SHUNTING MODEL BASED PATH PLANNING ALGORITHM ACCELERATOR USING FPGA

A Report

Presented to

The Faculty of Graduate Studies

of

The University of Guelph

by

Yingcai Dong

In partial fulfillment of requirements

for the degree of

Master of Engineering

Auguest, 2016

ABSTRACT

SHUNTING MODEL BASED PATH PLANNING ALGORITHM
ACCELERATOR USING FPGA

Yingcai Dong
University of Guelph, 2016

Advisor:
Professor Simon X. Yang

This research is mainly focusing on implement a neural network path planning algorithm into a hardware accelerator to achieve time optimization. The Shunting model is considered as the candidate of path planning algorithm. This algorithm can be applied in either Cartesian workspace or other woking environment such as multi-joint robot manipulators, so it is flexible to different working environment shows it is significant potential in real collision-free trajectory generation problems. Also, the collision-free trajectory generation can be generated without using prior knowledge of the environment. However, on the other hand, this algorithm heavily depends on the value(neural activity) of the shunting equation in each neuron. Updating each neural activity in sequence will take longer time compared to the parallelism way. This paper proposed a way of using High-Level Synthesis tools to implement the algorithm into the hardware which can take huge advantage from the nature of the hardware-parallel computation. Several different workspaces are presented to simulate the real robot working scenarios. The results of hardware accelerator have been demonstrated and discussed.

Dedication

To my parents, and those that supported me from the beginning.

Acknowledgements

I would like to acknowledge the contributions of all my friends, colleagues, and people for their great help and support during my studies at Guelph.

Sincere thanks goes to Dr. Simon Yang, Dr. Fantahun M. Defersha, and Dr. Omar Ahmed for their support and guidance throughout my graduate studies and the final project.

I am indebted to my parents Hai Dong and Huairong Xing for their continuous support during my two years study in Guelph. Last but not least, I would like to sincerely thank my dear friends that were there from the beginning.

Contents

List of Tables

List of Figures

List of Symbols

A	Passive decay rate of the neural activity
B, D	Upper and lower bounds of the neural activity
d_{ij}	Euclidean distance between positions q_i and q_j
I_i	External input to the i-th neuron
i, j	Indexes of a neuron in the neural network
k	Number of neighboring neurons of a neuron
q_i	Position vector of the i-th neuron in the state space
r_0	Receptive field constant
w_{ij}	Connection weight between the i-th and j-th neurons
x_i	Neural activity (membrane potential) of the i-th neuron
μ	Connection weight constant

Chapter 1

Introduction

Shunting model has a broad use in the field of robot path planning. For example, in solving robot tracking problem, in Chaomin Luo's paper (Luo *et al.*, 2002*a*), the author combining the shunting model with the traditional bang-bang control technique that achieves desired tracking trajectories while the robot moves in a smooth and continuous velocities when tracking. Another example is in Simon X. Yang's paper (Yang and Luo, 2004), the author proposed a novel approach for the complete coverage path planning cleaning robot with the ability to avoid obstacles in non-stationary environments, the path planning algorithm be characterized by the shunting equation.

1.1 Thesis Motivation and Objectives

Shunting model has numerous usage in robot path planning, all these implementations mentioned above are on the software simulation level; no real hardware implementation has been made. So my aim in this thesis is to develop a piece of hardware embedded with the shunting model based path planning algorithm. As for choosing the hardware, I proposed to use the FPGA (Field-programmable gate array) because the FPGA can be easily re-program and shorter time developing the prototype type. Besides, the hardware is more capable of executing instructions in parallel compared the General Purpose Processor, and the shunting model's characteristics determine that this algorithm can be faster when using hardware parallel processing.

1.2 Thesis Structure

This thesis contains seven chapters which can be grouped into four main parts. The first part is composed by chapter 1, 2 and 3, mainly focus on bringing the introduction, explaining the background and literature review. The second part is composed by chapter 4, mainly concentrate on presenting the procedures of the proposed implementation. The third part is composed by chapter 5, 6 and 7, where the result, discussion, and future work are presented. The last part is composed by chapter 8 where comprised all the programs used to test, simulate, synthesis and control the proposed hardware implementation. Each chapter's brief description is shown as follow:

- *Chapter 1* presents the introduction of the thesis including motivation and purpose, including a brief review of some implementations of shunting model based path planning algorithms. Then I will introduce the thesis structure for this report.

- *Chapter 2* presents the related background information for the proposed implementation. Including the mathematical explanation of the shunting model, and the mechanism of the shunting model based path planning algorithm. Also, I will also introduce the hardware and tools that are used in this research.

- *Chapter 3* presents the literature review about the tools used for synthesis the hardware, the related FPGA accelerator and researches on path planning hardware implementation. More detailed shunting model related implementation will be reviewed to prove the broad use of this model in path planning. Then I will review some FPGA High-Level Synthesis tools (HLS tools) to provide sufficient evidence that the HLS tools are now powerful and efficient enough for me to finish this research.

- *Chapter 4* provides the details of developing the path planning hardware accelerator from sketch to the final product. This research starts from writing the path planning algorithm in C and profiles its most time-consuming code block.

Then partition the time-consuming code block from the main code and use HLS tools to synthesis this block of code into RTL and validate its function. Next step is to build up the accelerator systems using Vivado. At last, using Vivado SDK to develop the software that controls this hardware accelerator.

- *Chapter 5* feed in different test cases (simulating different scenarios) into the hardware accelerator and get the result and list out the acceleration factor in various scenarios.

- *Chapter 6* analysis the result and validate that the hardware accelerator can effectively shorten the compute time.

- *Chapter 7* summarizes the drawbacks of the proposed implementation and provide some improvement suggestions for the future work.

Chapter 2

Background and Literature Review

In this chapter, I will focus on explaining the shunting model based path planning algorithm in detail. I will also briefly introduce the hardware and the related development tools. At last, relate literature review will be present.

2.1 Shunting Model's Characteristics

The shunting model was first proposed by Hodgkin and Huxley(Hodgkin and Huxley, 1952) and modified and implemented by Simon X. Yangs paper(Yang and Meng, 2001) into robotics path planning algorithm, the author proposed a way of solving dynamic collision-free trajectory problems using biologically inspired neural network. Shunting model has numerous advantages in actual implementation. For example, it can be applied in Cartesian workspace of cleaning robot or as the multipoint robot arms used in the factory. Besides that, since the shunting model is a neural network based model, the target information is transferred through each neuron and spread to the whole workspace. Shunting model defines that only adjacent neuron are connected, so the neural activity can only transfer within these limited connections. Compare to some other previous models (e.g. (Al-Sultan and Aliyu, 1996), (Brooks and Lozano-Prez, 1985), (Crowley, 1985), (Donald, 1987), (Ilari and Torras, 1990), (Kant and Zucker, 1986), (Li and Bui, 1998)) which use global methods to determine optimal paths in the workspace that suffers from intensive computation, the shunting

model on the other hand can effectively diminish the frequency of information communications when determining the way towards the target and prevent the parameter in each neuron from exploding or vanishing. In other words, shunting model reduces the computation expense, especially in the complex environment. Also, as a model to approach collision-free trajectory, the shunting model will retain the obstacle information within the obstacle's location while the target can continuing broadcasted information to attract the robot. Thus, the robot will neither attract by the obstacle nor move to obstacle location.

2.1.1 Shunting Equation

Shunting model's equation is written as

$$\frac{dx_i}{dt} = -Ax_i + (B - x_i)\left([I_i]^+ + \sum_{j=1}^{k} \omega_{ij}[x_j]^+\right) - (D + x_i)\,[I_i]^-, \tag{2.1}$$

where: A, B, and D are the neural activity's passive decay rate, upper and lower bound for the neural activity, respectively; x is the neuron's neural activity (x_i is the ith neuron's neural activity, while x_j is the adjacent neuron j's neural activity); k is the number of adjacent neurons of the ith neuron; $[I_i]^+ + \sum_{j=1}^{k} \omega_{ij}[x_j]^+$ is the excitatory and inhibitory inputs, respectively; I_i is the internal input to ith neuron, and I_j is the external input from ith neuron's adjacent neuron j. The external/internal input I's equation is written as

$$I = \begin{cases} E & \text{if there is a target} \\ -E & \text{if there is an obstacle} \\ 0 & \text{otherwise} \end{cases}, \tag{2.2}$$

where $E \gg B$ is a very large positive constant (Yang and Meng, 2001). Parameter ω_{ij} is the weight between connected ith neuron and jth neuron, and it is symmetric, which means $\omega_{ij} = \omega_{ji}$, and the weight function defines as:

$$\omega_{ij} = f\,(d_{ij}) \tag{2.3}$$

where $d_{ij} = |q_i - q_j|$ is the Euclidean distance between q_i and q_j in the workspace. The $f(a)$ in *Equation* (2.3) is defined as

$$f(a) = \begin{cases} \mu/a & \text{if} \quad 0 < a < r_0 \\ 0 & \text{if} \quad a \geq r_0 \end{cases}, \tag{2.4}$$

where μ and r_0 are positive constraints. r_0 stands for the neuron local connection radius.

As for $|a|^+$ and $|a|^-$ in *Equation* (2.1) stands for the non-linear above threshold and non-linear below threshold, receptively. The definitions are written as *Equation* (2.5) and *Equation* (2.6).

$$[a]^+ = \begin{cases} a & a > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.5}$$

$$[a]^- = \begin{cases} -a & a < 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.6}$$

2.1.2 Shunting Models Property Analysis

Obstacles Neural Activity: We divide the workspace into many square cells; each cell has the same area. Then each neural will generate a value use shunting equation, and we call this value: neural activity. So the robot will always follow the highest neural activity value in the surround. The initial value of all the neural activity is 0. As shown in *Equation* (2.1), if there is an obstacle on a neuron, the neural activity of the obstacle will always be negative. Whats more important, other adjacent neural will not receive the neural activity from the obstacle, because if $x_j < 0$, then $[x_j]^+ = 0$. Which means the obstacles activity will not affect other neurals activity, in other words, the obstacles information will not spread, and it will stay within the obstacle.

Target Neural Activity: On the other hand, the target neuron will not only have the highest neural activity in the workspace but also can affect other neurons. In other words, the information of the target will be spread to all over the workspace.

Other Neurals Activity: As for other neurons, they will affect each other through the nearby connection. They play the role of the conduct.

2.2 FPGA

FPGA is the aberration of "Field-Programmable Gate Array". It is an integrated circuit which can be configured by the designer or customer after manufacturing. The FPGA configuration is specified using Hardware Description Language (HDL) (Ling, 2009).

Programmable logic blocks and hierarchy of reconfigurable interconnects are the essential components of FPGAs. The interconnects connect the logic blocks based on different user configurations to perform various combinational functions.

Historically, FPGAs compare to their ASIC (Application Specific Integrated Circuit) counterparts, are less power efficient, slower and less functionality. However, recently, some newer coming out FPGA products such as Xilinx Virtex-7 and Altera Stratix 5 have come to rival corresponding ASIC solutions by providing lower power consumption, faster speed, cheaper materials cost and increased possibilities for re-configuration 'on-the -fly'. Which previously a design may need to have 6 to 10 ASICs, now the design can be achieved using only one FPGA (Kuon and Rose, 2006).

As for applications on the FPGA, not all applications are suitable for FPGA, although any problems that are computable, can be executed by the FPGA. Only some of the specific applications which take the advantage of the FPGA's parallel nature and optimality regarding the number of gates used for a certain process can achieve significant speed increase. Hardware acceleration is another trend on the usage of FPGAs, which the FPGA can accelerate certain parts of an algorithm and share the computation result with a generic processor.

2.3 Vivado Design Suit

Vivado Design Suit is a new IP and system-centric design environment (Feist, 2012) produced by Xilinx for synthesis, analysis of HDL designs and accelerates design productivity.

The design suite consists of three applications: Vivado High-Level Synthesis, Vivado, and Vivado SDK.

- **Vivado High-Level Synthesis (HLS) tool** bridges hardware and software domains, it automatically converts the C, C++, and SystemC programs into an RTL (Register Transfer Level) implementation that can directly synthesize into a Xilinx FPGA. The HLS tool also provides plenty of directives that take full advantage of the FPGA's parallel architecture. The user can easily include these directives into functions or programs so that the HLS tool can be more specifically optimizing and mapping the program onto the hardware.

- Vivado can help the user achieve more complex design using block design, the build in IP-integrator allows the user to quickly integrate and configure IP from the Xilinx IP library or other IPs made by the user. It works seamlessly with Vivado HLS. It also allows the user to compile and synthesize their designs, perform timing analysis, examine RTL diagrams and simulate design's reaction. It is a more advanced yet powerful software compare to its old opponent Vivado ISE.

- Xilinx Software Development Kit (SDK) allows the user to create software platforms and applications targeted Xilinx embedded processors. It works with hardware designs created by Vivado.

2.4 Literature Review

In Simon X. Yang and Chaomin Lous paper they discussed the using the shunting model of neural networks to solve the complete coverage path planning problem (Yang

and Meng, 2001). Moreover, this kind of algorithm is more easy to calculate. As for this type of method, the robot uses the neural activity to make a choice by its neighbor neural activities, by this approach, the robot can avoid stuck in the place where it is surrounding is either cleaned area or obstacles. Because using neural activities update every step, the place which is uncleaned will spread the information in each update and eventually attract the robot moving to the uncleaned area.

In another Simon X. Yang and Chaomin Lous paper (Yang *et al.*, 2002) they discussed using a novel biologically inspired neural computational algorithm is proposed for coverage path planning with sudden changes and moving obstacles in a varying environment. The proposed model algorithm is computationally efficient. Using the additive model, they achieved clean robot moving and cleaning in the dynamic environment. Moreover, the path is automatically generated from the dynamic activity landscape of the neural network and the previous robot location.

In Qiu X, Shirong Liu's paper: A Rolling Method for Complete Coverage Path Planning in Uncertain Environments (Qiu *et al.*, 2004), they discussed the question of Complete Coverage Path Planning with a novel planning method integrating rolling windows and biologically inspired neural networks. This algorithm can achieve collision-free path planning in uncertainty environment.

In the paper A solution to vicinity problem of obstacles in complete coverage path planning (Luo *et al.*, 2002*b*) it mainly discussed path planning of clean robot in some real scenarios such as whom to deal with the corners and when facing an obstacle, whom to plan the path effectively using neural networks.

In Winterstein's paper (Winterstein *et al.*, 2013), the author designed two test cases to compare the FPGA developing time consumption and the yield RTL (Register Transfer Level) design's performance between using Vivado High-Level Synthesis and using RTL languages. The test cases include two compute intensive machine learning related algorithms. From the algorithmic perspective, both algorithms yield the same result. However, these two algorithm has significantly different computational properties. Both algorithms have a implemented by hand-written RTL as a comparison. As the result, the performance between the handwritten and high-

level synthesis automatically generated RTL design are similar, yet the developing time spent using high-level synthesis is significantly shorter. Therefore, using Vivado High-Level Synthesis to finish the RTL design is not only practical but also significantly reduce design effort.

The High-Level Synthesis tools allow user to put more focus on higher abstraction and complex system level and algorithmic level problems, while no need to worry about the register transfer level issue (Meeus *et al.*, 2012). In this paper (Meeus *et al.*, 2012), the author also points out that the high-level synthesis tools can take care of the interface synthesis and yield a proper interface, the user does not need to consider the data transfer and control signals between the generated hardware and its periphery. Therefore making the hardware communication easier, and lower the threshold for making the complex system which contains numerous hardware that communicates with each other.

Chapter 3

Methodology

This chapter focuses on explaining how to implement the shunting model in C language and how to modify the code to target on the FPGA board using Xilinx HLS. The hardware used in this project is Vivado ZedBoard, the first community-based Zunq-7000 Embedded Processing Platform.

The shunting model based path planning algorithm's infrastructure is neural activity. The path planning robot will track down to the target based on the neural activity's strength gradient which is shown in the *Figure* 3.1 shown below.

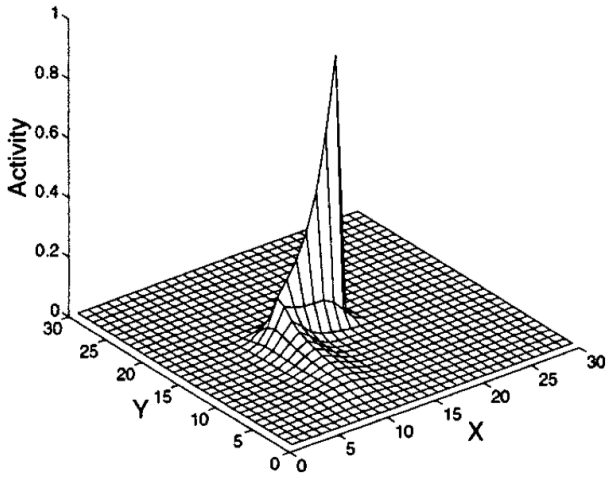

Figure 3.1: Neural activity landscape when robot arrives at the target position. (image from: Yang and Meng, 2001)

The shunting model path planning workflow is shown in *Figure* 3.2. The program

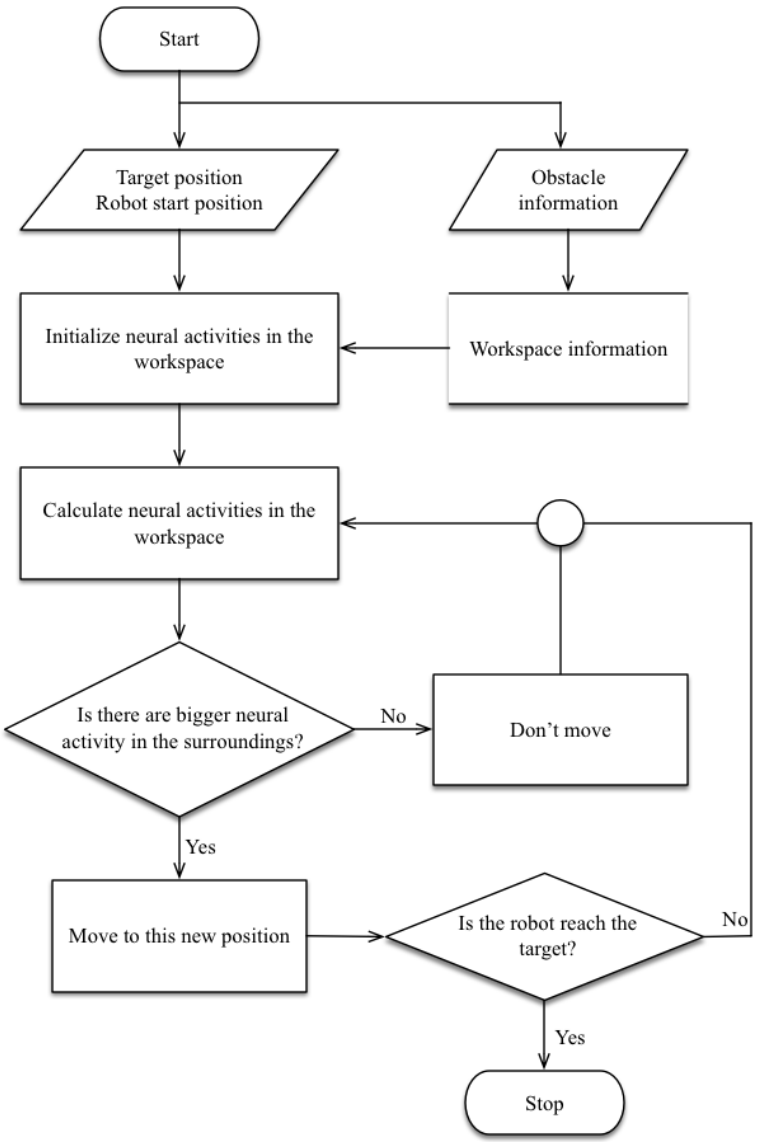

Figure 3.2: Shunting model based path planning algorithm's workflow

is designed to allow user modify different target position, robot start position and obstacle information in the workspace. The workspace is a predefined 32×32 matrix. After all the parameters have been set, the neural activities in the workspace will be initialized to zero. After initialization, the program starts to calculate neural activities in the whole workspace until the neurons around the robot, forms activity gradient and let the robot start moving towards the target.

3.1 Code Explanation

The C code is attached in the *Appendix A*.1. Only the most significant part of the code will be explained in detail. As for the rest of the code, please check the comment of the code in the appendix.

Shunting Model Parameters refers to: passive decay rate A, upper bound of neural activity B, lower bound neural activity D, positive constant E, constant μ and neuron local connection radius r_0. Changes of these parameters will affect the path planning performance and robot trajectory. The optimal parameters combination is listed in *Table* 3.1.

Table 3.1: Shunting Model Parameters

A	B	D	E	μ	r_0
10	1	1	100	1	2

Workspace & Initial Neural Activity as I mentioned at the beginning of this chapter, the workspace here is represented in the form of a matrix, thus in C, the workspace is implemented as a two dimension array. As you can see in the *Figure* 3.3, this is a 32×32 resolution workspace. I use number 4 (highlighted in red) as the obstacle, and number 1 (highlighted in green) as the target, number 7 (highlighted in green) as the robot start position. As for the rest of the workspace, I use 0 to present, the robot can move freely in this area. Of course, if I increase the workspaces resolution (by increasing the elements in the array, i.e. 1024×1024) the robot trajectory will be more optimized. However, at the same time, the computation complexity will also increase, and from the aspect of hardware, larger workspace larger data volume, and longer data transfer time. Thus more optimization techniques need to be considered when implementing in the hardware.

The initial neural activities can also be represented in a 32×32 array, and as mentioned at the beginning of this chapter, the initial value is 0.

```
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 1, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 4, 4, 4, 4, 4, 4, 4, 4, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

Figure 3.3: Workspace containing obstacle and target position information presented as a 32×32 matrix

Sum of External Excitatory Inputs (SEEI) written in mathematical expression is:$\sum_{j=1}^{k} \omega_{ij}[x_j]^+$. To get the weight equation (*Equation* (2.3)), we need to know which neuron is in the range because r_0 in *Equation* (2.4) is a parameter that can be changed by needs. In C code, I use the following step to achieve:

1. Using nested *for* loop to traversal all the element in the workspace and calculate the distance between the centre neuron and other neurons using $d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

2. Screen out the elements with the required distance.

3. Calculate the weight using *Equation* (2.3).

The C code implementation is given below.

```
double SEEI(int circlecenter_i, int circlecenter_j,
        double c_neural_activity[MATRIX_ROWS][MATRIX_COLS]){
    // MDF: monotonically decreasing function
```

```
4       double distance, MDF;
5       double sum = 0;
6       for (int i = 0; i< MATRIX_ROWS; i+=1) {
7           for (int j = 0; j< MATRIX_COLS; j+=1) {
8               distance = sqrt(pow((i-circlecenter_i), 2)
9                   + pow((j-circlecenter_j), 2));
10              if ((distance > 0) && (distance < radious)) {
11                  MDF = mu/distance;
12              } else {
13                  MDF = 0;
14              }
15              sum = sum + MDF * UpperBound(c_neural_activity[i][j]);
16          }
17      }
18      return sum;
19  }
```

3.1.1 Implementation Method

The implementation process is shown in *Figure* 3.4. Thus, firstly I execute the code (*Appendix A*.1) in the integrated development environment (IDE) Xcode7.3 (*Figure* 3.5 shows the Xcode7.3's interface.) to make sure the functionality meets design requirement: adjustable robot position, target position and change obstacle information. Also, the code has to simulate the same behavior as the shunting model's mathematical indicate.

3.1.2 Time Profile&Code Adjustments

Using the Xcode7.3 built-in time profile instrument, I was able to get the time profile result shown in *Figure* 3.6. The result indicates that the program's total execution time is 142ms, and 99.8% of the time has been used in calling the function SEEI. Thus, in this code, there is a significantly time-consuming function.

It is reasonable to consider implementing function SEEI into hardware. However,

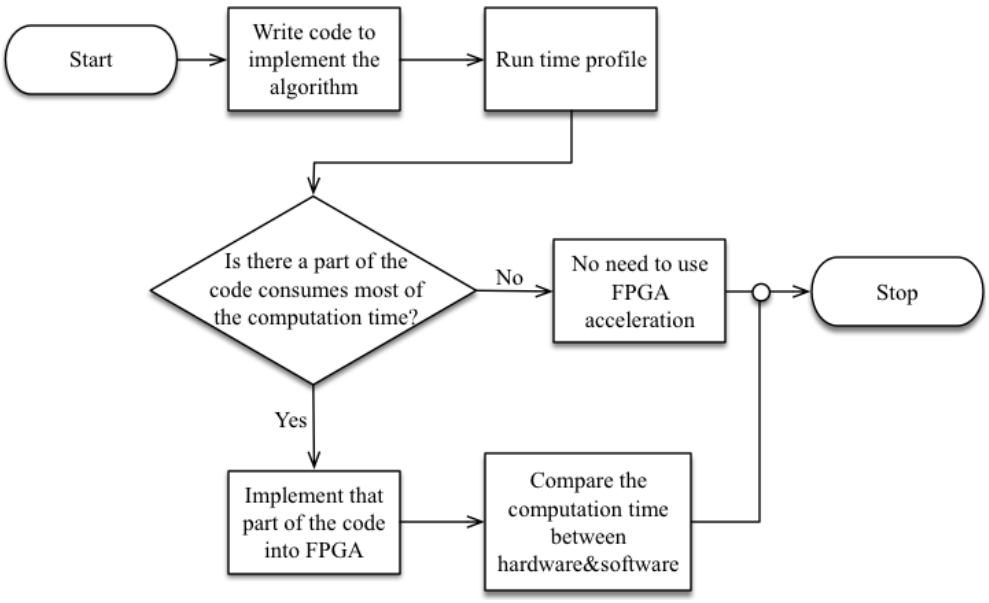

Figure 3.4: The process of how compute intensive algorithm implement into hardware accelerator

this function contains a square-root, which is extremely luxury to using in hardware because it will not only waste hardware resource but also lag the computation. The reason to use square-root here is that it is more generalized and flexible, the parameter μ and r_0 are all adjustable, and this flexibility allows me to tuning the parameter to get the optimal performance and trajectory.

Since all the parameters have been adjusting to the optimal state (parameters can be found in *Table* 3.1), the program can be more efficient if it is less generalized, and be more specific.

Thus, some adjustments need to apply in the code. The modified code can be found in *Appendix A*.2. The main changes are:

- Replaced the square-root algorithm in SEEI function with a more specific and efficient implementation.
- Removed all unnecessaried global variables.
- Packed the SEEI function with other functions into a new function: ComputeCore.

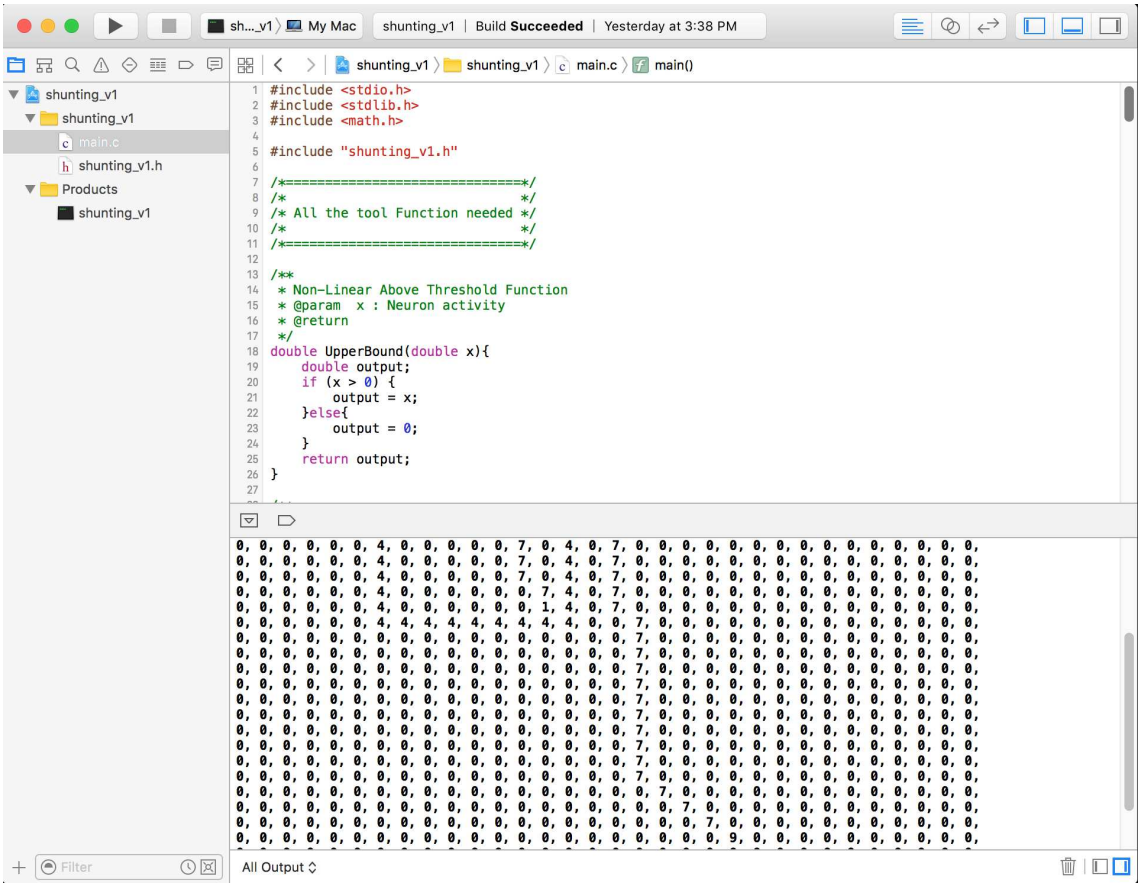

Figure 3.5: Xcode 7's interface

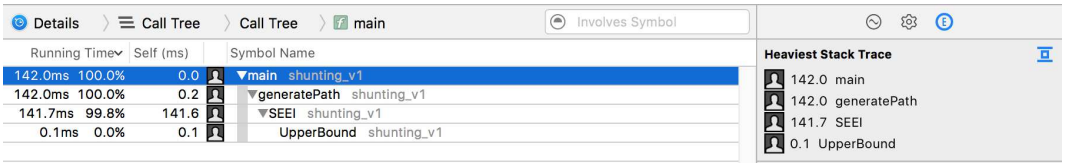

Figure 3.6: Time profile result

- The new function can now simulate the whole *Equation* (2.1), instead of only part of the Equation.

The ComputeCore function is shown down-below, this function simulates the time integral of the *Equation* (2.1). *float a* is the current neural activity while *float b* represents the workspace information. The output is the 32×32 neural activities array.

```
1 void ComputeCore(float c_act[LEN][LEN], float work_info[LEN][LEN]) {
2     float dx = 0;
3     for (int p = 0; p < LEN; ++p){
```

```
4           for (int q = 0; q < LEN; ++q){
5               float sum = 0;
6               for(int i = -1; i < 2; ++i){
7                   for(int j = -1; j < 2; ++j){
8                       if(boundary_check(i+p, q+j) == 1){
9                           if (i == 0 || j == 0) {
10                              if ((i+j) !=0) {
11                                  sum += UpperBound(c_act[i+p][q+j]);
12                              }
13                          } else {
14                              sum += 0.707107*UpperBound(c_act[i+p][j+q]);
15                          }
16                      }
17                  }
18              }
19              dx = -(10 * c_act[p][q]) + (1 - c_act[p][q]) * (UpperBound(
                    ExternalInput(work_info[p][q])) + sum) - (1 + c_act[p][q
                    ]) * LowerBound(work_info[p][q]);
20              c_act[p][q] += (dx * 0.01);
21          }
22      }
23      return;
24  }
```

The time profile for this modified code (*Appendix A.2*) is shown in *Figure* 3.7. The profile shows that the IDE takes $1079ms$ to execute the modified code. However, because the modified code is significantly more efficient compared to the old version, the IDE could not capture the running time in a single execution. Thus I purposely add a for loop to let the code loop 100 times. So the real execution time is $1079ms \div 100ms = 10.79ms$.

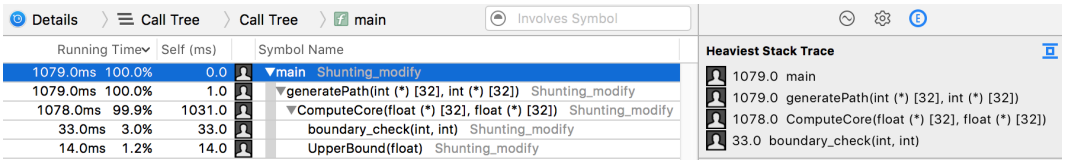

Figure 3.7: Time profile for the modified code

Also, as shown in *Figure* 3.7, the function ComputeCore used up to 99% of the computation time. It is even more worth to implement in the hardware.

3.2 Code Partition

Now we know which part of the program consumes most of the computation time. So, in this section, I am going to partition the code into two parts: TestBench and function ComputeCore and the ComputeCore will later convert to RTL. *Figure* 3.8 illustrate the relevance between the TestBench and Source file.

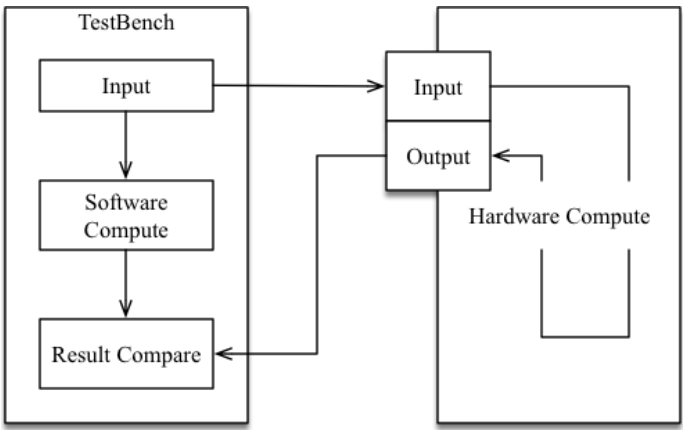

Figure 3.8: Code partition

The input/output port in the Hardware Compute shown in *Figure* 3.8, is responsible for receiving input data and returning the result. However, when considering hardware implementation, some extra functions will be added to the program to communicate with the BUS interface, which I will discuss in the next section. The partition code at this stage is identical to the code shown in Section 4.1.2

3.3 AXI Interface

When writing the code that applies to the hardware, we need to consider not only the functions that compatible with the mathematical expression but also needs to consider the yield output IP core synthesized by Vivado HLS.

There are three issues need to consider. The HLS IP core is not a standalone system; it is only the hardware that helps to accelerate part of the algorithm. So the function of measuring time (compare the computation time between pure software and software/hardware coherence) and read/write large amount of data from the memory needs more hardware to achieve. As shown in *Figure* 3.9 the system need to add at least two more IP cores to achieve the functions mentioned above.

- The first issue is do we need to build these IP cores ourself?
- The second issue is what is the communication standard among these hardwares (IP cores).
- The third one is in order to insure the buses transferring data correctly among hardwares, do we need to modify the bus interface of these IP cores?

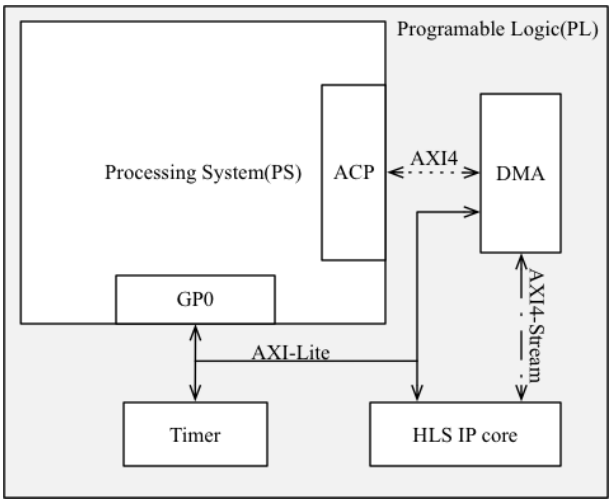

Figure 3.9: PS and PL Partitions Diagram

As shown in *Figure* 3.9, there are three different types of connections: AXI4, AXI-Lite, and AXI4-Stream. Besides, there are two IP cores except the HLS IP core. However, this does not mean that the user needs to build these connections, bus interfaces, and IP cores from the sketch.

Regarding the first and the second issue, Vivado has a built-in IP core library which provides abundant feature that meets users numerous demands. Over And Above Vivado integrated a powerful feature called: IP Integrator. IP Integrator

liberates the user from the sophisticated system design by applying automated IP subsystem generation (more details will be discussed in Section 4.6). Illustrated by the example of the system in *Figure* 3.9, I can add DMA, Timer and HLS exported IP core (shunting model based path planning algorithm) to the design. Then uses the IP Integrator built-in feature "Run Block Automation" to configure interconnect, peripherals, memory map, and device driver. Accordingly, the Vivado will take care of the first and the second issue. The user also does not need to worry about the interface for those IP provided by Xilinx. So the user only needs to concern about the interface of the HLS IP core. Therefore the HLS IP core need to add the interface that can receive control signals from the ARM Cortex A9 CPU in the Processing System, and the interface for receiving input data and sending output result. In my implementation, interface directives will be used to specify these two interfaces; Some extra functions written in C++ will be mainly used for specifying the side channels of the AXI4-Stream interface (meanwhile converting between the floating point data and unsigned data of AXI4 protocol).

AXI is not a newly created protocol, in fact, it is a more powerful successor of the AMBA (Advanced Microcontroller Bus Architecture) 3.0 protocol. It is a high performance, high bandwidth, low latency host bus. The relationship between the AMBA 3.0 and AXI can be concluded in the figure shown below.

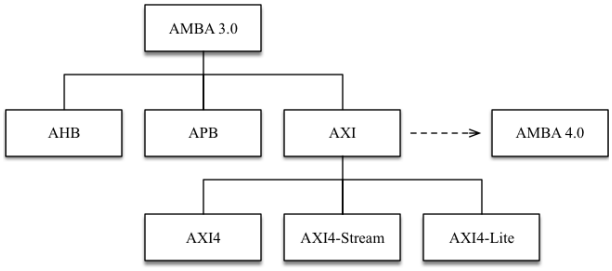

Figure 3.10: AXI4 Protocol

The AXI4-Lite is commonly used for sending the control signal or transferring light weight data. However, the AXI4-Stream is commonly used in high-speed data transfer scenarios such as image, video, and large matrix transfer. Now I will explain how the AXI4-Stream interface is designed. The first thing to consider is the number of AXI4-

Stream interface I need to implement. The AXI4-Stream interface can be applied to both array input and output. The function "ComputeCore" mentioned in Section 4.1.2 contains two parameters: $c\_act\,[32]\,[32]$ and $work\_info\,[32]\,[32]$. They are both 2-dimensional array responsible for the input. However, the AXI4_Stream interface can not handle argument which is both read and write. The main idea is to pack array $c\_act\,[32]\,[32]$ and array $work\_info\,[32]\,[32]$ into a 1-dimensional array in_stream[1024] and use one AXI4_Stream interface as input. Then create another AXI4_Stream interface dedicated for the output. The second thing to consider is the AXI4_Stream Side-Channels. I manually configured the AXI4_Stream Side-Channels in the code that is because in the later block design phase, to use the IP-Integrator to connect the DMA automatically with the AXI4_Stream interface, both sides should have the same signal port. Besides, in Vivado HLS, the AXI4_Stream Side-Channels are optional signals which mean Vivado HLS will only abstract TREADY, and TVALID signals from the user's code and the user can define other signals in their code depend on their needs. The Third thing to consider is how to transfer data on the bus. The "ComputeCore" function receives *floating point* data type as input and output. However, on the bus, the data is transferred in *unsigned int* format. So the AXI4_Stream interface also needs to convert data type between *float* and *unsigned int*.

The code is shown below modified the AXI4_Stream Side-Channel by including the header file "ap_axi_sdata.h" provided by Xilinx, and defined a user-defined data type: AXI_S.

```
1 #include <ap_axi_sdata.h>
2 typedef ap_axiu<32,4,5,5> AXI_S;
```

As for the *stream_in* and *stream_out* function shown below specified the interface with Side-Channel signals, and also converts the data into the correspond format.

```
1 template <typename T, int U, int TI, int TD> T stream_in(ap_axiu <sizeof
    (T)*8,U,TI,TD> const &e){
2 #pragma HLS INLINE
3
4         assert(sizeof(T) == sizeof(int));
5         union{
```

```
6                   int ival;
7                   T oval;
8           } converter;
9           converter.ival = e.data;
10          T ret = converter.oval;
11
12          volatile ap_uint<sizeof(T)> strb = e.strb;
13          volatile ap_uint<sizeof(T)> keep = e.keep;
14          volatile ap_uint<U> user = e.user;
15          volatile ap_uint<1> last = e.last;
16          volatile ap_uint<TI> id = e.id;
17          volatile ap_uint<TD> dest = e.dest;
18
19          return ret;
20  }
```

```
1   template <typename T, int U, int TI, int TD> ap_axiu<sizeof(T)*8,U,TI,TD
        > stream_out(T const &v, bool last = false){
2   #pragma HLS INLINE
3           ap_axiu<sizeof(T)*8,U,TI,TD> e;
4           assert(sizeof(T) == sizeof(int));
5           union{
6                   int oval;
7                   T ival;
8           } converter;
9           converter.ival = v;
10          e.data = converter.oval;
11
12          // set it to sizeof(T) ones
13          e.strb = -1;
14          e.keep = 15; //e.strb;
15          e.user = 0;
16          e.last = last ? 1 : 0;
17          e.id = 0;
18          e.dest = 0;
19          return e;
```

```
20 }
```

3.4 Apply Solutions

Vivado has built-in pliantly of directives that give the user more control over the code implementation. The directive is a new feature that allows the user to access the powerful capabilities in each implementation command. It is a new command option that directs the commands behavior towards a different objective than when using the commands default. Within a command, the directive can actives different flows, objectives and different set of algorithms. The directive helps the user to unlock much more exploration of design solutions than what is possible before.

In this section, I am going to apply different directives and compare three different solutions from the latency and resources usage perspective. The best solution among the three will be exported and used in the next design stage.

The solution targets to the device $xc7z010clg400-1$ and the clock period sets to $10ns$. When providing the code, Vivado HLS will:

- Converts each C/C++ code operation into corresponding hardware operation then schedules those operations into clock cycles. Vivado HLS will pack as many operations as possible in a single clock cycle using the user-selected clock period and device delays.

- The user can write his customized interface to define how data can be transferred to the hardware block and written out. Alternatively, HLS will use interface synthesis to automatically synchronized the data communication issue.

- Maps each of the hardware operations on a correspond hardware unit.

- Apply users specified directive commands to optimize the latency, resources, etc.

- Export the final design and the correspond reports.

3.4.1 Solution 1

Solution 1′s result yield by the default synthesis result. Which means heres no directives adding to the code. The result determines that it take 215111 clock cycles to compute the result based on the specified clock period and target hardware. The design could execute with a maximum clock period of 8.63ns.

The area estimates in *Figure* 3.11 shows the expected amount of resources to use on the PL: 24 DSP48 slices, about 2770 FFs (Flip-Flops) and 5728 LUTs (Look-Up Tables). The figures are just estimation result. Since the RTL synthesis result still

Performance Estimates

- **Timing (ns)**
  - **Summary**

Clock	Target	Estimated	Uncertainty
default	10.00	8.63	1.25

- **Latency (clock cycles)**
  - **Summary**

Latency		Interval		Type
min	max	min	max	
215111	215111	215112	215112	none

  - **Detail**

Utilization Estimates

- **Summary**

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	227
FIFO	-	-	-	-
Instance	0	24	2682	5389
Memory	6	-	0	0
Multiplexer	-	-	-	112
Register	-	-	88	-
Total	6	24	2770	5728
Available	280	220	106400	53200
Utilization (%)	2	10	2	10

Figure 3.11: Solution 1's synthesis report

needs further transformation, which from RTL code to gate-level components, also placing and routing process and other gate-level optimizations will further affect the

final result. The estimated data output rate is 0.54 KSPS(Kilo Samples Per Second), as shown in *Equation* (3.1).

$$\begin{aligned} 215111 \times 8.63 \quad ns = 1.856 \quad ms \\ 1/1.856 \quad ns = 0.54 \quad KSPS \end{aligned} \tag{3.1}$$

3.4.2 Solution 2

The first design (solution 1) can be optimized. The optimizations mainly rely on pipeline directive command.

Pipelining allows the operations inside the loop or function to behave in parallel. Based on the time profile analysis, the most time consumption part of the code contains four *for* loop nests, and this block of codes function is to update the neural activity for each neuron in the 32×32 matrix simultaneously. Hence, applying the pipeline directive to the loop can take advantage of the parallelism - letting each neuron update neural activity in parallel.

The pipeline directive will unroll all sub-loops nested inside the loop or function. To illustrate how pipeline works, please first check the example code given below.

```
1 for (int rows = 0; rows < 32; ++rows){
2         for (int cols = 0; cols < 32; ++cols){
3                 Read_data;
4                 Compute_data;
5                 Write_output;
6         }
7 }
```

This example code contains two *for* loops. The total iteration time is $32 \times 32 = 1024$ times. However, without the pipeline, to execute this *for* loop, the operations will be processed like *Figure* 3.12 (here we assume each operation takes one clock cycle to execute). After using the pipeline, the *for* loop will be unrolled and execute in parallel; the process is shown in *Figure* 3.13. The latency remains the same, but the throughput is better, which means fewer iterations.

The drawback of using pipeline is that this directive may use up many hardware

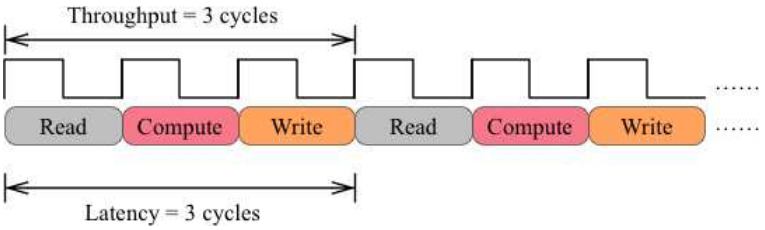

Figure 3.12: Operation without pipeline

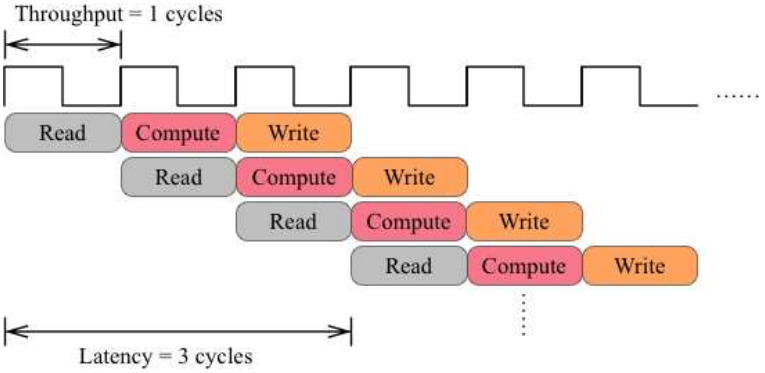

Figure 3.13: Operation with pipeline

resources especially when using it in multi-level loop nests. However, the synthesis result shown in *Figure* 3.14 implies that the design used 2% of BRAM, 9% of DSP48E slices, 3% of FFs (Flip-Flops) and 13% of LUTs (Look-Up Tables) resources respectively. Accordingly, from the resources usage perspective, the optimization is acceptable, besides from the reduce-latency aspect, the result is also satisfying. The estimated data output rate is 13.98 KSPS, as shown in *Equation* (3.2) compared to solution 1 with the data output rate: 0.54 KSPS.

$$\begin{aligned} 8286 \times 8.63 \quad ns = 0.0715 \quad ms \\ 1/0.0715 \quad ms = 13.98 \quad KSPS \end{aligned} \tag{3.2}$$

3.4.3 Solution 3

Solution 3 focused on using the inline directive command to optimize the latency further. Inline directive embedded the separate function entity in the hierarchy to the calling functions and forms an integral entity. Hence, the inlined function will no longer exist as a distinct level of hierarchy. To illustrate how inline command works, an example code is shown below.

Performance Estimates

- **Timing (ns)**
  - **Summary**

Clock	Target	Estimated	Uncertainty
default	10.00	8.63	1.25

- **Latency (clock cycles)**
  - **Summary**

Latency		Interval		Type
min	max	min	max	
8286	8286	8287	8287	none

  - **Detail**

Utilization Estimates

- **Summary**

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	1486
FIFO	-	-	-	-
Instance	0	21	1877	4138
Memory	6	-	0	0
Multiplexer	-	-	-	922
Register	-	-	2143	502
Total	6	21	4020	7048
Available	280	220	106400	53200
Utilization (%)	2	9	3	13

- **Detail**

Figure 3.14: Solution 2's synthesis report

```
1 int add_sub(int in1, int in2, int *out1, int out2){
2         *out1 = in1 + in2;
3         *out2 = in1 - in2;
4 }
5
6 int shift(int in1, int in2, int *out1, int *out2){
7         *out1 = in1 >> 1;
8         *out2 = in2 >> 2;
9 }
```

```
10
11 int calling_function(int in1, int in2, int *out1, int *out2){
12         int temp1, temp2;
13
14         add_sub(in1, in2, &temp1, &temp2);
15         shift(temp1, temp2, out1, out2);
16 }
```

Based on the code, *Figure* 3.15 shows the difference between "no inlining" and "inlining" RTL function hierarchy. When applied inline directive into the code, as

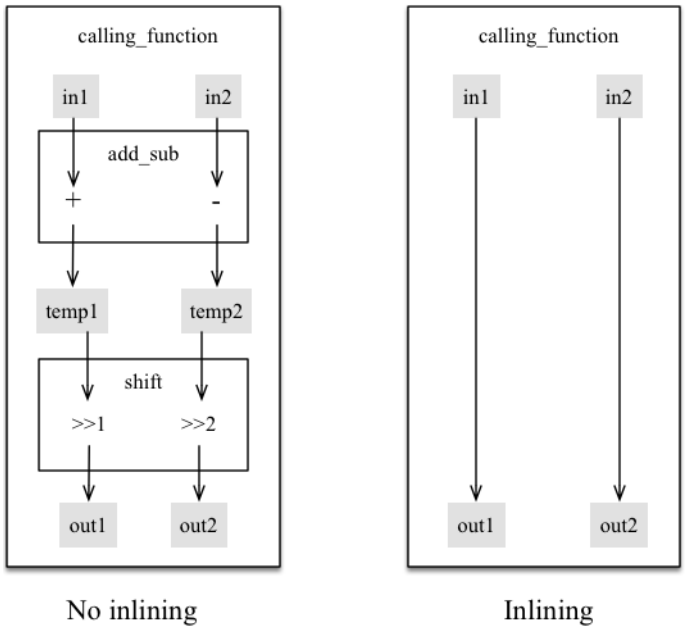

Figure 3.15: Inlining demonstration

mentioned ahead, the separate function will be devolved and integrated into the upper-level function hierarchy. As a result, the translated RTL block will be simplified and thus, achieve less latency. The synthesis result is shown in *Figure* 3.16. Compare to solution 2; a slight latency decrease is achieved: from 8286 to 8280. So the estimated data output rate is 13.99 KSPS, as shown in *Equation* (3.3), that is the highest data output rate among these three solutions.

$$\begin{aligned} 8280 \times 8.63 \quad ns = 0.071 \quad ms \\ 1/0.071 \quad ms = 13.99 \quad KSPS \end{aligned} \tag{3.3}$$

Performance Estimates

- **Timing (ns)**
  - **Summary**

Clock	Target	Estimated	Uncertainty
default	10.00	8.63	1.25

- **Latency (clock cycles)**
  - **Summary**

Latency		Interval		Type
min	max	min	max	
8280	8280	8281	8281	none

  - **Detail**

Utilization Estimates

- **Summary**

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	1475
FIFO	-	-	-	-
Instance	0	26	2325	4987
Memory	6	-	0	0
Multiplexer	-	-	-	922
Register	-	-	2122	502
Total	6	26	4447	7886
Available	280	220	106400	53200
Utilization (%)	2	11	4	14

- **Detail**

Figure 3.16: Solution 3's synthesis report

3.5 FPGA Block Design

The *Figure* 3.17 shows the design of system block using Vivado's built-in feature: IP-Integrator. Compare to the PS and PL Partitions diagram shown in *Figure* 3.9, and we can observe that in real hardware implementation, Vivado automatically adds more IP-core ensure the whole system operates properly.

In this section, I will focus on explaining two things. The first one is why I choose to use AXI DMA, and the second is why I choose to use the ARM ACP port instead

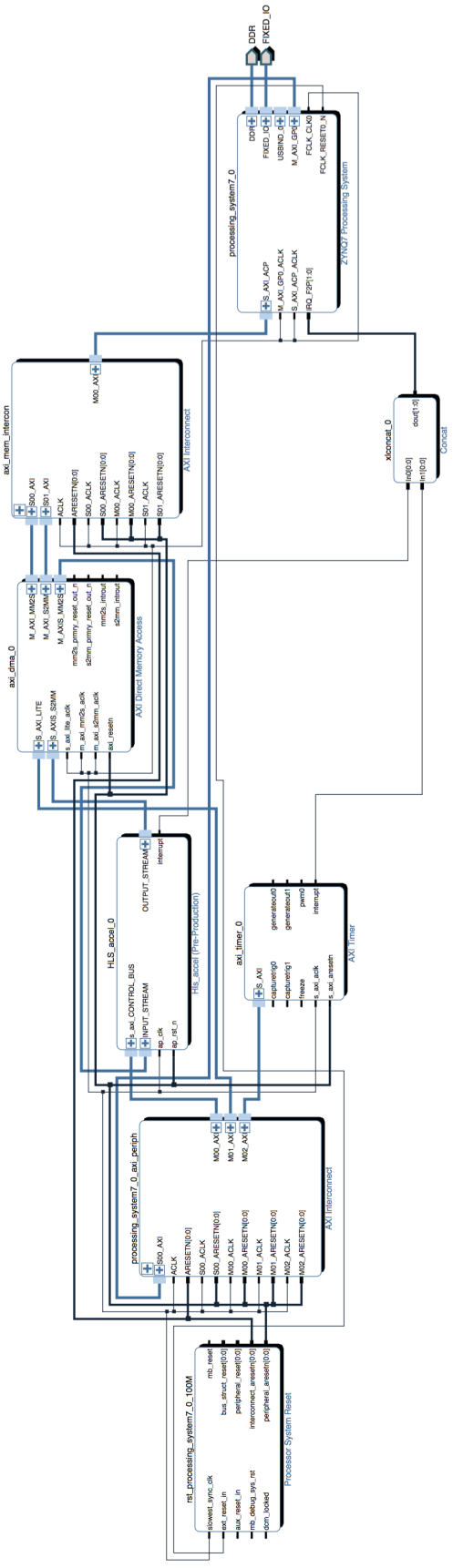

Figure 3.17: System block design

of other ports.

3.5.1 AXI DMA

The AXI DMA allows peripherals to access the memory directly in high speed and high bandwidth. In our case, the peripheral is HLS IP core. The IP core consumes one array which contains 2048 floating point elements and generate one array that contains 1024 floating point elements per execution. Thus, with this amount of data throughput, using DMA to bridge the HLS IP core with the memory will be more efficient.

Besides, because I already manually modified the Side-Channel's signals in the code, so the HLS IP core can directly connect its INPUT_STREAM port with M_AXIS_MM2S port and OUTPUT_STREAM port with the S_AXIS_S2MM port.

3.5.2 ARM ACP

Before introducing the ARM ACP (Accelerator Coherence Port), I have to mention one more thing regarding the AXI DMA. The HLS IP core can connect to the DDR memory or L2 cache through AXI DMA using the same interface which I implemented in the previous section. The L2 cache is controlled by the SCU (Snoop Control Unit), and the SCU is directly connected with the two Cortex A9 CPU. Thus, using L2 cache will have less latency in communication with the CPU compared to the DDR memory.

3.6 Software Design Using SDK

As for collaborative development, I can use the Vivado to structure the hardware system (as I discussed in the previous section). However, if I want the hardware to operate according to the needs, then I need to design the software using Vivado SDK. As for Zynq, the software development means for developed software for the ARM CPU. The code can be seen in the Appendix.

To accurately compare the speed acceleration and make the result convincible, the following requirement/statement are strictly followed.

- Base line in comparison: The acceleration comparison is between the FPGA (partially hardware implementation) and the ARM CPU (fully software implementation). Technically, in the FPGA implementation, the ARM CPU also takes a small amount of data processing work. The hardware (HLS IP core) computes the neural activity within the workspace then transfer this data to the ARM CPU, and it is the software part to decide the next step movement based on the neural activities.

- Coding style: To compare the computation speed, we should exclude the differences causing by the different coding style. So I chose to use the same code segments from the Vivado High-Level Synthesis phase into the base line design, and partially for the FPGA comparison design (partially here is because part of the code has already become the hardware, but the rest of the code remains the same).

- Test samples: To make the result more convincible, the software must be able to change: the obstacles in the workspace, target position, and robot start position.

- Trajectory check: The result should not only compare the speed difference between the software and hardware accelerator but also should make sure that both computed trajectory are the same. If the trajectory is different, then it is meaningless to compare the speed difference.

Chapter 4

Results

In the test phase, six different test examples are used. These six test examples include different obstacles distribution in the workspace, target position, and robot start position. The aim is to as much as possible to cover different scenarios in path planning. In these test cases, all the software computed trajectories are identical to the software&hardware's results.

The raw output data is shown in *Figure* 4.1, to make the result more readable, I made some adjustments, and the adjustments are shown in *Table* 4.1.

Table 4.1: Output Adjustments

Number shown in result	Replaced by
0	White
4	Black
7	Red
9	Dot
1	Check mark

4.1 Case 1

Test case 1 shown in *Figure* 4.2, the timer counts 54954576 cycles for the software to work out the result on the ARM CPU. As for the hardware part, the total clock

4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4
4	0	0	0	0	0	0	0	0	0	0	0	7	7	7	7	7	7	7	7	7	0	0	0	0	0	0	0	0	0	0	4
4	0	0	0	0	0	0	0	0	0	0	7	0	0	0	0	0	0	0	0	0	7	7	7	7	0	0	0	0	0	0	4
4	0	0	0	0	0	0	0	0	0	7	0	0	4	0	0	0	0	0	0	0	0	0	0	0	7	7	0	0	0	0	4
4	0	0	0	0	0	0	0	0	7	0	0	0	4	0	0	0	0	0	4	4	4	4	4	4	4	4	7	0	0	0	4
4	0	0	0	0	0	0	0	7	0	0	0	0	4	0	0	0	0	0	4	0	0	0	0	0	0	7	0	0	0	0	4
4	0	0	0	0	0	0	7	0	0	0	0	0	4	0	0	0	0	0	4	0	0	0	0	0	7	0	0	0	0	0	4
4	0	0	0	0	0	7	0	0	0	0	0	0	4	0	0	0	0	0	4	0	0	0	0	7	0	0	0	0	0	0	4
4	0	0	0	0	7	0	0	0	0	0	0	0	4	0	0	0	0	0	4	0	0	7	7	0	0	0	0	0	0	0	4
4	0	0	0	7	4	4	4	4	4	0	0	0	4	0	0	0	0	0	4	0	7	4	4	4	4	0	0	0	0	0	4
4	0	0	0	7	0	0	0	0	4	0	0	0	4	0	0	0	0	0	4	0	0	7	0	0	4	0	0	0	0	0	4
4	0	0	0	7	0	0	0	0	4	0	0	0	4	0	0	0	0	0	4	0	0	0	7	1	4	0	0	0	0	0	4
4	0	0	0	7	0	0	0	0	4	0	0	0	4	0	0	0	0	0	4	0	0	0	0	0	4	0	0	0	0	0	4
4	0	0	0	7	0	0	0	0	4	0	0	0	4	0	0	0	0	0	4	0	0	4	4	4	4	0	0	0	0	0	4
4	0	0	0	0	9	0	0	0	4	0	0	0	4	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	4
4	0	0	0	0	0	0	0	0	4	0	0	0	4	0	0	0	0	0	4	4	4	4	4	4	4	4	0	0	0	0	4
4	0	0	0	0	0	0	0	0	4	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4
4	0	0	0	0	0	0	0	0	4	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4
4	0	0	0	4	4	4	4	4	4	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4
4	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4
4	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4
4	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4
4	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4
4	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4
4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4

Figure 4.1: Raw output data composed by different numbers indicate different information: obstacles(4), robot start position(9), target position(1), and robot moving trajectory(7)

cycles for the ARM CPU + AXI DMA + hardware accelerator (referred as software&hardware) is 4908931 cycles. Thus, the acceleration factor is 11.1948153274, calculate by *Equation* (4.1).

$$\begin{aligned} AccelerationFactor &= SoftwareTime/HardwareTime \\ AccelerationFactor &= 54954576/4908931 = 11.1948153274 \end{aligned} \tag{4.1}$$

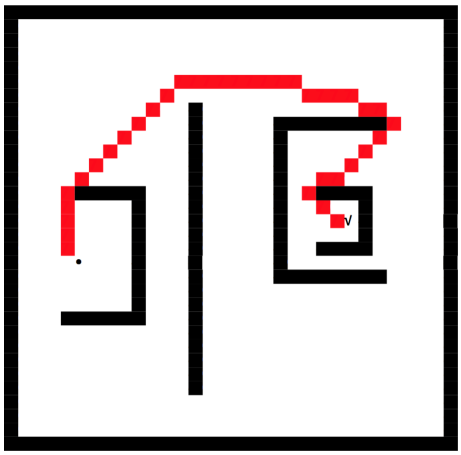

Figure 4.2: Test Case 1's Moving Trajectory

4.2 Case 2&3

Test case 2 and 3 shown in *Figure* 4.3 and *Figure* 4.4. In these two tests, the obstacles are in the same position. However, the robot start position and target position are different. The total runtime for software in test case 2 is 38007700 cycles and the total run time for the software&hardware accelerator is 3148540 cycles. Thus the acceleration factor for test case 2 is 12.071531567, calculate by *Equation* (4.2).

$$\begin{aligned} AccelerationFactor &= SoftwareTime/HardwareTime \\ AccelerationFactor &= 38007700/3148540 = 12.071531567 \end{aligned} \tag{4.2}$$

As for test case 3, the software running time is 40529932 cycles, the software&hardware running time is 3539394 cycles. So the acceleration factor is 11.4510936053. The calculation is shown in *Equation* (4.3).

$$\begin{aligned} AccelerationFactor &= SoftwareTime/HardwareTime \\ AccelerationFactor &= 40529932/3539394 = 11.4510936053 \end{aligned} \tag{4.3}$$

4.3 Case 4

Test case 4 simulates a more complex scenario, as shown in *Figure* 4.5. The software part takes 48364151 cycles to compute the trajectory, and the software&hardware

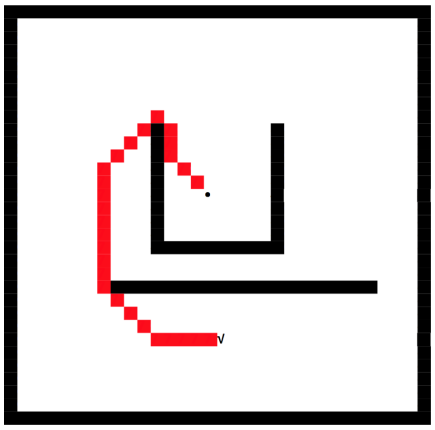

Figure 4.3: Test Case 2's Moving Trajectory

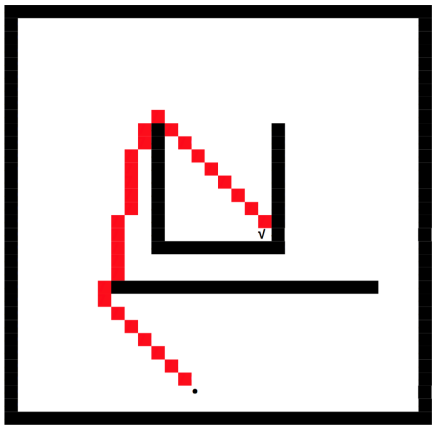

Figure 4.4: Test Case 3's Moving Trajectory

accelerator part uses 4321851 cycles to work out the result. So the acceleration for test case 4 is 11.1906105. The factor is calculate by *Equation* (4.4).

$$\begin{aligned} AccelerationFactor &= SoftwareTime/HardwareTime \\ AccelerationFactor &= 48364151/4321851 = 11.1906105 \end{aligned} \tag{4.4}$$

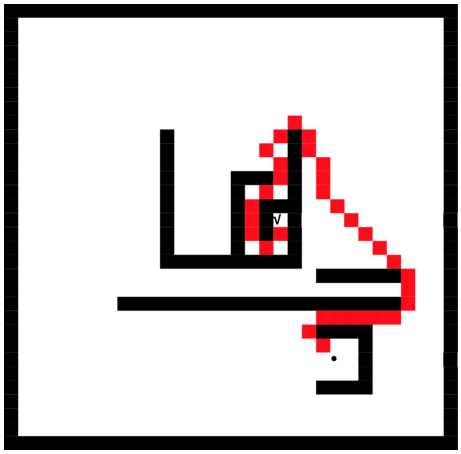

Figure 4.5: Test Case 4's Moving Trajectory

4.4 Case 5&6

Test case 5 and 6 are two more cases that simulate the complex scenario. In case 5, the software part uses 68989811 cycles to compute the path and 6571926 cycles for

the software&hardware accelerator. The acceleration factor is 10.4976548732, based on *Equation* (4.5).

$$\begin{aligned} AccelerationFactor &= SoftwareTime/HardwareTime \\ AccelerationFactor &= 68989811/6571926 = 10.4976548732 \end{aligned} \tag{4.5}$$

As for test case 6, the software part takes 81908006 cycles, and the software&hardware accelerator part uses 7941894 cycles. The acceleration factor is 10.3134096224 based on *Equation* (4.6).

$$\begin{aligned} AccelerationFactor &= SoftwareTime/HardwareTime \\ AccelerationFactor &= 81908006/7941894 = 10.3134096224 \end{aligned} \tag{4.6}$$

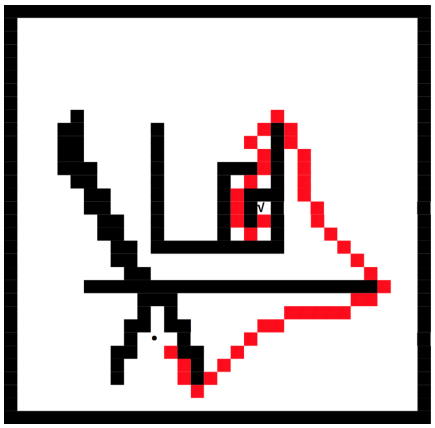

Figure 4.6: Test Case 5's Moving Trajectory

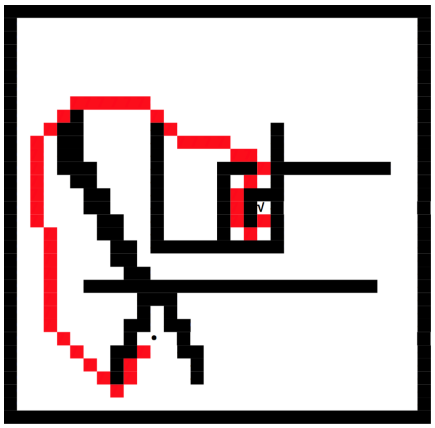

Figure 4.7: Test Case 6's Moving Trajectory

Chapter 5

Discussions

From the previous chapter, the results showed that the software + DMA + hardware accelerator could achieve more than ten times speed acceleration. The *Table* 5.1 also illustrates the Number of Obstacles each test case contains (the number of black squares each case contains) and the Trajectory Steps (number of steps robot moves from the start position to the target). The relations between the Acceleration Factor and the Number of Obstacles is shown in *Figure* 5.1, the Acceleration Factor and Number of Obstacles have an inverse relationship. The acceleration will start to ascend when the Number of Obstacles decreases. The trend also fits in *Figure* 5.2, when the number of Trajectory Steps decrease, the more acceleration can be achieved.

Table 5.1: Result Status

	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6
Acceleration Factor	11.194	12.071	11.451	11.190	10.497	10.313
Number of Obstacles	201	172	172	201	231	239
Trajectory Steps	36	27	29	32	41	46

We can observe the speed acceleration factor decrease when the number of Trajectory Steps increase. My hypothesis is that compared to purely running the program in software (shown in *Figure* 5.3), the software + DMA + hardware accelerator architecture will have more latency between software and DMA, DMA and hardware

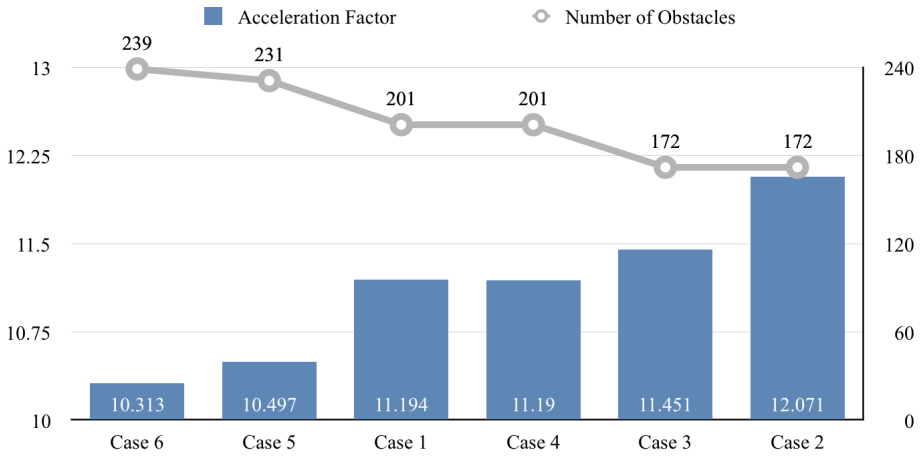

Figure 5.1: Relatzionship Between Acceleration Factor and Number of Obstacles

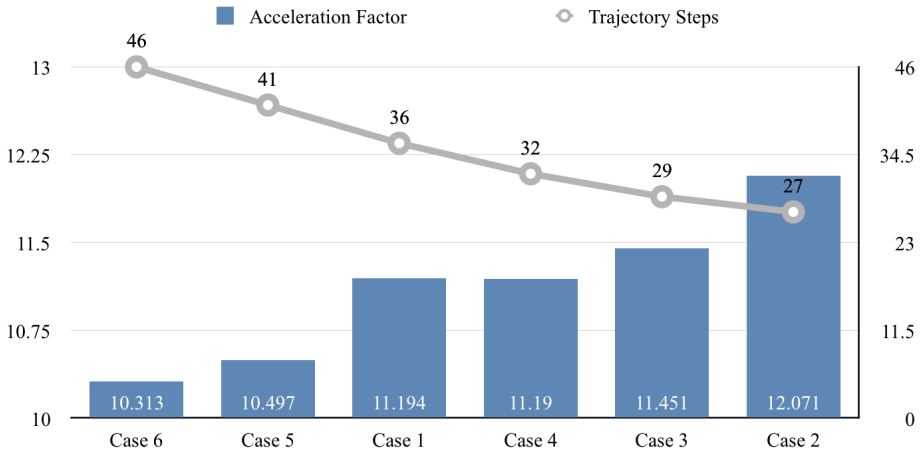

Figure 5.2: Relationship Between Acceleration Factor and Trajectory Steps

accelerator (shown in *Figure* 5.4). When the Trajectory Steps increase, the iterations for both implementation will increase, but the latency between software and hardware will be magnified, and this might be the reason why when the Trajectory Steps increase, the Acceleration Factor will decrease. However, I think the Number of Obstacles is not the main reason leads to Acceleration Factor decrease. In general, more obstacles means the neural activity is harder to spread and causes more iterations on both implementations.

Lastly as shown on *Figure* 5.2, the Acceleration Factor in Case 1 and Case 4 are not following the inverse trends (when Trajectory Steps decrease, the Acceleration Factor also decrease). As the hypothesis explained, the Acceleration Factor decrease might cause by the communication latency; the latency will superimpose when the

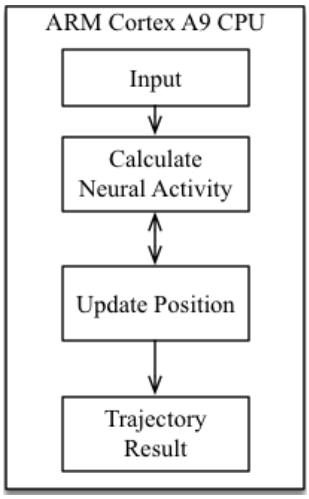

Figure 5.3: Pure software architecture

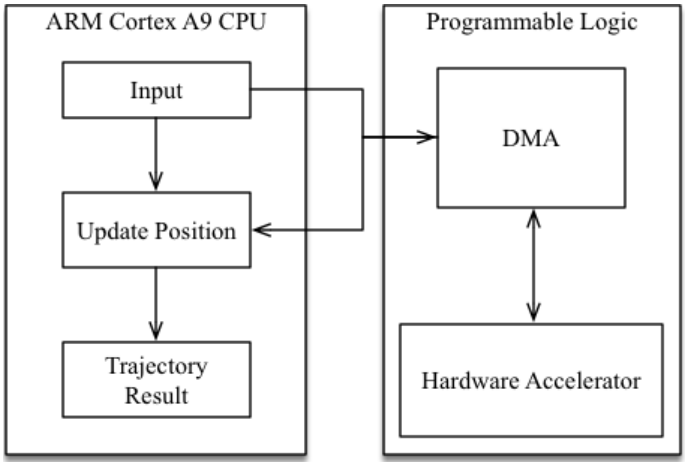

Figure 5.4: Software + DMA + Hardware Accelerator Architecture

program iterations increase. However, the Trajectory Steps cannot truly reflect the number of iterations because the robot will not move until the neural activity forms the gradient, so before this happens, the Trajectory Steps will be zero. In other words, the real program iteration time is always longer or equal to the Trajectory Steps.

Chapter 6

Future Works

As we discussed in the previous chapter, the software + DMA + hardware accelerator architecture (shown in *Figure* 5.4) has its drawbacks. Due to the data communication latency, the speed acceleration will decrease in some complex path planning scenarios. This architecture can be called "partially hardware acceleration architecture." Logically the alternative way of implement this hardware accelerator is: the software part only responsible for modifying and feed the initial data to the hardware accelerator and the hardware accelerator is responsible for all the data computation include the position update function. This full hardware accelerator, in theory, should have less data communication latency compared to the partial version.

However, the difficulty here for the full hardware architecture is to let hardware determine the whether the has already reached the target position and if it needs to stop the trajectory computation.

In Vivado HLS, I tried to use the *while* syntax (the example is shown below). However, the code containing similar syntax could not pass the C/RTL co-simulation in Vivado (takes extremely long time to simulate, the longest simulation lasts for 48 hours and still going on, so I have to force it to stop). The purpose of C/RTL co-simulation is to validate whether the C (or C++) code and the synthesized RTL functions in the same way. If the C/RTL co-simulation failed or the user skipped this step may cause the real hardware, not functions as expected.

```
1 while(robotPosX != targetPosX && robotPosY != targetPosY){
```

```
2         /* Code */
3 }
```

In my case, the C/RTL co-simulation takes almost forever, I choose to skip this step and continue the rest of the work, but unfortunately, this implementation failed to work on the hardware (Zynq board). Hence, in the future work, I will try to explore other ways to implement the full hardware architecture.

References

Al-Sultan, K. S. and M. D. S. Aliyu (1996). A new potential field-based algorithm for path planning. *Journal of Intelligent and Robotic Systems* **17**(3), 265–282.

Brooks, R. A. and T. Lozano-Prez (1985). A subdivision algorithm in configuration space for findpath with rotation. *IEEE Transactions on Systems, Man, and Cybernetics* **SMC-15**(2), 224–233.

Crowley, J. (1985). Navigation for an intelligent mobile robot. *IEEE Journal on Robotics and Automation* **1**(1), 31–41.

Donald, Bruce R. (1987). A search algorithm for motion planning with six degrees of freedom. *Artificial Intelligence* **31**(3), 295–353.

Feist, Tom (2012). Vivado design suite. *White Paper*.

Hodgkin, A. L. and A. F. Huxley (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology* **117**(4), 500–544.

Ilari, Joan and Carme Torras (1990). 2d Path Planning: A Configuration Space Heuristic Approach. *The International Journal of Robotics Research* **9**(1), 75–91.

Kant, Kamal and Steven W. Zucker (1986). Toward Efficient Trajectory Planning: The Path-Velocity Decomposition. *The International Journal of Robotics Research* **5**(3), 72–89.

Kuon, Ian and Jonathan Rose (2006). Measuring the Gap Between FPGAs and ASICs. In: *Proceedings of the 2006 ACM/SIGDA 14th International Sympo-*

sium on Field Programmable Gate Arrays. FPGA '06. ACM. New York, NY, USA. pp. 21–30.

Li, Z. X. and T. D. Bui (1998). Robot Path Planning Using Fluid Model. *Journal of Intelligent and Robotic Systems* **21**(1), 29–50.

Ling, Tan Swee (2009). DESIGN A MP3 PLAYER USING FPGA.

Luo, Chaomin, S X Yang and Xiaobu Yuan (2002*a*). *Real-time area-covering operations with obstacle avoidance for cleaning robots*. Vol. 3. IEEE.

Luo, Chaomin, S X Yang, D A Stacey and J C Jofriet (2002*b*). *A solution to vicinity problem of obstacles in complete coverage path planning*. Vol. 1. IEEE.

Meeus, Wim, Kristof Van Beeck, Toon Goedemé, Jan Meel and Dirk Stroobandt (2012). An overview of today's high-level synthesis tools. *Design Automation for Embedded Systems* **16**(3), 31–51.

Qiu, Xuena, Shirong Liu and S X Yang (2004). A Rolling Method for Complete Coverage Path Planning in Uncertain Environments. In: *2004 IEEE International Conference on Robotics and Biomimetics*. IEEE. pp. 146–151.

Winterstein, Felix, Samuel Bayliss and George A Constantinides (2013). *High-level synthesis of dynamic data structures: A case study using Vivado HLS*. IEEE.

Yang, S X and C Luo (2004). A neural network approach to complete coverage path planning. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics* **34**(1), 718–724.

Yang, S X and M Meng (2001). Neural network approaches to dynamic collision-free trajectory generation. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics* **31**(3), 302–318.

Yang, S X, Chaomin Luo and M Meng (2002). *A neural computational algorithm for coverage path planning in changing environments*. Vol. 2. IEEE.

Appendix A

Appendix

A.1 Shunting Model

A.1.1 Main File

```
1  //
2  //  main.c
3  //  main
4  //
5  //  Created by Yingcai Dong on 2016-05-07.
6  //  Copyright 2016 Yingcai Dong. All rights reserved.
7  //
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <math.h>
11 #include "shunting_v1.h"
12 /*==================================*/
13 /*                                  */
14 /* All the tool Function needed */
15 /*                                  */
16 /*==================================*/
17
18 /**
19  * Non-Linear Above Threshold Function
```

```
20  * @param  x : Neuron activity
21  * @return
22  */
23 double UpperBound(double x){
24     double output;
25     if (x > 0) {
26         output = x;
27     }else{
28         output = 0;
29     }
30     return output;
31 }
32
33 /**
34  * Non-linear Below Threshold Function
35  * @param  x : Target/obstacle representive by number
36  * @return
37  */
38 double LowerBound(int x){
39     double output;
40     if (x == 4) {
41         output = E;
42     }else{
43         output = 0;
44     }
45     return output;
46 }
47
48 /**
49  * External Input to i-th Neuron
50  * @param  i         : Vertical position
51  * @param  j         : Horizontal position
52  * @param  workspace : Workspace representived
53         by matrix containing obscacle/target information
54  * @return
55  */
```

```
56 double ExternalInput(int i, int j, int workspace[MATRIX_ROWS][
       MATRIX_COLS]){
57     int output;
58     if (workspace[i][j] == 0 || workspace[i][j] == 7) {
59         output = 0;
60     }else if (workspace[i][j] == 4){
61         output = -E;
62     }else if (workspace[i][j] == 1){
63         output = E;
64     }
65     return output;
66 }
67
68 /**
69  * Sum of External Excitatory Inputs
70  * @param  circlecenter_i    : Robot vertical position
71  * @param  circlecenter_j    : Robot horizontal position
72  * @param  c_neural_activity : Neuron activity information in the
        workspace
73  * @return
74  */
75 double SEEI(int circlecenter_i, int circlecenter_j, double
       c_neural_activity[MATRIX_ROWS][MATRIX_COLS]){
76     // MDF: monotonically decreasing function
77     double distance, MDF;
78     double sum = 0;
79     for (int i = 0; i< MATRIX_ROWS; i+=1) {
80         for (int j = 0; j< MATRIX_COLS; j+=1) {
81             distance = sqrt(pow((i-circlecenter_i), 2) + pow((j-
                   circlecenter_j), 2));
82             if ((distance > 0) && (distance < radious)) {
83                 MDF = mu/distance;
84             } else {
85                 MDF = 0;
86             }
87             sum = sum + MDF * UpperBound(c_neural_activity[i][j]);
```

```
88          }
89      }
90      return sum;
91  }
92
93  /**
94   * Combin Position Information to Single Variable
95   * @param  x : Vertical position
96   * @param  y : Horizontal position
97   * @return
98   */
99  struct position get_postion(int x, int y){
100     struct position position;
101     position.x = x;
102     position.y = y;
103     return position;
104 }
105
106 /**
107  * Find Biggest Neuron activity Within Two Neurons
108  * @param  x                 : First neuron's vertical position
109  * @param  y                 : First neuron's horizontal position
110  * @param  x2                : Second neuron's vertical position
111  * @param  y2                : Second neuron's horizontal position
112  * @param  n_neural_activity : Neuron activity in the workspace
113  * @return                   : Biggest neuron activity & neuron's
         position
114  */
115 struct max get_max(int x, int y, int x2, int y2, double
        n_neural_activity[MATRIX_ROWS][MATRIX_COLS]){
116     struct max _max;
117     if (n_neural_activity[x][y] >= n_neural_activity[x2][y2]) {
118         _max.p = get_postion(x, y);
119         _max.result = n_neural_activity[x][y];
120     }else {
121         _max.p = get_postion(x2, y2);
```

```
122             _max.result = n_neural_activity[x2][y2];
123         }
124         return _max;
125     }
126 
127     /**
128      * Check Valid Position
129      * @param  x       : Vertical position
130      * @param  y       : Horizontal position
131      * @param  origion : Current position
132      * @return         : Valid new position/origional position
133      */
134     struct position boundaryCheck(int x, int y, struct position origion) {
135         if (x < 0 || x > (MATRIX_ROWS-1) || y < 0 || y > (MATRIX_COLS-1)) {
136             return origion;
137         } else {
138             return get_postion(x, y);
139         }
140     }
141 
142     /**
143      * Get Robot's Surround Position
144      * @param  x : Robot vertical position
145      * @param  y : Robot horizontal position
146      * @return   : 8 surround position packed in a variable
147      */
148     struct surround _surround(int x, int y){
149         struct surround get_s;
150         struct position origion = get_postion(x, y);
151         get_s.s1 = boundaryCheck(x-1, y-1, origion);
152         get_s.s2 = boundaryCheck(x+1, y-1, origion);
153         get_s.s3 = boundaryCheck(x, y-1, origion);
154         get_s.s4 = boundaryCheck(x, y+1, origion);
155         get_s.s5 = boundaryCheck(x-1, y, origion);
156         get_s.s6 = boundaryCheck(x+1, y, origion);
157         get_s.s7 = boundaryCheck(x-1, y+1, origion);
```

```
158     get_s.s8 = boundaryCheck(x+1, y+1, origion);
159     return get_s;
160 }
161
162 /**
163  * Find Surround Neuron Contain Max Neuron Activity
164  * @param  x                 : Robot vertical position
165  * @param  y                 : Robot horizontal position
166  * @param  n_neural_activity : Neuron activity in the workspace
167  * @return                   : The position of the neuron containing
168         max neuron activity
169  */
170 struct position maxInSurround(int x, int y,
171         double n_neural_activity[MATRIX_ROWS][MATRIX_COLS]){
172     struct surround get_s;
173     get_s = _surround(x, y);
174     struct max temp1,temp2,temp3,temp4;
175     struct max t1,t2;
176     struct max r;
177     temp1 = get_max(get_s.s1.x, get_s.s1.y, get_s.s2.x, get_s.s2.y,
        n_neural_activity);
178     temp2 = get_max(get_s.s3.x, get_s.s3.y, get_s.s4.x, get_s.s4.y,
        n_neural_activity);
179     temp3 = get_max(get_s.s5.x, get_s.s5.y, get_s.s6.x, get_s.s6.y,
        n_neural_activity);
180     temp4 = get_max(get_s.s7.x, get_s.s7.y, get_s.s8.x, get_s.s8.y,
        n_neural_activity);
181     t1 = get_max(temp1.p.x, temp1.p.y, temp2.p.x, temp2.p.y,
        n_neural_activity);
182     t2 = get_max(temp3.p.x, temp3.p.y, temp4.p.x, temp4.p.y,
        n_neural_activity);
183     r = get_max(t1.p.x, t1.p.y, t2.p.x, t2.p.y, n_neural_activity);
184     if (n_neural_activity[x][y] == r.result) {
185         r.p.x = x;
186         r.p.y = y;
187     }else {
```

```
188         ;
189     }
190     return r.p;
191 }
192 
193 /**
194  * Generate Robot Trajectory
195  * @param input_workspace : Workspace containing obstacle/target
        information
196  * @param SWRESULT       : Robot trajectory as result
197  */
198 void generatePath(int input_workspace[MATRIX_ROWS][MATRIX_COLS], int
    SWRESULT[MATRIX_ROWS][MATRIX_COLS]){
199     float dx;
200     struct position target, c_po, start;
201     double c_neural_activity[MATRIX_ROWS][MATRIX_COLS];
202     double n_neural_activity[MATRIX_ROWS][MATRIX_COLS];
203     // init HW_c_neural_activity
204     for (int i = 0;i < MATRIX_ROWS;i++) {
205         for (int j = 0;j < MATRIX_COLS;j++) {
206             c_neural_activity[i][j] = 0;
207             if (input_workspace[i][j] == 7) {
208                 c_po.x = i;
209                 c_po.y = j;
210                 start.x = i;
211                 start.y = j;
212             }
213             if (input_workspace[i][j] == 1) {
214                 target.x = i;
215                 target.y = j;
216             }
217         }
218     }
219     while (c_po.x != target.x || c_po.y != target.y) {
220         // Caculate all the neurals in the work space
221         for (int p = 0; p < MATRIX_ROWS; p += 1) {
```

```
222             for (int q = 0; q < MATRIX_COLS; q += 1) {
223                 // Shunting model equation
224                 dx = -(A * c_neural_activity[p][q]) + (B -
                        c_neural_activity[p][q]) *(UpperBound(ExternalInput(
                        p, q, input_workspace)) + SEEI(p, q,
                        c_neural_activity)) - (D + c_neural_activity[p][q])
                        *LowerBound(input_workspace[p][q]);
225                 n_neural_activity[p][q] = c_neural_activity[p][q] + (dx
                        * dt);
226             }
227         }
228         c_po = maxInSurround(c_po.x, c_po.y, n_neural_activity);
229         // update neural activity
230         for (int m = 0; m < MATRIX_ROWS; m += 1) {
231             for (int n = 0; n < MATRIX_COLS; n += 1) {
232                 c_neural_activity[m][n] = n_neural_activity[m][n];
233             }
234         }
235         SWRESULT[c_po.x][c_po.y] = 7;
236     }
237
238     for (int s_i = 0; s_i < MATRIX_ROWS; s_i += 1) {
239         for (int s_j = 0; s_j < MATRIX_COLS; s_j += 1) {
240             SWRESULT[s_i][s_j] += input_workspace[s_i][s_j];
241         }
242     }
243     SWRESULT[start.x][start.y] = 9;
244     SWRESULT[target.x][target.y] = 1;
245
246     return;
247 }
248
249 /*==============================*/
250 /*                              */
251 /*      The main function       */
252 /*                              */
```

```
253 /*==========================================*/
254 int main(void) {
255     int input_workspace[32][32] = {
256 {4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4},
257 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
258 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
259 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
260 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
261 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
262 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
263 {4,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
264 {4,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,4,4,4,4,4,4,4,0,0,0,0,0,4},
265 {4,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,4},
266 {4,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,4},
267 {4,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,4},
268 {4,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,4},
269 {4,0,0,0,0,4,4,4,4,4,0,0,0,4,0,0,0,0,0,4,0,0,4,4,4,4,0,0,0,0,0,4},
270 {4,0,0,0,0,0,0,0,0,0,4,0,0,0,4,0,0,0,0,0,4,0,0,0,0,0,4,0,0,0,0,4},
271 {4,0,0,0,0,0,0,0,0,0,4,0,0,0,4,0,0,0,0,0,4,0,0,0,0,1,4,0,0,0,0,4},
272 {4,0,0,0,0,0,0,0,0,0,4,0,0,0,4,0,0,0,0,0,4,0,0,0,0,0,4,0,0,0,0,4},
273 {4,0,0,0,0,0,0,0,0,0,4,0,0,0,4,0,0,0,0,0,4,0,0,4,4,4,4,0,0,0,0,4},
274 {4,0,0,0,0,7,0,0,0,4,0,0,0,4,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,4},
275 {4,0,0,0,0,0,0,0,0,0,4,0,0,0,4,0,0,0,0,0,4,4,4,4,4,4,4,0,0,0,0,4},
276 {4,0,0,0,0,0,0,0,0,0,4,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
277 {4,0,0,0,0,0,0,0,0,0,4,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
278 {4,0,0,0,4,4,4,4,4,4,4,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
279 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
280 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
281 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
282 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
283 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
284 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
285 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
286 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
287 {4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4}
288     };
```

```
289     int returnValue = 0;
290     // declear SWresult matrix
291     int SWRESULT[MATRIX_ROWS][MATRIX_COLS];
292     // init them values
293     for (int m = 0; m < MATRIX_ROWS; m += 1) {
294         for (int n = 0; n < MATRIX_COLS; n += 1) {
295             SWRESULT[m][n] = 0;
296         }
297     }
298     // calculate on SW
299     generatePath(input_workspace, SWRESULT);
300
301     //  print result
302     printf("software =\n");
303     for (int i = 0;i < MATRIX_ROWS;i++) {
304         for (int j = 0;j < MATRIX_COLS;j++) {
305             printf("%d, ", SWRESULT[i][j]);
306         }
307         printf("\n");
308     }
309     return returnValue;
310 }
```

A.1.2 Header File

```
1  //
2  //  shunting_v1.h
3  //  shunting_v1
4  //
5  //  Created by Yingcai Dong on 2016-05-07.
6  //  Copyright 2016 Yingcai Dong. All rights reserved.
7  //
8  #ifndef shunting_v1_h
9  #define shunting_v1_h
10
```

```
11 #define MATRIX_ROWS 32
12 #define MATRIX_COLS 32
13 // the time
14 #define dt 0.01
15 // shunting model parameters
16 #define A 10
17 #define B 1
18 #define D 1
19 #define E 100
20 // The parameter Mu
21 #define mu 1
22 // The radious used to define the connection area
23 #define radious 2//if i change to 3 then it work fine.
24
25 // define some structs
26 struct position{
27     int x;
28     int y;
29 };
30 struct max{
31     struct position p;
32     double result;
33 };
34 struct surround{
35     struct position s1;
36     struct position s2;
37     struct position s3;
38     struct position s4;
39     struct position s5;
40     struct position s6;
41     struct position s7;
42     struct position s8;
43 };
44
45
46 #endif /* shunting_v1_h */
```

A.2 Modified Shunting Model

```
1  //
2  //  main.cpp
3  //  Shunting_modify
4  //
5  //  Created by Yingcai Dong on 2016-05-11.
6  //  Copyright    2016 Yingcai Dong. All rights reserved.
7  //
8
9  #include <stdio.h>
10 #include <stdlib.h>
11
12 #define LEN 32
13
14 /**
15  * Non-Linear Above Threshold Function
16  * @param  x : Neuron activity
17  * @return
18  */
19 float UpperBound(float x){
20     float output;
21     if (x > 0) {
22         output = x;
23     }else{
24         output = 0;
25     }
26     return output;
27 }
28
29 /**
30  * Non-linear Below Threshold Function
31  * @param  x : Target/obstacle representative by number
32  * @return
33  */
34 float LowerBound(float x){
```

```
35      float output;
36      if (x == 4) {
37          output = 100;
38      }else{
39          output = 0;
40      }
41      return output;
42  }
43
44  /**
45   * External Input to i-th Neuron
46   * @param  m : Neural activity
47   * @return
48   */
49  float ExternalInput(float m){
50      int output;
51      if (m == 0) {
52          output = 0;
53      }else if(m == 7) {
54          output = 0;
55      }else if (m == 4){
56          output = -100;
57      }else if (m == 1){
58          output = 100;
59      }
60      return output;
61  }
62
63  /**
64   * Check Invalid Input
65   * @param  x : Vertical position
66   * @param  y : Horizontal position
67   * @return   : Input valid return 1, otherwise 0
68   */
69  int boundary_check(int x, int y){
70      if (x < 0 || x > (LEN-1) || y < 0 || y > (LEN-1)) {
```

```
71          return 0;
72      } else {
73          return 1;
74      }
75  }
76
77  /**
78   * Handle Most of the Path Calculation Task
79   * @param c_act : Current neural activity in workspace
80   * @param work_info : Workspace information
81   */
82  void ComputeCore(float c_act[LEN][LEN], float work_info[LEN][LEN]) {
83      float dx = 0;
84      for (int p = 0; p < LEN; ++p){
85      for (int q = 0; q < LEN; ++q){
86          float sum = 0;
87          for(int i = -1; i < 2; ++i){
88          for(int j = -1; j < 2; ++j){
89          if(boundary_check(i+p, q+j) == 1){
90                  if (i == 0 || j == 0) {
91                  if ((i+j) !=0) {
92                  sum += UpperBound(c_act[i+p][q+j]);
93                  }
94              } else {
95              sum += 0.707107*UpperBound(c_act[i+p][j+q]);
96                  }
97          }
98       }
99       }
100      dx = -(10 * c_act[p][q]) + (1 - c_act[p][q])*(UpperBound(
             ExternalInput(work_info[p][q])) + sum) - (1 + c_act[p][q]) *
             LowerBound(work_info[p][q]);
101      c_act[p][q] += (dx * 0.01);
102     }
103     }
104     return;
```

```
105 }
106
107 /**
108  * Generate Robot Trajectory
109  * @param input_workspace : Workspace information
110  * @param SWRESULT        : Robot trajectory as result
111  */
112 void generatePath(int input_workspace[LEN][LEN],int SWRESULT[LEN][LEN]){
113     // init HW_c_neural_activity
114     float c_activity[LEN][LEN], fworkspace[LEN][LEN];
115     int r_x, r_y, t_x, t_y, s_x, s_y;
116     for (int i = 0;i < LEN;i++) {
117         for (int j = 0;j < LEN;j++) {
118             if (input_workspace[i][j] == 7) {
119                 r_x = i;
120                 r_y = j;
121                 s_x = i;
122                 s_y = j;
123             }
124             if (input_workspace[i][j] == 1) {
125                 t_x = i;
126                 t_y = j;
127             }
128         }
129     }
130     for(int i = 0; i < LEN; i++) {
131         for(int j = 0; j < LEN; j++){
132             c_activity[i][j] = 0;
133             fworkspace[i][j] = (float)input_workspace[i][j];
134         }
135     }
136     while (r_x != t_x || r_y != t_y) {
137         // Caculate all the neurals in the work space
138         ComputeCore(c_activity, fworkspace);
139         // update neural activity
140         float max = c_activity[r_x][r_y];
```

```
141         int tempX = r_x; int tempY = r_y;
142         int i,j;
143         for (i = -1;i < 2; i++) {
144             for (j = -1;j < 2; j++) {
145                 if (boundary_check(r_x+i, r_y+j)) {
146                     if (c_activity[r_x+i][r_y+j] > max) {
147                         max = c_activity[r_x+i][r_y+j];
148                         tempX = r_x+i; tempY = r_y+j;
149                     }
150                 }
151             }
152         }
153         r_x = tempX; r_y = tempY;
154         SWRESULT[r_x][r_y] = 7;
155     }
156     for (int s_i = 0; s_i < LEN; s_i += 1) {
157         for (int s_j = 0; s_j < LEN; s_j += 1) {
158             if(input_workspace[s_i][s_j]==4){
159                 SWRESULT[s_i][s_j] = input_workspace[s_i][s_j];
160             }
161         }
162     }
163     SWRESULT[s_x][s_y] = 9;
164     SWRESULT[t_x][t_y] = 1;
165     return;
166 }
167
168 int main(int argc, char const *argv[]){
169     int res_sw[LEN][LEN];
170     int workspace[LEN][LEN] = {
171 {4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4},
172 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
173 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
174 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
175 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
176 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
```

```
177 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
178 {4,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
179 {4,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,4,4,4,4,4,4,4,0,0,0,0,4},
180 {4,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,4},
181 {4,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,4},
182 {4,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,4},
183 {4,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,4},
184 {4,0,0,0,0,4,4,4,4,4,0,0,0,4,0,0,0,0,0,4,0,0,4,4,4,4,0,0,0,0,4},
185 {4,0,0,0,0,0,0,0,0,4,0,0,0,4,0,0,0,0,0,4,0,0,0,0,0,4,0,0,0,0,4},
186 {4,0,0,0,0,0,0,0,0,4,0,0,0,4,0,0,0,0,0,4,0,0,0,0,1,4,0,0,0,0,4},
187 {4,0,0,0,0,0,0,0,0,4,0,0,0,4,0,0,0,0,0,4,0,0,0,0,0,4,0,0,0,0,4},
188 {4,0,0,0,0,0,0,0,0,4,0,0,0,4,0,0,0,0,0,4,0,0,4,4,4,4,0,0,0,0,4},
189 {4,0,0,0,0,7,0,0,0,4,0,0,0,4,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,4},
190 {4,0,0,0,0,0,0,0,0,4,0,0,0,4,0,0,0,0,0,4,4,4,4,4,4,4,0,0,0,0,4},
191 {4,0,0,0,0,0,0,0,0,4,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
192 {4,0,0,0,0,0,0,0,0,4,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
193 {4,0,0,0,4,4,4,4,4,4,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
194 {4,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
195 {4,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
196 {4,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
197 {4,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
198 {4,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
199 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
200 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
201 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
202 {4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4}
203     };
204     for(int i = 0; i<LEN; i++){
205         for(int j = 0; j<LEN; j++)
206         {
207             res_sw[i][j] = 0;
208         }
209     }
210     for (int i = 0; i < 100; i++) {
211         generatePath(workspace, res_sw);
212     }
```

```
213
214     printf("\nres_sw=\n");
215     for(int last_i = 0; last_i < LEN; last_i++)
216     {
217         for(int last_j = 0; last_j < LEN; last_j++)
218         {
219             printf("%d",res_sw[last_i][last_j]);
220         }
221         printf("\n");
222     }
223     return 0;
224 }
```

A.3 Code for HLS

A.3.1 Header File

```
1  #include <assert.h>
2  #include <ap_axi_sdata.h>
3
4  #define LEN 32
5  #define ELEMENTS 1024
6  typedef ap_axiu<32,4,5,5> AXI_S;
7
8  // function prototypes
9  void HLS_accel (AXI_S in_stream[2*ELEMENTS], AXI_S out_stream[ELEMENTS])
       ;
10
11 // ----------------------------------------------------------------
12 // functions to insert and extract elements from an axi stream
13 // includes conversion to correct data type
14
15 template <typename T, int U, int TI, int TD> T stream_in(ap_axiu <sizeof
       (T)*8,U,TI,TD> const &e){
16 #pragma HLS INLINE
```

```
17          assert(sizeof(T) == sizeof(int));
18          union{
19                  int ival;
20                  T oval;
21          } converter;
22          converter.ival = e.data;
23          T ret = converter.oval;
24
25          volatile ap_uint<sizeof(T)> strb = e.strb;
26          volatile ap_uint<sizeof(T)> keep = e.keep;
27          volatile ap_uint<U> user = e.user;
28          volatile ap_uint<1> last = e.last;
29          volatile ap_uint<TI> id = e.id;
30          volatile ap_uint<TD> dest = e.dest;
31
32          return ret;
33  }
34
35  template <typename T, int U, int TI, int TD> ap_axiu<sizeof(T)*8,U,TI,TD
        > stream_out(T const &v, bool last = false){
36  #pragma HLS INLINE
37          ap_axiu<sizeof(T)*8,U,TI,TD> e;
38
39          assert(sizeof(T) == sizeof(int));
40          union{
41                  int oval;
42                  T ival;
43          } converter;
44          converter.ival = v;
45          e.data = converter.oval;
46
47          // set it to sizeof(T) ones
48          e.strb = -1;
49          e.keep = 15; //e.strb;
50          e.user = 0;
51          e.last = last ? 1 : 0;
```

```
52          e.id = 0;
53          e.dest = 0;
54          return e;
55  }
```

A.3.2 Source File

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   #include "shunting.h"
5
6   /**
7    * Non-Linear Above Threshold Function
8    * @param  x : Neuron activity
9    * @return
10   */
11  float UpperBound(float x){
12  #pragma HLS INLINE
13          float output;
14          if (x > 0) {
15                  output = x;
16          }else{
17                  output = 0;
18          }
19          return output;
20  }
21
22  /**
23   * Non-linear Below Threshold Function
24   * @param  x : Target/obstacle representative by number
25   * @return
26   */
27  float LowerBound(float x){
28  #pragma HLS INLINE
```

```
29          float output;
30          if (x == 4) {
31                  output = 100;
32          }else{
33                  output = 0;
34          }
35          return output;
36  }
37
38  /**
39   * External Input to i-th Neuron
40   * @param  i         : Vertical position
41   * @param  j         : Horizontal position
42   * @param  workspace : Workspace represented by
43   matrix containing obstacle/target information
44   * @return
45   */
46  float ExternalInput(float m){
47  #pragma HLS INLINE
48      int output;
49      if (m == 0) {
50          output = 0;
51      }else if(m == 7) {
52          output = 0;
53      }else if (m == 4){
54          output = -100;
55      }else if (m == 1){
56          output = 100;
57      }
58      return output;
59  }
60
61  /**
62   * Check Invalid Input
63   * @param  x : Vertical position
64   * @param  y : Horizontal position
```

```
65  * @return    : Input valid return 1, otherwise 0
66  */
67 int boundary_check(int x, int y){
68 #pragma HLS INLINE
69     if (x < 0 || x > (ELEMENTS-1) || y < 0 || y > (ELEMENTS-1)) {
70         return 0;
71     } else {
72         return 1;
73     }
74 }
75
76 /**
77  * Handle Most of the Path Calculation Task
78  * @param c_act    : Current neural activity in workspace
79  * @param work_info    : Workspace information
80  * @param out : Updated neural activity in workspace
81  */
82 void ComputeCore_hw(float c_act[LEN][LEN], float work_info[LEN][LEN],
    float out[LEN][LEN]){
83 #pragma HLS INLINE
84 float dx;
85 for (int p = 0; p < LEN; ++p)
86 for (int q = 0; q < LEN; ++q){
87 #pragma HLS PIPELINE II=2
88 float sum = 0;
89 for(int i = -1; i < 2; i++){
90 for(int j = -1; j < 2; j++){
91 #pragma HLS PIPELINE II=2
92         if(boundary_check(i+p, q+j) == 1){
93                 if (i == 0 || j == 0) {
94                         if ((i+j) !=0) {
95                                 sum += UpperBound(c_act[i+p][q+j]);
96                         }
97                 } else {
98                         sum += 0.707107*UpperBound(c_act[i+p][j+q]);
99                 }
```

```
100         }
101 }
102 }
103 dx = -(10 * c_act[p][q]) + (1 - c_act[p][q])      *(UpperBound(
        ExternalInput(work_info[p][q]))+ sum) - (1 + c_act[p][q])*LowerBound
        (work_info[p][q]);
104 out[p][q] = c_act[p][q] + (dx * 0.01);
105 }
106 return;
107 }
108
109 /**
110  * Packing data&Converting data
111  * @param in_stream  : Input data stream
112  * @param out_stream : Output data stream
113  */
114 void wrapped_ComputCore_hw (AXI_S in_stream[2*ELEMENTS], AXI_S
        out_stream[ELEMENTS]){
115 #pragma HLS INLINE
116         float c_act[LEN][LEN];
117         float work_info[LEN][LEN];
118         float out[LEN][LEN];
119
120         assert(sizeof(float)*8 == 32);
121
122         // stream in first matrix
123         for(int i=0; i<LEN; i++)
124                 for(int j=0; j<LEN; j++){
125 #pragma HLS PIPELINE II=1
126                         int k = i*LEN+j;
127                         c_act[i][j] = stream_in<float,4,5,5>(in_stream[k
                            ]);
128                 }
129
130         // stream in second matrix
131         for(int i=0; i<LEN; i++)
```

```
132                 for(int j=0; j<LEN; j++){
133 #pragma HLS PIPELINE II=1
134                         int k = i*LEN+j+ELEMENTS;
135                         work_info[i][j] = stream_in<float,4,5,5>(
                                in_stream[k]);
136                 }
137
138         // do HW Computation
139         ComputeCore_hw(c_act, work_info, out);
140
141         // stream out result matrix
142         for(int i=0; i<LEN; i++)
143                 for(int j=0; j<LEN; j++){
144 #pragma HLS PIPELINE II=1
145                         int k = i*LEN+j;
146                         out_stream[k] = stream_out<float,4,5,5>(out[i][j
                                ],k == (ELEMENTS-1));
147                 }
148         return;
149 }
150
151 /**
152  * Top Function for HLS
153  * @param INPUT_STREAM  : Input stream
154  * @param OUTPUT_STREAM : Output stream
155  */
156 void HLS_accel (AXI_S INPUT_STREAM[2*ELEMENTS], AXI_S OUTPUT_STREAM[
    ELEMENTS]){
157 #pragma HLS INTERFACE s_axilite port=return      bundle=CONTROL_BUS
158 #pragma HLS INTERFACE axis       port=OUTPUT_STREAM
159 #pragma HLS INTERFACE axis       port=INPUT_STREAM
160
161         wrapped_ComputCore_hw (INPUT_STREAM, OUTPUT_STREAM);
162
163         return;
164 }
```

A.3.3 TestBench

```
#include <stdio.h>
#include <stdlib.h>

#include "shunting.h"

/**
 * Non-Linear Above Threshold Function
 * @param  x : Neuron activity
 * @return
 */
float UpperBound2(float x){
        float output;
        if (x > 0) {
                output = x;
        }else{
                output = 0;
        }
        return output;
}

/**
 * Non-linear Below Threshold Function
 * @param  x : Target/obstacle representative by number
 * @return
 */
float LowerBound2(float x){
        float output;
        if (x == 4) {
                output = 100;
        }else{
                output = 0;
        }
        return output;
}
```

```
35
36 /**
37  * Non-linear Below Threshold Function
38  * @param  x : Target/obstacle representative by number
39  * @return
40  */
41 float ExternalInput2(float m){
42     int output;
43     if (m == 0) {
44         output = 0;
45     }else if(m == 7) {
46         output = 0;
47     }else if (m == 4){
48         output = -100;
49     }else if (m == 1){
50         output = 100;
51     }
52     return output;
53 }
54
55 /**
56  * Check Invalid Input
57  * @param  x : Vertical position
58  * @param  y : Horizontal position
59  * @return   : Input valid return 1, otherwise 0
60  */
61 int boundary_check2(int x, int y){
62     if (x < 0 || x > (ELEMENTS-1) || y < 0 || y > (ELEMENTS-1)) {
63         return 0;
64     } else {
65         return 1;
66     }
67 }
68
69 /**
70  * Handle Most of the Path Calculation Task
```

```
71  * @param c_act : Current neural activity in workspace
72  * @param work_info : Workspace information
73  */
74 void ComputeCore_sw(float c_act[LEN][LEN],float work_info[LEN][LEN],
       float out[LEN][LEN]){
75 float dx;
76 for (int p = 0; p < LEN; ++p)
77 for (int q = 0; q < LEN; ++q){
78 float sum = 0;
79 for(int i = -1; i < 2; i++){
80 for(int j = -1; j < 2; j++){
81         if(boundary_check2(i+p, q+j) == 1){
82                 if (i == 0 || j == 0) {
83                         if ((i+j) !=0) {
84                                 sum += UpperBound2(c_act[i+p][q+j]);
85                         }
86                 } else {
87                         sum += 0.707107*UpperBound2(c_act[i+p][j+q]);
88                 }
89         }
90 }
91 }
92 dx = -(10 * c_act[p][q])
93 + (1 - c_act[p][q])*(UpperBound2(ExternalInput2(work_info[p][q]))+ sum)
94 - (1 + c_act[p][q])*LowerBound2(work_info[p][q]);
95 out[p][q] = c_act[p][q] + (dx * 0.01);
96 }
97 return;
98 }
99
100
101 int main(void){
102         int err;
103         float activity[LEN][LEN];
104         float workspace_float[LEN][LEN];
105         float result_sw[LEN][LEN];
```

```
106         float result_hw[LEN][LEN];
107
108         int workspace[32][32] ={
109 {4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4},
110 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
111 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
112 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
113 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
114 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
115 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
116 {4,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
117 {4,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,4,4,4,4,4,4,4,0,0,0,0,0,4},
118 {4,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,4},
119 {4,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,4},
120 {4,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,4},
121 {4,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,4},
122 {4,0,0,0,0,4,4,4,4,4,0,0,0,4,0,0,0,0,0,4,0,0,4,4,4,4,0,0,0,0,0,4},
123 {4,0,0,0,0,0,0,0,0,0,4,0,0,0,4,0,0,0,0,0,4,0,0,0,0,0,4,0,0,0,0,0,4},
124 {4,0,0,0,0,0,0,0,0,0,4,0,0,0,4,0,0,0,0,0,4,0,0,0,0,1,4,0,0,0,0,0,4},
125 {4,0,0,0,0,0,0,0,0,0,4,0,0,0,4,0,0,0,0,0,4,0,0,0,0,0,4,0,0,0,0,0,4},
126 {4,0,0,0,0,0,0,0,0,0,4,0,0,0,4,0,0,0,0,0,4,0,0,0,4,4,4,4,0,0,0,0,4},
127 {4,0,0,0,0,7,0,0,0,4,0,0,0,4,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,4},
128 {4,0,0,0,0,0,0,0,0,0,4,0,0,0,4,0,0,0,0,0,4,4,4,4,4,4,4,0,0,0,0,4},
129 {4,0,0,0,0,0,0,0,0,0,4,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
130 {4,0,0,0,0,0,0,0,0,0,4,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
131 {4,0,0,0,4,4,4,4,4,4,4,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
132 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
133 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
134 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
135 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
136 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
137 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
138 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
139 {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
140 {4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4}
141                 };
```

```
142
143         /** Matrix Initiation */
144         for(int i = 0; i<LEN; i++){
145                 for(int j = 0; j<LEN; j++){
146                         activity[i][j] = 0.0;
147                         workspace_float[i][j] = (float)(workspace[i][j])
                                ;
148                 }
149         }
150
151         // prepare data
152         AXI_S inp_stream[2*ELEMENTS];
153         AXI_S out_stream[ELEMENTS];
154
155         assert(sizeof(float)*8 == 32);
156         // stream in the first input  matrix
157         for(int i=0; i<LEN; i++)
158                 for(int j=0; j<LEN; j++){
159                         int k = i*LEN+j;
160                         inp_stream[k] = stream_out<float,4,5,5>(activity
                                [i][j],0);
161                 }
162
163         // stream in the second input  matrix
164         for(int i=0; i<LEN; i++)
165                 for(int j=0; j<LEN; j++){
166                         int k = i*LEN+j;
167                         inp_stream[k+ELEMENTS] = stream_out<float
                                ,4,5,5>(workspace_float[i][j],k == (ELEMENTS
                                -1));
168                 }
169
170         HLS_accel(inp_stream, out_stream);
171
172         // extract the output matrix from the out stream
173         for(int i=0; i<LEN; i++)
```

```
174                 for(int j=0; j<LEN; j++){
175                         int k = i*LEN+j;
176                         result_hw[i][j] = stream_in<float,4,5,5>(
                                out_stream[k]);
177                 }
178
179
180         /* reference software ComputeCore */
181         ComputeCore_sw(activity, workspace_float, result_sw);
182
183         /** Result comparison */
184         err = 0;
185         for (int i = 0; (i<LEN); i++){
186                 for (int j = 0; (j<LEN); j++){
187                         printf("%f, ", result_sw[i][j]);
188                         if (result_sw[i][j] - result_hw[i][j] > 0.000002
                                ||
189                         result_sw[i][j] - result_hw[i][j] < -0.000002)
190                                 err++;
191                 }printf("\n");
192         }printf("\n");
193         if (err == 0)
194                 printf("Test successful!\r\n");
195         else
196                 printf("Test failed!\r\n");
197
198         return err;
199 }
```

A.4 Software Design Code

A.4.1 Main File

```
1 #include <stdio.h>
2 #include <stdlib.h>
```

```
3  #include "platform.h"
4  #include "xparameters.h"
5  #include "xtmrctr.h"
6  #include "xaxidma.h"
7  #include "shunting_drive.h"
8  #include "xil_cache.h"
9
10 #define XPAR_AXI_TIMER_DEVICE_ID                    (
       XPAR_AXI_TIMER_0_DEVICE_ID)
11
12 // TIMER Instance
13 XTmrCtr timer_dev;
14
15 // AXI DMA Instance
16 XAxiDma AxiDma;
17
18
19 int init_dma(){
20         XAxiDma_Config *CfgPtr;
21         int status;
22
23         CfgPtr = XAxiDma_LookupConfig( (XPAR_AXI_DMA_0_DEVICE_ID) );
24         if(!CfgPtr){
25                 print("Error looking for AXI DMA config\n\r");
26                 return XST_FAILURE;
27         }
28         status = XAxiDma_CfgInitialize(&AxiDma, CfgPtr);
29         if(status != XST_SUCCESS){
30                 print("Error initializing DMA\n\r");
31                 return XST_FAILURE;
32         }
33         //check for scatter gather mode
34         if(XAxiDma_HasSg(&AxiDma)){
35                 print("Error DMA configured in SG mode\n\r");
36                 return XST_FAILURE;
37         }
```

```
38          /* Disable interrupts, we use polling mode */
39          XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,
                XAXIDMA_DEVICE_TO_DMA);
40          XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,
                XAXIDMA_DMA_TO_DEVICE);
41
42          // Reset DMA
43          XAxiDma_Reset(&AxiDma);
44          while (!XAxiDma_ResetIsDone(&AxiDma)) {}
45
46          return XST_SUCCESS;
47  }
48
49  int boundary_check(int x, int y){
50      if (x < 0 || x > (LEN-1) || y < 0 || y > (LEN-1)) {
51          return 0;
52      } else {
53          return 1;
54      }
55  }
56
57
58  int main(int argc, char **argv){
59          int i, j, k;
60          int err=0;
61          int status;
62          float c_act[LEN][LEN];
63          float work_info[LEN][LEN];
64          int res_hw[LEN][LEN];
65          int res_sw[LEN][LEN];
66
67          int workspace[LEN][LEN] ={
68  {4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4},
69  {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
70  {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
71  {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
```

```
72  {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
73  {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
74  {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
75  {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
76  {4,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
77  {4,0,0,0,4,4,0,0,0,0,0,4,0,0,0,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,4},
78  {4,0,0,0,4,4,0,0,0,0,0,4,0,0,0,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,4},
79  {4,0,0,0,4,4,0,0,0,0,0,4,0,0,0,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,4},
80  {4,0,0,0,4,4,4,0,0,0,0,4,0,0,0,0,4,4,4,0,4,4,4,4,4,4,4,4,0,0,4},
81  {4,0,0,0,0,4,4,0,0,0,0,4,0,0,0,0,4,0,0,0,4,0,0,0,0,0,0,0,0,0,4},
82  {4,0,0,0,0,0,4,4,0,0,0,4,0,0,0,0,4,0,4,4,4,0,0,0,0,0,0,0,0,0,4},
83  {4,0,0,0,0,0,4,4,0,0,0,4,0,0,0,0,4,0,4,1,4,0,0,0,0,0,0,0,0,0,4},
84  {4,0,0,0,0,0,0,4,4,0,0,4,0,0,0,0,4,0,4,0,4,0,0,0,0,0,0,0,0,0,4},
85  {4,0,0,0,0,0,0,0,4,4,0,0,4,0,0,0,0,4,0,0,0,4,0,0,0,0,0,0,0,0,4},
86  {4,0,0,0,0,0,0,0,0,4,4,0,4,4,4,4,4,4,4,4,4,4,0,0,0,0,0,0,0,0,4},
87  {4,0,0,0,0,0,0,0,0,4,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
88  {4,0,0,0,0,0,0,0,0,0,4,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
89  {4,0,0,0,0,0,0,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,0,0,4},
90  {4,0,0,0,0,0,0,0,0,0,0,4,4,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
91  {4,0,0,0,0,0,0,0,0,0,0,4,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
92  {4,0,0,0,0,0,0,0,0,0,4,4,0,4,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
93  {4,0,0,0,0,0,0,0,0,0,4,0,7,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
94  {4,0,0,0,0,0,0,0,0,4,4,0,0,0,4,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
95  {4,0,0,0,0,0,0,0,0,4,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
96  {4,0,0,0,0,0,0,0,0,4,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
97  {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
98  {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
99  {4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4}
100         };
101
102         unsigned int dma_size = SIZE * sizeof(float);
103
104     float acc_factor;
105         unsigned int init_time, curr_time, calibration;
106         unsigned int begin_time;
107         unsigned int end_time;
```

```
108         unsigned int run_time_sw = 0;
109         unsigned int run_time_hw = 0;
110
111         init_platform();
112
113         xil_printf("\r**********************************************\n\
                r");
114         xil_printf("\r 32x32 MATRIX workspace shunting model path
                planning -> AXI_DMA -> ARM_ACP \n\r");
115         xil_printf("\r**********************************************\n\
                n\r");
116
117         /* ************************************************************/
118         // Init DMA
119         status = init_dma();
120         if(status != XST_SUCCESS){
121                 print("\rError: DMA init failed\n");
122                 return XST_FAILURE;
123         }
124         print("\r\nDMA Init done\n\r");
125
126         /* ************************************************************/
127         // Setup HW timer
128         status = XTmrCtr_Initialize(&timer_dev, XPAR_AXI_TIMER_DEVICE_ID
                );
129         if(status != XST_SUCCESS){
130                 print("\rError: timer setup failed\n");
131                 //return XST_FAILURE;
132         }
133         XTmrCtr_SetOptions(
134         &timer_dev, XPAR_AXI_TIMER_DEVICE_ID, XTC_ENABLE_ALL_OPTION);
135
136         // Calibrate HW timer
137         XTmrCtr_Reset(&timer_dev, XPAR_AXI_TIMER_DEVICE_ID);
138         init_time = XTmrCtr_GetValue(&timer_dev,
                XPAR_AXI_TIMER_DEVICE_ID);
```

```
139         curr_time = XTmrCtr_GetValue(&timer_dev,
                XPAR_AXI_TIMER_DEVICE_ID);
140         calibration = curr_time - init_time;
141
142         // Loop measurement
143         XTmrCtr_Reset(&timer_dev, XPAR_AXI_TIMER_DEVICE_ID);
144         begin_time = XTmrCtr_GetValue(&timer_dev,
                XPAR_AXI_TIMER_DEVICE_ID);
145         for (i = 0; i< 1; i++);
146         end_time = XTmrCtr_GetValue(&timer_dev, XPAR_AXI_TIMER_DEVICE_ID
                );
147         run_time_sw = end_time - begin_time - calibration;
148         xil_printf("\rLoop 1 time is %d cycles.\r\n", run_time_sw);
149
150         /* ************************************************************/
151         // input data Initiation
152          for(i = 0; i<LEN; i++)
153                 for(j = 0; j<LEN; j++){
154                         res_sw[i][j] = 0;
155                         res_hw[i][j] = 0;
156                 }
157         /** End of Initiation */
158         /* ************************************************************/
159         // call the software version of the function
160         xil_printf("\rRunning shunting model in SW\n");
161         XTmrCtr_Reset(&timer_dev, XPAR_AXI_TIMER_DEVICE_ID);
162         begin_time = XTmrCtr_GetValue(&timer_dev,
                XPAR_AXI_TIMER_DEVICE_ID);
163
164         generatePath(workspace,res_sw);
165
166         end_time = XTmrCtr_GetValue(&timer_dev, XPAR_AXI_TIMER_DEVICE_ID
                );
167         run_time_sw = end_time - begin_time - calibration;
168         xil_printf("\r\nTotal run time for SW on Processor is %d cycles
                .\r\n",
```

```
169                     run_time_sw);
170
171         /* ********************************************************/
172         // call the HW accelerator
173         XTmrCtr_Reset(&timer_dev, XPAR_AXI_TIMER_DEVICE_ID);
174         begin_time = XTmrCtr_GetValue(&timer_dev,
                XPAR_AXI_TIMER_DEVICE_ID);
175         // Setup the HW Accelerator
176         int p,q;
177         int r_x, r_y, t_x, t_y, s_x, s_y;
178         for (p = 0;p < 32;p++) {
179                 for (q = 0;q < 32;q++) {
180                         if (workspace[p][q] == 7) {
181                                 r_x = p; r_y = q; s_x = p; s_y = q;
182                         }
183                         if (workspace[p][q] == 1) {
184                                 t_x = p; t_y = q;
185                         }
186                         c_act[p][q] = 0;
187                         work_info[p][q] = (float)workspace[p][q];
188                 }
189         }
190         Setup_HW_Accelerator(c_act, work_info, c_act, dma_size);
191
192         while(r_x != t_x || r_y != t_y) {
193         int speed;
194         for(speed = 0; speed < SPEED; speed++){
195         Start_HW_Accelerator();
196         Run_HW_Accelerator(c_act, work_info, c_act, dma_size);}
197         Xil_DCacheFlushRange((unsigned int)c_act,dma_size);
198
199         float max = c_act[r_x][r_y];
200         int temp_x = r_x; int temp_y = r_y;
201         for(p = -1; p < 2; p++){
202                 for(q = -1; q < 2; q++){
203                         if(boundary_check(r_x+p, r_y+q)){
```

```
204                             if(c_act[r_x+p][r_y+q] > max){
205                                     max = c_act[r_x+p][r_y+q];
206                                     temp_x = r_x+p; temp_y = r_y+q;
207                             }
208                     }
209             }
210     }
211     r_x = temp_x; r_y = temp_y;
212     res_hw[temp_x][temp_y] = 7;
213     }
214     for (p = 0; p < LEN; p += 1) {
215             for (q = 0; q < LEN; q += 1) {
216                     if(workspace[p][q] == 4){
217                             res_hw[p][q] = workspace[p][q];
218                     }
219             }
220     }
221     res_hw[s_x][s_y] = 9;
222     res_hw[t_x][t_y] = 1;
223     end_time = XTmrCtr_GetValue(&timer_dev, XPAR_AXI_TIMER_DEVICE_ID
        );
224     run_time_hw = end_time - begin_time - calibration;
225     xil_printf(
226                     "\rTotal run time for AXI DMA + HW accelerator
                            is %d cycles.\r\n",
227                     run_time_hw);
228
229     /* ********************************************** */
230     //Compare the results from sw and hw
231
232     for (i = 0; i < LEN; i++)
233             for (j = 0; j < LEN; j++)
234                     if (res_hw[i][j] != res_sw[i][j]) {
235                             err += 1;
236                             printf("\nposition: x = %d, y = %d\n", i
                                    , j);
```

```
237                         }
238
239         // HW vs. SW speedup factor
240         acc_factor = (float) run_time_sw / (float) run_time_hw;
241         xil_printf("Acceleration factor: %d.%d\n\n",
242                         (int) acc_factor, (int) (acc_factor * 1000) %
                                1000);
243
244         if (err == 0){
245                 print("\rSW and HW results match!\n\r");
246         }else{
247                 printf("ERROR: results mismatch %d\n", err);
248         }
249
250         int last_i, last_j;
251
252         printf("\nres_sw = \n");
253         for(last_i = 0; last_i < LEN; last_i++){
254                 for(last_j = 0; last_j < LEN; last_j++){
255                         printf("%d",res_sw[last_i][last_j]);
256                 }
257                 printf("\n");
258         }
259         printf("\n\n");
260
261         printf("\nres_hw = \n");
262         for(last_i = 0; last_i < LEN; last_i++){
263                 for(last_j = 0; last_j < LEN; last_j++){
264                         printf("%d",res_hw[last_i][last_j]);
265                 }
266                 printf("\n");
267         }
268         printf("\n\n");
269     cleanup_platform();
270     return 0;
271 }
```

A.4.2 HLS Hardware Driver File

```
#include <stdio.h>
#include <stdlib.h>
#include "platform.h"
#include "xparameters.h"
#include "xscugic.h"
#include "xaxidma.h"
#include "xhls_accel.h"
#include "shunting_drive.h"
#include "xil_printf.h"


volatile static int RunExample = 0;
volatile static int ResultExample = 0;

XHls_accel shunting_dev;

XHls_accel_Config shunting_config = {
        0,
        XPAR_HLS_ACCEL_0_S_AXI_CONTROL_BUS_BASEADDR
};

//Interrupt Controller Instance
XScuGic ScuGic;

// AXI DMA Instance
extern XAxiDma AxiDma;


int XMmultSetup(){
        return XHls_accel_CfgInitialize(&shunting_dev,&shunting_config);
}

void XShuntingStart(void *InstancePtr){
        XHls_accel *pExample = (XHls_accel *)InstancePtr;
```

```
35          XHls_accel_InterruptEnable(pExample,1);
36          XHls_accel_InterruptGlobalEnable(pExample);
37          XHls_accel_Start(pExample);
38  }
39
40
41  void XShuntingIsr(void *InstancePtr){
42          XHls_accel *pExample = (XHls_accel *)InstancePtr;
43
44          //Disable the global interrupt
45          XHls_accel_InterruptGlobalDisable(pExample);
46          //Disable the local interrupt
47          XHls_accel_InterruptDisable(pExample,0xffffffff);
48
49          // clear the local interrupt
50          XHls_accel_InterruptClear(pExample,1);
51
52          ResultExample = 1;
53          // restart the core if it should run again
54          if(RunExample){
55                  XShuntingStart(pExample);
56          }
57  }
58
59  int XShuntingSetupInterrupt(){
60          //This functions sets up the interrupt on the ARM
61          int result;
62          XScuGic_Config *pCfg = XScuGic_LookupConfig(
                XPAR_SCUGIC_SINGLE_DEVICE_ID);
63          if (pCfg == NULL){
64                  print("Interrupt Configuration Lookup Failed\n\r");
65                  return XST_FAILURE;
66          }
67          result = XScuGic_CfgInitialize(&ScuGic,pCfg,pCfg->CpuBaseAddress
                );
68          if(result != XST_SUCCESS){
```

```
69                  return result;
70          }
71          // self test
72          result = XScuGic_SelfTest(&ScuGic);
73          if(result != XST_SUCCESS){
74                  return result;
75          }
76          // Initialize the exception handler
77          Xil_ExceptionInit();
78          // Register the exception handler
79          //print("Register the exception handler\n\r");
80          Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT, (
                Xil_ExceptionHandler)XScuGic_InterruptHandler,&ScuGic);
81          //Enable the exception handler
82          Xil_ExceptionEnable();
83          // Connect the Adder ISR to the exception table
84          //print("Connect the Adder ISR to the Exception handler table\n\
                r");
85          result = XScuGic_Connect(&ScuGic
86          ,XPAR_FABRIC_HLS_ACCEL_0_INTERRUPT_INTR
87          ,(Xil_InterruptHandler)XShuntingIsr,&shunting_dev);
88          if(result != XST_SUCCESS){
89                  return result;
90          }
91          //print("Enable the Adder ISR\n\r");
92          XScuGic_Enable(&ScuGic,XPAR_FABRIC_HLS_ACCEL_0_INTERRUPT_INTR);
93          return XST_SUCCESS;
94  }
95
96  int Setup_HW_Accelerator(float c_act[LEN][LEN], float work_info[LEN][LEN
        ], float res_hw[LEN][LEN], int dma_size){
97          int status = XMmultSetup();
98          if(status != XST_SUCCESS){
99                  print("Error: example setup failed\n");
100                 return XST_FAILURE;
101         }
```

```
102         status =  XShuntingSetupInterrupt();
103         if(status != XST_SUCCESS){
104                 print("Error: interrupt setup failed\n");
105                 return XST_FAILURE;
106         }
107
108         XShuntingStart(&shunting_dev);
109
110         //flush the cache
111         Xil_DCacheFlushRange((unsigned int)c_act,dma_size);
112         Xil_DCacheFlushRange((unsigned int)work_info,dma_size);
113         Xil_DCacheFlushRange((unsigned int)res_hw,dma_size);
114
115         return 0;
116 }
117
118 //================================================
119 //==================code for software result part1
120 float UpperBound2(float x){
121         float output;
122         if (x > 0) {
123                 output = x;
124         }else{
125                 output = 0;
126         }
127         return output;
128 }
129
130 float LowerBound2(float x){
131         float output;
132         if (x == 4) {
133                 output = 100;
134         }else{
135                 output = 0;
136         }
137         return output;
```

```
138 }
139
140 float ExternalInput2(float m){
141     int output;
142     if (m == 0) {
143         output = 0;
144     }else if(m == 7) {
145         output = 0;
146     }else if (m == 4){
147         output = -100;
148     }else if (m == 1){
149         output = 100;
150     }
151     return output;
152 }
153
154 int boundary_check2(int x, int y){
155     if (x < 0 || x > (LEN-1) || y < 0 || y > (LEN-1)) {
156         return 0;
157     } else {
158         return 1;
159     }
160 }
161
162 void shunting_sf_ref(float a[LEN][LEN], float b[LEN][LEN]) {
163 int p,q,i,j;
164 float dx = 0;
165 for (p = 0; p < LEN; ++p){
166 for (q = 0; q < LEN; ++q){
167 float sum = 0;
168 for(i = -1; i < 2; ++i){
169         for(j = -1; j < 2; ++j){
170                 if(boundary_check2(i+p, q+j) == 1){
171                         if (i == 0 || j == 0) {
172                                 if ((i+j) !=0) {
173                                         sum += UpperBound2(a[i+p][q+j]);
```

```
174                                     }
175                             } else {
176                                     sum += 0.707107*UpperBound2(a[i+p][j+q])
                                            ;
177                             }
178                     }
179             }
180     }
181     dx = -(10 * a[p][q]) + (1 - a[p][q])*(UpperBound2(ExternalInput2(b[p][q
            ])) + sum) - (1 + a[p][q])*LowerBound2(b[p][q]);
182     a[p][q] += (dx * 0.01);
183     }
184     }
185     return;
186     }
187     //==================code for software result part1 ends here
188     //=============================================================
189
190     void Start_HW_Accelerator(){
191             int status = XMmultSetup();
192             if(status != XST_SUCCESS){
193                     print("Error: example setup failed\n");
194                     return XST_FAILURE;
195             }
196             status =  XShuntingSetupInterrupt();
197             if(status != XST_SUCCESS){
198                     print("Error: interrupt setup failed\n");
199                     return XST_FAILURE;
200             }
201
202             XShuntingStart(&shunting_dev);
203     }
204
205     int Run_HW_Accelerator(float c_act[LEN][LEN], float work_info[LEN][LEN],
            float res_hw[LEN][LEN], int dma_size){
206             //transfer c_act to the Vivado HLS block
```

```
207         int status = XAxiDma_SimpleTransfer(&AxiDma, (unsigned int)
                c_act, dma_size, XAXIDMA_DMA_TO_DEVICE);
208         if (status != XST_SUCCESS) {
209                 //print("Error: DMA transfer to Vivado HLS block failed\
                        n");
210                 return XST_FAILURE;
211         }
212         /* Wait for transfer to be done */
213         while (XAxiDma_Busy(&AxiDma, XAXIDMA_DMA_TO_DEVICE)) ;
214
215         status = XAxiDma_SimpleTransfer(&AxiDma, (unsigned int)
                work_info, dma_size, XAXIDMA_DMA_TO_DEVICE);
216         if (status != XST_SUCCESS) {
217                 //print("Error: DMA transfer to Vivado HLS block failed\
                        n");
218                 return XST_FAILURE;
219         }
220         /* Wait for transfer to be done */
221         while (XAxiDma_Busy(&AxiDma, XAXIDMA_DMA_TO_DEVICE)) ;
222
223         //get results from the Vivado HLS block
224         status = XAxiDma_SimpleTransfer(&AxiDma, (unsigned int) res_hw,
                dma_size,
225                                 XAXIDMA_DEVICE_TO_DMA);
226         if (status != XST_SUCCESS) {
227                 //print("Error: DMA transfer from Vivado HLS block
                        failed\n");
228                 return XST_FAILURE;
229         }
230         /* Wait for transfer to be done */
231         while (XAxiDma_Busy(&AxiDma, XAXIDMA_DMA_TO_DEVICE)) ;
232
233         //poll the DMA engine to verify transfers are complete
234         /* Waiting for data processing */
235         /* While this wait operation, the following action would be done
236          * First: Second matrix will be sent.
```

```
237             * After: Multiplication will be compute.
238             * Then: Output matrix will be sent from the accelerator to DDR
                    and
239             * it will be stored at the base address that you set in the
                    first SimpleTransfer
240             */
241         while ((XAxiDma_Busy(&AxiDma, XAXIDMA_DEVICE_TO_DMA)) || (
                XAxiDma_Busy(&AxiDma, XAXIDMA_DMA_TO_DEVICE))) ;
242
243         return 0;
244 }
245
246 //=====================================================
247 //==================code for software result part2
248 void generatePath(int input_workspace[LEN][LEN],int SWRESULT[LEN][LEN]){
249         // init HW_c_neural_activity
250         float c_activity[LEN][LEN], fworkspace[LEN][LEN];
251         int r_x, r_y, t_x, t_y, s_x, s_y;
252         int i,j;
253         for (i = 0;i < LEN;i++) {
254                 for (j = 0;j < LEN;j++) {
255                         if (input_workspace[i][j] == 7) {
256                                 r_x = i;
257                                 r_y = j;
258                                 s_x = i;
259                                 s_y = j;
260                         }
261                         if (input_workspace[i][j] == 1) {
262                                 t_x = i;
263                                 t_y = j;
264                         }
265                 }
266         }
267         for(i = 0; i < LEN; i++) {
268                 for(j = 0; j < LEN; j++){
269                         c_activity[i][j] = 0;
```

```
270                     fworkspace[i][j] = (float)input_workspace[i][j];
271             }
272         }
273         while (r_x != t_x || r_y != t_y) {
274             // Caculate all the neurals in the work space
275             int speed;
276             for(speed = 0; speed < SPEED; speed++){
277                     shunting_sf_ref(c_activity, fworkspace);}
278             // update neural activity
279             float max = c_activity[r_x][r_y];
280             int tempX = r_x; int tempY = r_y;
281             int i,j;
282         for (i = -1;i < 2; i++) {
283             for (j = -1;j < 2; j++) {
284                 if (boundary_check2(r_x+i, r_y+j)) {
285                     if (c_activity[r_x+i][r_y+j] > max) {
286                         max = c_activity[r_x+i][r_y+j];
287                         tempX = r_x+i; tempY = r_y+j;
288                     }
289                 }
290             }
291         }
292         r_x = tempX; r_y = tempY;
293         SWRESULT[r_x][r_y] = 7;
294
295         }
296         int s_i,s_j;
297                 for (s_i = 0; s_i < LEN; s_i += 1) {
298                         for (s_j = 0; s_j < LEN; s_j += 1) {
299                                 if(input_workspace[s_i][s_j]==4){
300                                         SWRESULT[s_i][s_j] =
                                             input_workspace[s_i][s_j];
301                                 }
302                         }
303                 }
304                 SWRESULT[s_x][s_y] = 9;
```

```
305                     SWRESULT[t_x][t_y] = 1;
306             return;
307     }
```

A.4.3 Header File

```
1
2   #ifndef SHUNTING_DIVE_H
3   #define SHUNTING_DIVE_H
4
5   #define LEN     32
6   #define SIZE    ((LEN)*(LEN))
7   #define SPEED 10
8
9   int Setup_HW_Accelerator(float c_act[LEN][LEN],float work_info[LEN][LEN
        ], float res_hw[LEN][LEN], int dma_size);
10
11  int Run_HW_Accelerator(float c_act[LEN][LEN], float work_info[LEN][LEN],
         float res_hw[LEN][LEN], int dma_size);
12
13  void Start_HW_Accelerator();
14
15  void generatePath(int input_workspace[LEN][LEN],int SWRESULT[LEN][LEN]);
16
17  #endif
```