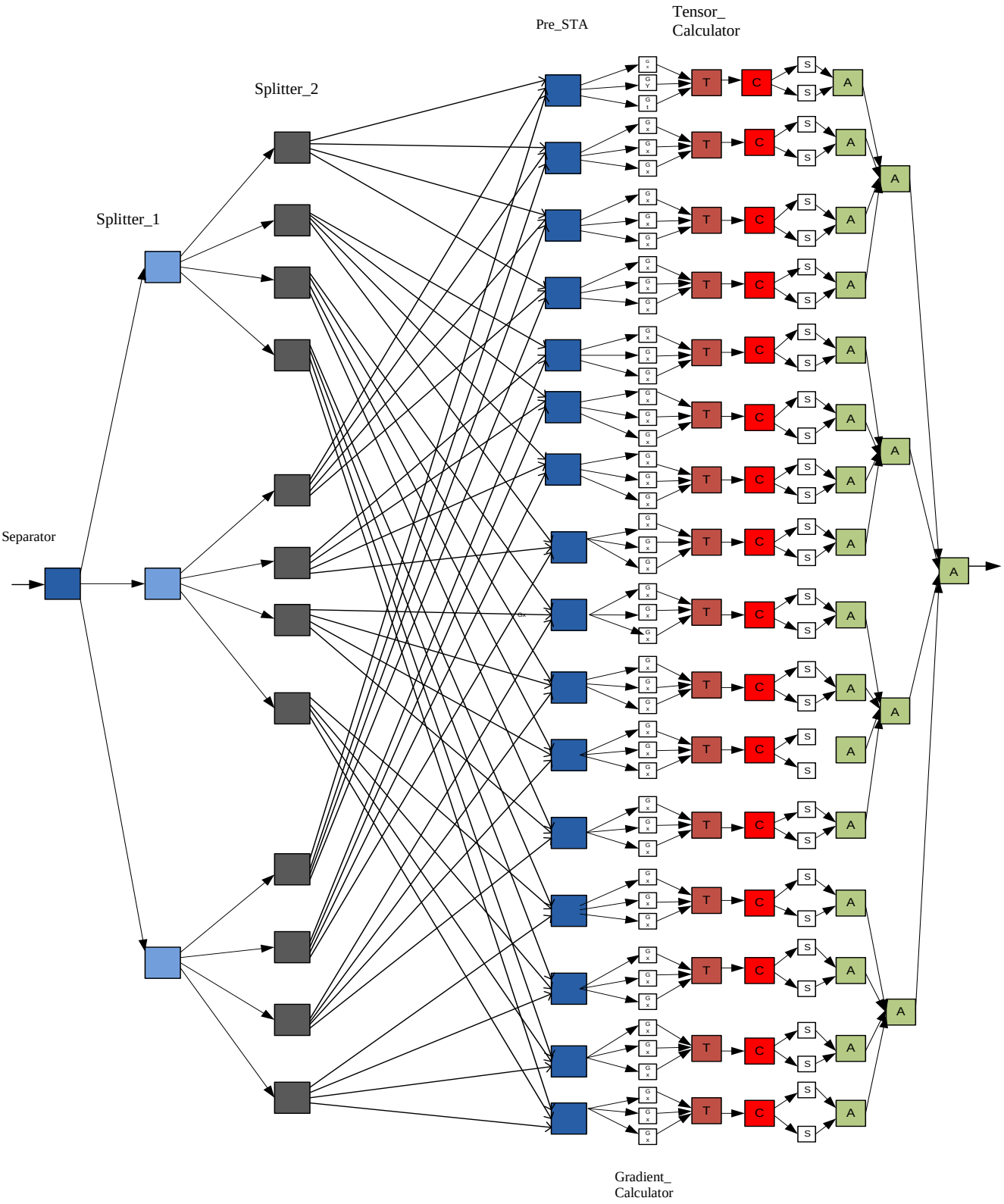


Figure 5.3: Data flow diagram of Fine-grained implementation

In this design a total of 165 processors are used of them sixty-nine are SR processors and ninety-six are SRD processors out of 336 processors.

The algorithm takes sequential images as an input like video sequence and calculates all parts of the algorithm in the pipeline manner. Now we are able to discuss each of the parts of the algorithm.

5.3 Three stages of 3D-STA

In this section we describe implementation of the three stages in 3D-STA i.e., gradient calculation, outer product calculation and smoothing operations.

5.3.1 The Gradient Calculation

Gradient block accepts three sequential images (g) and perform three dimensional gradient calculations such as horizontal (gx), vertical (gy) and over the time components (gt), using 3D sobel edge operator. To calculate gradient, '*pre_STA*' processors create data streams for gx, gy and gt starting from the top-left corner of the image in 3x3 format and then compute the actual gradient. That means, nine pixels are consumed from each image to calculate gradient for the first iteration. So, *pre_STA* consumes twenty seven values from three images, because of 3D sobel operator kernel size 3x3x3, and send all of them to *gradient_calculator* processor. *Gradient_calculator* processor multiplies sobel operator's coefficient with pixels values in three directions and produce three gradient values in three directions (gx, gy and gt). Figure 5.4 and 5.5 show the gradient block diagram and part of the codes.

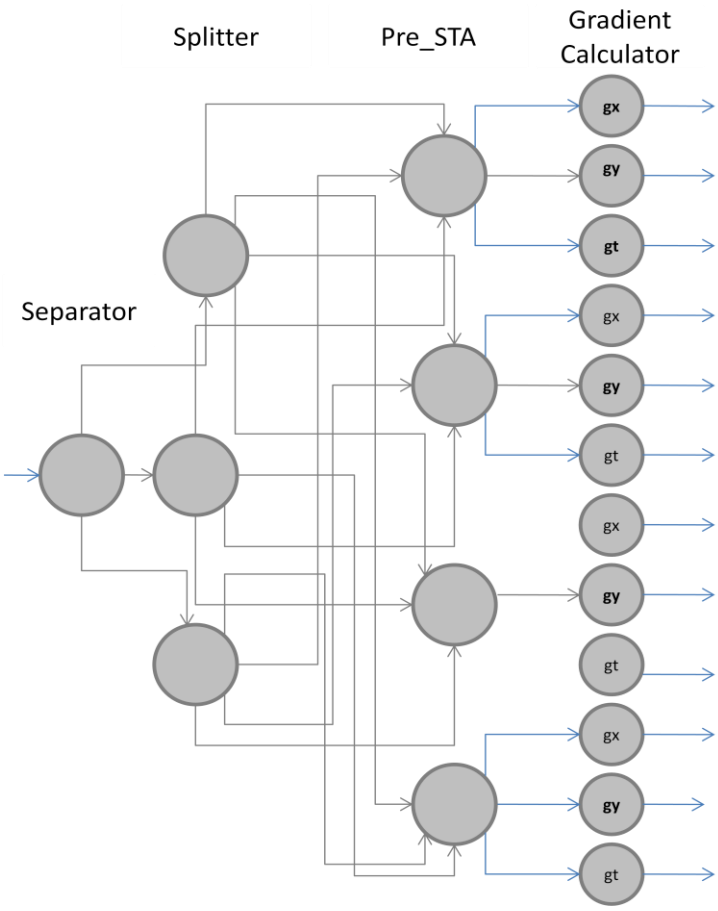


Figure 5.4: Design of the Gradient calculation block on Ambric hardware

Supporting codes are given bellow for X direction-

```
c18=in.readInt();
c17=in.readInt();
c16=in.readInt();

cnu=in.readInt();
cnu=in.readInt();
cnu=in.readInt();

c9=in.readInt();
c8=in.readInt();
c7=in.readInt();

c15=in.readInt();
c14=in.readInt();
c13=in.readInt();

cnu=in.readInt();
cnu=in.readInt();
cnu=in.readInt();

c6=in.readInt();
c5=in.readInt();
c4=in.readInt();

c12=in.readInt();
c11=in.readInt();
c10=in.readInt();
```

not used values, because of sobel operator in x direction 9 values are zero, if we multiply those pixel values with zero. Result is also zero

not used values

```
cnu=in.readInt();
cnu=in.readInt();   } not used values
cnu=in.readInt();

c3=in.readInt();
c2=in.readInt();
c1=in.readInt();

// equation to calculate gradient

Gx=c1+(c2<<a)+c3+(c4<<a)+(c5<<b)+(c6<<a)+c7+(c8<<a)+c9-(c10+(c11<<a)+c12+(c13<<a)+(c14<<b)+(c15<<a)+c16+(c17<<a)+c18);
```

Figure 5.5: Part of codes to calculate gradient (gx) in the x direction

In the above Figure 5.5, to calculate one pixel for gradient calculation it requires 27 pixel values but only 18 values are used because in 3D sobel operator nine coefficients are zero. Although, those pixel values are useless, we have to read them since data order is very important.

In our implementation instead of multiplication we used bit shift operator. As can be seen in chapter-2 the 3D sobel operator coefficients are 1, 2 and 4, so instead of multiplying by coefficient two we use one bit left shift operation and instead of multiplying four we use two bit left shift operator. This is the biggest advantage of using the sobel operator to implement in ambric because multiplication takes more time than bit shift operator. Figure 5.5 shows part of code gradient calculation in X direction, same way others direction (Y and T) gradient calculation is done according to different sobel operator in each direction.

5.3.2 Tensor block

Tensor calculation procedure uses equation-4 which is described in the chapter-2. The previous stage gives three gradient values which correspond to one pixel. In this stage, the three values are received as a row vector and then it is transposed to column vector (Figure 5.6). Finally, nine tensor values are computed by multiplying the row and column vector. Among the nine tensor values three of them are repeated values and six are independent values as shown in chapter 2. So, to make the implementation faster we have computed six values instead of nine by ignoring the computation of the three repeated values. A part of the code is depicted in Figure 5.7.

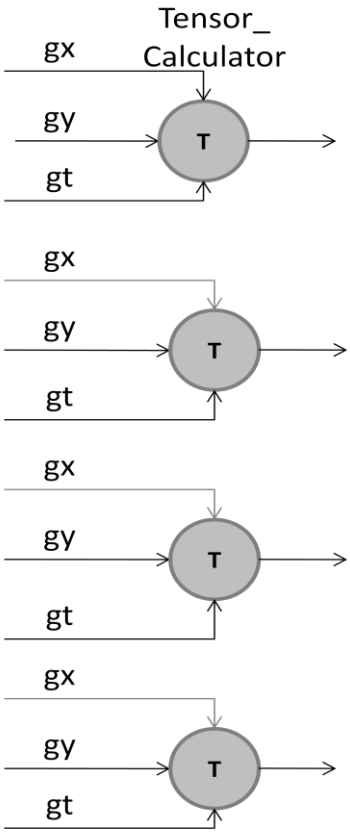


Figure 5.6: Design of the Tensor calculation block in Ambric

```
x=in_1.readInt();
y=in_2.readInt();
t=in_3.readInt();
t1= x*x;
t2= x*y;
t3= x*z;
// t4= y*x; //t1 and t4 are same value
t5= y*y;
//t6= y*z; //t6 and t8 is same value
//t7=z*x; //t3 and t7 are same value
t8= z*y;
t9= z*z;
```

Here, received three direction (x,y and t) gradient values

Figure 5.7: Part of tensor calculation program

5.3.3 Smoothing the tensor values

Smoothing the tensor values is the final stage of the algorithm. There are many algorithms to perform the smoothing operation of the tensor values. In our case we used a simple averaging technique which is 8x8 square filters. To implement this part, we used processors named *tensor_classifier* to classify the tensor values (into four parts for coarse-grained and two parts for fine-grained parallelism) and to send eight tensor values one by one to *smother*. Each tensor corresponds to six independent values. *Smother* processors receives 64 tensor values and sum all of them to produce a smoothed tensor value. Figure 5.8 shows the smoothing block and Figure 5.9 shows part of the code.

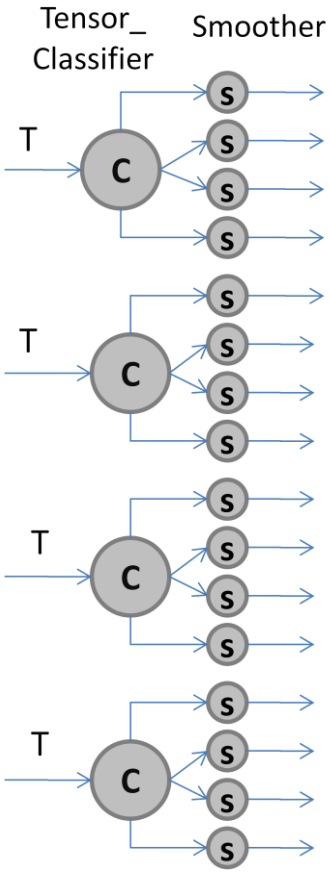


Figure 5.8: Smoothing design block in Ambric implementation

```
xx=in.readInt();
xy=in.readInt();
xz=in.readInt();
//yx=in.readInt();
yy=in.readInt();
//yz=in.readInt();
//zx=in.readInt();
zy=in.readInt();
zz=in.readInt();

smooth=smooth+1;
```

Here, in smoothing processor received six tensor values,

```
if (smooth>32)
{
   smooth=1;
}
if (smooth<=8) // in each iteration six tensors vales received and sum until fulfill the condition
{
            smooth11_tt=smooth11_tt+xx;
            smooth12_tx=smooth12_tx+xy;
            smooth13_ty=smooth13_ty+xz;
            //smooth14_xx=smooth14_xx+yx;
            smooth15_xy=smooth15_xy+yy;
            //smooth16_yy=smooth16_yy+yz;
            //smooth17=smooth17+zx;
            smooth18=smooth18+zy;
            smooth19=smooth19+zz;
            smoothA1=smoothA1+1;

            if (smoothA1==64){
                                    // when sixty four tensor is added then send to the output
}}
else if (smooth<=16)
{
     if(smoothA2==64){
}}
else if(smooth<=24)
{
   if(smoothA3==64){

}}
else if (smooth<=32)
{
  if(smoothA4==64){
}}
```

Figure 5.9: Part of code for smoothing tensor

Chapter 6

Result Analysis and Comparison

This chapter discusses the ambric implementation performance results for the two designs and compares them with the sequential Matlab program that is implemented by my thesis partner [15]. It also compares the results with an implementation of optical flow on Ambric which is specified in the related work [4].

6.1 Result Analysis

In order to analyze the performance of the design we used the perfHarness library. This library is still under development and is provided by Nethra Imaging. The performance analysis result may have variation. Whereas some portions of the design are beginning to run even as the chip configuration process is completing. This is one of the side effects of using asynchronous channel communications on the Am2045 chip - one end of the chip is up and running part of the design while the other is completing the configuration. Any attempt to write or read data on channels, which are still under configuration, simply gets blocked. The first performance result variation is therefore related to the variations in the Ambric card communications during configuration. Since the input could not be divided into workloads, first performance result needed to be used. In order to reduce the configuration-dependent time a delay object is placed in front of the PerfHarness library objects. Thus the input can be delayed until the configuration is finished on entire chip. However, there are still small variations on the results which are related to the configuration time. These variations are small enough to be ignored.

The performance result is measured using an input of 20 images of the size 256x256 for the two designs approaches. We observed that the performance result is 25 fps (frames per second) for coarse-grained parallelism implementation and 100 fps for fine-grained parallelism implementation. The performance result of fine-grained parallelism is four times faster than the coarse-grained parallelism approach, but we have used 165 processors for fine-grained and 49 processors for coarse-grained parallelism. Table 6.1 and 6.2 show the details of number and types of processors used in both approaches.

Separator	**Splitter**	**Pre-STA**	**Gradient Calculator**	**Tensor Calculator**	**Tensor Classifier**	**Smoother**	**Accumulator**
SR	SR	SR	SRD	SRD	SR	SRD	SR
1	3	4	12	4	4	16	5
Total 49 processors used in above model of them 17 processors are SR and 32 processors are SRD							

Table 6.1: Total number of processors used in coarse-grained implementation

Separator	**Splitter_1 +Splitter_2**	**Pre_STA**	**Gradient Calculator**	**Tensor Calculator**	**Tensor Classifier**	**Smoother**	**Accumulator**
SR	SR	SR	SRD	SRD	SR	SRD	SR
1	(3+12)	16	48	16	16	32	21
Total 165 processors are used of them 69 processors are SR and 96 processors are SRD							

Table 6.2: Total number of processors used in fine-grained implementation

From the above Table 6.1 and 6.2, we observed that SR processors are only used for distributing the data and for collecting the data. SRD processors are mainly used for all of the calculation of 3D-STA. As a result of SRD processors are more powerful than the SR processors and SR processor does not support multiplication.

The performance analysis library provided the total cycle counts and this total cycle count is divided by 300MHz (ambric chip clock frequency) to obtain the time in seconds. The performance result parameters are image size, number of processors, cycle count, run time in second and frame per second. The coarse-grained and fine-grained parallelism performance results and also three steps of the algorithm (gradient, tensor and smoothing) execution times are shown in Table 6.3 and 6.4 respectively. Nonetheless, the three steps of the algorithm execution times are shown in Table 6.4 separately.

Image size 20x256x256	**Coarse-grained implementation**	**Fine-grained implementation**
Total number of processors	49	165
Cycle counts	241239956	60098694
Run-time (seconds)	0,80	0.20
Frame rate	25	100

Table 6. 3: Performance results of coarse-grained implementation and fine-grained implementation

We can notice from the above table, coarse-grained implementation of 3D-STA achieved 25 frames per second (FPS) on an image stream consisting of 256x256 images and fine-grained achieved 100 fps with same image size. Coarse-grained design the frame rate ratio (100 fps over 25 fps) gives a factor of 4. The main reason of the frame rate speed-up is that the image is split into more fine-grained that allows more parallelism over the coarse-grained one.

Three steps of the algorithm	Gradient	Tensor	Smoothing
Execution time (%)	69.65%	15.96%	14.39%

Table 6. 4: Execution time for gradient, tensor and smoothing

As we can see in the Table 6.4, for the execution of coarse-grained implementation, the gradient, tensor and smoothing blocks on Ambric have taken approximately 70%, 16% and 14% of time respectively. The gradient block takes more time to execute than the other parts of the algorithm because each pixel of gradient calculation requires 54 multiplications (used faster bit shift operator) in three directions. On the other hand, fine-grained implementation on Ambric chip has taken same percent of time (0.80 seconds) to execute three stages of the algorithm because of the similarity in calculation procedures and design techniques in both of the designs.

6.2 Result Comparison

In this section, the performance results of the 3D-STA of the two implementations (Coarse-grained parallelized implementation & Fine-grained parallelized implementation) are compared with Matlab simulation. In addition, the performance results of the 3D-STA and optical flow implementation are compared.

6.2.1 Comparison of the result for two design approaches and Matlab

The Matlab implementation has been executed on Intel(R) Core 2 duo @2.10 GHz processor. On the other hand, 49 processors are used for coarse-grained and 165 processors used for fine-grained implementation. The comparison parameters are number of processors, clock speed, cycle counts, run time and frame per second. Table 6.5 shows comparison of the results of the coarse-grained and fine-grained design approaches with the sequential Matlab.

Image size 20x256x256	Coarse-grained Parallelized	Fine-grained Parallelized	Matlab program Intel(R) Core 2 duo @2.10 GHz
Number of processors	49	165	1
Clock speed	300 MHz	300 MHz	2.1 GHz

Cycle counts	241239956	60098694	-
Run time (seconds)	0.80917698	0.20	46.06
Frame per second (fps)	25.00	100.00	0,4342

Table 6. 5: Result comparison among coarse-grained and fine-grained parallelism implementation and Matlab

In coarse-grained implementation, the Ambric chip uses 49 processors running at 300 MHz and it can process 25 fps, while Matlab can process 0.43 fps. So, coarse-grained parallelism is faster than the Intel(R) Core 2 duo @2.10 GHz processor used in Matlab simulation. The frame rate speed-up ratio is approximately a factor of 58.

On the other hand, in fine-grained implementation, the Ambric chip uses 165 processors running at 300MHz. It can process 100 fps. However, Matlab process 0,43fps. Hence, fine-grained parallelism implementation is faster than the Matlab simulation. The frame rate speed-up ratio is approximately a factor of 233.

6.2.2 Comparison of the results for two design approaches and Optical flow Implementation by Brigham Young University [4]

Table 6.6 shows the result comparison among coarse-grained, fine-grained and optical flow Implementation by Brigham Young University [4].

Image size 20x256x256	**Coarse-grained implementation**	**Fine-grained implementation**	**Optical flow Implemented, BradHutchings, Brent Nelson, Stephen West [4]**
Number of processors	49(17=SR and 32=SRD)	165(SRD=96 and SR=69)	158
Clock speed	300 MHz	300 MHz	300 MHz
Cycle counts	241239956	60098694	-
Run time (seconds)	0,80	0.20	-
Frame per second (fps)	25	100	43

Table 6. 6: Result comparison among coarse-grained, fine-grained and optical flow implementation by Brigham Young University [4]

We have observed optical flow implementation on Ambric chip uses 158 processors and it can process 43 fps with a resolution of 256x256, while the coarse-grained implementation on Ambric chip uses 49 processors and it can process 25 fps. So the optical implementation is faster on ambric than the coarse-gained parallelism, which can be achieved performance approximately half (43/25) of the optical flow implementation [4].

Also as we see table 6.6, the fine-grained parallelism implementation on Ambric chip uses 165 processors and it can process 100 fps with a resolution of the 256x256. On the other hand, optical flow implementation on Ambric chip uses 158 processors and it can process is 43 fps with same resolution. The frame rate speed-up ratio gives approximately a factor of 2.33. If we applied our design technique to implement remaining parts (smoothing t direction and to calculate velocity vector) of optical flow according to the above result, we expect that our implementation will be faster than their [4] implementation.

Optical flow implementation by Hutchings, Nelson, West and Curtis [4] has multi computational stages and due to the following reasons our implementation will be faster.

The first step is derivative block, this stage calculation is done using 1D convolution in the x, y and t direction using the five elements mask to produce gradient. Actually, to calculate gradient they did not use proper gradient operator for 3D images but in our implementation we used 3D sobel operators to get proper derivative of the image. We achieved this result because of using the 3D sobel operator coefficients to calculate gradients of images. To calculate one gradient value in three directions, it requires 18x3 multiplications. However in our calculation, since 3D sobel operator coefficients are only 1, 2 and 4, we use bit shift operators instead of multiplication.

The second stage is the smoothing operation. It smoothed the gradient values to produce gxs, gys and gts and get better derivative using 3D convolution using 3x1 convolutions in t direction, 5x1 in x direction and 1x5 in y direction. In our implementation, we do not need to smooth the gradient values because the 3D sobel operator gives the most ‘correct’ gradient. This is why we achieved the good performance.

The third step is outer product calculation. This calculation has produced six new matrices (xx; yy; tt; xy; yt; xt) by performing point by point multiplications between elements of the previously computed and smoothed gxs, gyx, and gtx images. In our implementation tensor calculation as same as their implementation process but also produces six new matrices (xx; yy; tt; xy; yt; xt) by performing point by point multiplications between elements of the previously computed gradient gx, gy, and gt images.

The fourth step is smoothing tensor values, in their implementation they used 2D filter to smooth the tensor values. However, in our implementation we used a very simple method

called 8 by 8 box filters for smoothing the tensor values to ignore multiplication. This is another technique that we used to get a better performance in our implementation.

After that, they calculated the velocity vector using the equation that is mentioned in related works [4]. Finally, they smoothed the velocity vector using kernel size 7x1 and 1x7. If we compute up to optical flow, we only need to smooth in t direction and then finally also need to calculate the velocity vector. After calculating the velocity vector we can ignore smoothing. There are many optimizations we did and can be visible in our calculation. Therefore, we can say that our implementation will be faster than their implementation.

Chapter 7

Conclusion and future work

In this chapter, we will discuss our conclusions and proposals for future work.

7.1 Conclusions

The aim of this thesis work was to implement 3D structure tensor algorithm on the MPPA architecture and analyze the performance of the Ambric chip in term of speed and efficiency. The goal has been achieved by first analyzing the algorithm and identifying its three computational stages, i.e., gradient, outer products (tensor) calculation, and smoothing the tensor values. There are two design approaches are proposed to implement 3D-STA. The first approach is called coarse-grained implementation and second approach is called fine-grained implementation. The fine-grained implementation is more parallelized and got satisfactory result than the coarse-grained one.

7.2 Performance

There are 336 processors on the Ambric chip and 168 of them are SRD processors and 168 processors are SR processors. However, only 32 SRD processors and 17 SR processors are used in the coarse-grained implementation. On the other hand, only 96 SRD and 69 SR processors are used in the fine-gained implementation.

The performance of the two implementations is measured using a video stream input consisting of a sequence of images of size 20x256x256. The performance result of coarse-grained implementation is 25 frames per second (fps) and fine-grained implementation is 100 fps. Fine-grained design is four times faster than the coarse-grained design.

The results are also compared with the result of the Matlab implementation, running on Intel(R) Core 2 duo @2.10 GHz processor and we also compared it with another optical flow implementation, in terms of speed and efficiency.

Our implementation on the Ambric chip shows much better performance when compared to the Matlab implementation on Intel(R) Core 2 duo @2.10 GHz processor. The coarse-grained implementation Ambric chip exhibits a performance of 25 fps and fine-grained implementation exhibits 100 fps. If we look at the results of Matlab implementation on Intel(R) Core 2 duo @2.10 GHz processor, it can process only at a rate of 0.43 fps and optical flow implementation on the Ambric [4] can process 43 fps.

Coarse-grained parallelism implementation is 58 times faster than the Matlab implementation but it achieved approximately half (25/43) of performance of optical flow [4], whereas fine-grained parallelism implementation is 230 times faster than Matlab and 2 times faster than the optical flow implementation [4].

The implementation fulfills the real time performance requirement for 3D-STA that is used in optical flow algorithm. It competes well against optical flow implementation of Brigham Young University [4] in terms of speed and efficiency.
The main reasons for achieving real time performance is

1. The use of 3D sobel operator.
2. Efficient utilization of Ambric platform which is assigning SR and SRD processors properly.
3. The use of fast operators such as bitwise shift instead of slow operations such as multiplication.
4. Ignoring redundant computation while calculating tensor values.

7.3 Parallelism

Increasing the number of processors is a traditional approach to speed up the processing. Fine-grained has more parallelism than the coarse one because image split into more parts and distributed to more processors. Therefore, fine-grained shows better performance than the other one.

7.4 Optimization

There are several optimization have been done to compare calculation procedure of the optical flow implementation [4]. Firstly, to calculate gradient, we were able to use bitwise shift operation because of using 3D sobel operator. It gives faster gradient calculation than their [4] gradient calculation approach. Secondly, we have ignored smoothing operation after gradient calculation due to the use of 3D sobel operator and it gives the most correct gradient for 3D image. Thirdly, for smoothing operation, we used simple box filter (8x8) instead of 3D convolution to ignore multiplication. Finally, after velocity calculation, we can ignore smoothing operation.

7.5 Future work

The future works can be defined as fallows-

- In this thesis work, the entire program of the algorithm is written in aJava. Instead of aJava assembly language can be used (at least for the critical part (Gradient calculation) of the 3D-STA) to get a better performance.

- Slow part (Gradient calculation) of the algorithm can be divided into more portions (to grain more parallelism) and distributed on more processors in order not to slow down the rest of the parts of the 3D-STA.

- This project is implementation of 3D structure tensor algorithm that is used in optical flow. For the future, optical flow can also be implemented on Ambric platform. Our work can compute optical flow by smoothing the tensor in t direction and computing the velocity vector.

- This work can be compute image analysis applications for instance to detect local 3D structures and their directions. Applications include autonomous systems, robotics and CT scans.

Bibliography

[1] *Am2000 Family Architecture Reference*, release July 1, 2008.

[2] *aDesigner Users Guide*, Release 2008.3, November 14, 2008.

[3] Tom R. Halfhill, "*Ambric's new parallel processor*", Microprocessor Report, October 10, 2006.

[4] Brad Hutchings, Brent Nelson, Stephen West, Reed Curtis, "*Optical Flow on the Ambric Massively Parallel Processor Array (MPPA)*", NSF Center for High-Performance Reconfigurable Computing (CHREC), Department of Electrical and Computer Engineering Brigham Young University, International Journal of Reconfigurable Computing, July 2008.

[5] Zhaoyi Wei, "*Real- time Optical Flow sensor design and its Application on obstacle detection*. Degree of Doctor of Philosophy", Brigham Young University, August 2009.

[6] Nan Lu, Jihong Wang, Q.H. Wu and Li Yang, "*An Improved Motion Detection Method for Real-Time Surveillance*", IAENG International Journal of Computer Science, 35:1, IJCS_35_1_16, 19 February 2008.

[7] Zhaoyi Wei, Dah-Jye Lee and Brent E. Nelson, " *FPGA-based Real-time Optical Flow Algorithm Design and Implementation*", Department of Electrical and Computer Engineering, Brigham Young University, Provo, Utah, 84602 USA, Journal of Multimedia, VOL. 2, NO. 5, September 2007.

[8] Josef Bigun, "*Vision with Direction A Systematic Introduction to Image Processing and Computer Vision*".

[9] Victor Podlozhnyuk, "*Image Convolution with CUDA*", June 2007.

[10] Generation5, "*An Introduction to Edge Detection: the Sobel Edge Detector*" http://www.generation5.org/content/2002/im01.asp, visited on September 2011.

[11] John M. Bodily, "*An Optical Flow Implementation Comparison Study*", Department of Electrical and Computer Engineering, Brigham Young University, April 2009.

[12] Hai-Yun Wang and Kai-Kuang Ma*, "Structure Tensor based motion field classification and optical flow estimation*", School *of* Electrical and Electmnic Engineering, Nanyzmg Tedmological University, Singapore *639798,* ICICS-PCM 2003.

[13] Jeff Chase, Brent Nelson, John Bodily, Zhaoyi Wei, and Dah-Jye Lee, "*Real-Time Optical Flow Calculations on FPGA and GPU Architectures: A Comparison Study*", Department of Electrical and Computer Engineering, Brigham Young University, 978-0 7695-3307-0/08©, 2008 IEEE, DOI 10.1109/FCCM.2008.24

[14] "*Ambric Performance Utilities Library Documentation*", Version 1.0, Nethra Imaging, 2010.

[15] Md. Eakhlasur Rahman, "*3D Structure tensor implementation on Matlab and suggestion how to implement on Ambric*", MS thesis, Halmstad Universtiy, Sweden.

[16] [1]Dina A. Hafiz, [1]Walaa M. Sheta, [2]Sahar Bayoumi and [1]Bayumy A.B. Youssef "*A New Approach for 3D Range Image Segmentation using Gradient Method*", [1]Department of Computer Graphics and Virtual Reality, Informatics Research Institute, Mubarak City for Scientific Research and Technology, Borg El Arab, Alexandria, Egypt. [2]Department of Information Technology, Institute of Graduate Studies and Research, Alexandria, Egypt.

[17] Thomas Brox [a],*, Joachim Weickert [a], Bernhard Burgeth [a], Pavel Mra´zek [a,b], " *Nonlinear structure tensors*", [a] Mathematical Image Analysis Group, Faculty of Mathematics and Computer Science, Saarland University, Building 27, 66041 Saarbru¨cken, Germany [b] UPEK Prague R&D Center, Husinecka 7, 13000 Praha 3, Czech Republic.