University of Alberta

AN FPGA-BASED ACCELERATION PLATFORM FOR THE AUCTION ALGORITHM

by

Pengfei Zhu

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer, Microelectronics Devices, Circuits and Systems

Department of Electrical and Computer Engineering

Fall 2012
Edmonton, Alberta

Abstract

Auction algorithms have been applied in various linear network problems, such as assignment, transportation, max-flow and shortest path problem. The inherent parallel characteristics of these algorithms are well suited for Field-Programmable Gate Array (FPGA) hardware implementation. In this work, we focus on the acceleration of auction algorithms to solve the assignment problem.

The main contribution is to set up a flexible platform to generate efficient and extensible application-based hardware acceleration platform. It aims at solving both symmetric and asymmetric assignment problem. Our experimental results show that a 10X speedup can be achieved using 128 Processing Elements for a problem size of 500.

Acknowledgements

I would like to extend my sincere thanks to my supervisor, Dr. Yu Hu, and my co-supervisor Dr. Hua Li for his support and supervision. Their continuous advice, guidance and encouragement have been instrumental in making this work a success.

I also thank the members of my thesis committee, Dr. Bruce Cockburn and Dr. Zhijun Qiu, for their valuable comments and advice in improving this thesis.

Many thanks to my colleagues and friends at ECE department for their help and support in various aspects of my research. Moreover, they played an important role of my wonderful life here in Edmonton, Alberta.

Finally, my sincere appreciation and gratitude go to my family for their constant love, support and encouragement throughout my stay in Edmonton.

Table of Contents

List of Tables

List of Figures

List of Symbols

W	Channel width.
F_s	The number of wires to which each incoming wire con connect in a switch block.
F_c	The number of wires in each channel to which a logic block input or output pin can connect.

Acronyms

CPU Central Processing Unit

FPGA Field-Programmable Gate Array

HDL Hardware Description Language

RTL Register-Transfer Level

IP Intellectual Property

ECO Engineering Change Order

GPU Graphics Processing Unit

CLB Configurable Logic Block

SB Switch Block

CB Connection Block

BLE Basic Logic Element

LUT Look-Up Table

RAM Random-Access Memory

BRAM Block RAM

SoC System on Chip

FIFO First-In First-Out

DCM Digital Clock Managers

PCI Peripheral Component Interconnect

MAC Media Access Control

DSP Digital Signal Processing

AP Assignment Problem

HA Hungarian Algorithm

FAA Forward Auction Algorithm

RAA Reverse Auction Algorithm

FRAA Forward/Reverse Auction Algorithm

PE Processing Element

CAD Computer-Aided Design

PEIe Peripheral Component Interconnect Express

SIMD Single-Instruction Multiple-Data

MIMD Multiple-Instruction Multiple-Data

Chapter 1

Introduction

1.1 Background

The assignment problem (AP) is one of the fundamental combinatorial optimization problems, which tries to find the maximum weight matching in a weighted bipartite graph. It has a wide range of applications in most fields. Naturally, it is useful to solve such problems as assigning employees to tasks or machines to production jobs. Furthermore, it is also a key building block in several applications of network modeling real world. For example, in transportation system, it is a tremendous task to allocate and route the traffic in a large city, like New York, where hundreds of thousands of vehicles are running around. In integrated circuits, there are millions of gates and wires needed to be routed to meet the needs of performance. Thus it is essential to develop an efficient implementation of assignment problem to deal with these ambitious tasks.

Due to its importance, the assignment problem has been studied extensively in the literature [1] [2] [3] [4] [5]. Among various approaches, the auction [6] and Hungarian [5] [1] algorithms are two efficient methods to solve the problem optimally. Compared to the Hungarian algorithm (HA), the auction algorithm exhibits a natural computational parallelism, which is well suited for hardware acceleration. Although they have various forms, different auction-based algorithms share similar working flow as [7] [8]. Therefore, although we focus on the method in [8] in this thesis, the approach can be extended to other auction algorithms with little efforts.

Recently, along with the rapid technology development, FPGAs have found their place in high-performance computation. Compared to the fixed structure of Graphic Processing Unit (GPU), FPGAs are more flexible and have better data processing granularity. They are essentially high-density arrays of uncommitted logic so that developers can directly build up module-to-module hardware infrastructure and then trade-off performance by partitioning applications differently [9].

In existing work of FPGA implementations to solve the assignment problem [10] [11], Hung and Wang proposed a computing system based on a recurrent neural network. It takes a few seconds to solve 100×100 assignment problem, which is not efficient enough to meet the practical needs. In contrast, our proposed architecture can deal with a 500×500 problem in milliseconds. Other work related to hardware implementations of the auction algorithm can be covered in Chapter 4. Recently, Vasconcelos and Rosenhahn [12] used a GPU to accelerate the auction algorithm and achieved 7X with problems of size 400 to 4000.

1.2 Problem Formulation

The assignment problem is important in many practical contexts, but it is also of great theoretical importance. Despite its simplicity, it embodies a fundamental linear programming structure. One of the most important types of linear programming problem, the minimum cost network flow problem, can be reduced to the assignment problem by means of a simple reformulation. Thus, any method for solving the assignment problem can be generalized to solve the minimum cost flow problem. For this reason, the assignment problem has served as a convenient starting point for important algorithmic ideas in linear programming. Now we will begin by defining assignment problem in a mathematical way.

In graph theoretic terms, the assignment problem aims at finding a maximum weight matching of a weighted bipartite graph. In general, given n agents and m objects ($n \leq m$ in general), and a benefit a_{ij} associated with the assignment of object j to agent i, the assignment problem is to find an assignment of each agent to exactly one object that maximizes the total benefit. Furthermore, since the number

of objects may be larger than that of the agents, some objects are allowed to remain unassigned. We are given a set μ of pairs (i, j) that can be matched. For each agent i, we denote by $A(i)$ the set of objects that can be matched with agent i,

$$A(i) = \{j|(i, j) \in \mu\}$$

and for each object j, we denote by $B(j)$ the set of persons that can be matched with object j,

$$B(j) = \{i|(i, j) \in \mu\}$$

By an assignment we mean a set Φ of agent-object pairs (i, j) such that each agent is involved in at most one pair from Φ. If the number of pairs in Φ is n, we say that Φ is feasible; otherwise Φ is infeasible. We seek a feasible assignment which is optimal in the sense that it maximizes the total benefit $\sum_{i=1}^{n} a_{ij_i}$.

AP can be described as a linear programming problem as:

$$Maximize \sum_{i=0}^{n-1}\sum_{j=0}^{m-1} a_{ij} \cdot x_{ij} \tag{1.1}$$

$$Subject\ to \sum_{i=0}^{n-1} x_{ij} \leq 1, \forall j = 0, 1, \ldots, m-1 \tag{1.2}$$

$$\sum_{j=0}^{m-1} x_{ij} = 1, \forall i = 0, 1, \ldots, n-1 \tag{1.3}$$

$$x_{ij} \in \{0, 1\}, \forall i = 0, 1, \ldots, n-1, j = 0, 1, \ldots, m-1 \tag{1.4}$$

where $x_{ij} = 1$ if and only if object j is assigned to agent i.

In this thesis, we propose to build a common platform for accelerating the auction algorithm using FPGAs. The assignment problem is used as a concrete example to illustrate the effectiveness of the proposed method. Two implementation architectures, Jacobi and Gauss-Seidel, are discussed, and the latter one is proved to be more efficient for hardware acceleration. In addition, a computer-aided design (CAD) flow is implemented to automatically map the problem of different sizes into hardware. To the best of our knowledge, this is the first work to set up an auction-based architecture based on FPGAs.

1.3 Outline of thesis

The organization of this thesis is as follows. In Chapter 2, we introduce FPGA devices by reviewing their architecture, resources and design flow, to appreciate why more and more applications are implemented on FPGAs. Specifically, we will have a look at the capacities of Xilinx Virtex-5 FPGAs in detail.

In Chapter 3, we will introduce the auction algorithm which was the focus of our work. We will also discuss its two variations and then review their advantages.

In Chapter 4, two different kinds of hardware implementation architectures are discussed and compared. Then we will illustrate our architecture design, module by module.

In Chapter 5, we present our experimental results. The results show that our architecture achieves good speedup compared to a modern CPU implementation.

In Chapter 6 we finally conclude this work by both summarizing our research and discussing some future directions.

Chapter 2

FPGA Design Background

Difficulties in scaling single-thread performance without undue power dissipation have forced CPU vendors to integrate multiple cores onto a single die. On the other hand, FPGA (field-programmable gate array) and GPGPU (general purpose computing on graphics processing units)-based software/hardware co-design are becoming increasingly popular methods to assist general purpose processors in performing complex and intensive computations. Compared to fixed structure of GPU which is easier to program, FPGAs provide the best expectation of performance, flexibility and low overhead. They are essentially high-density arrays of uncommitted logic so that users can directly build up module-to-module hardware infrastructure and trade-off performance by partitioning the implementation of applications differently. In the following sections, we will talk about the architecture of FPGA and the steps of the FPGA design flow.

2.1 The Field-Programmable Gate Array

An FPGA is an integrated circuit designed to be configured by a customer after manufacturing, so it includes input/output (I/O) blocks and the core programmable fabric. The I/O blocks are located around the periphery of the chip, providing programmable I/O connections and various I/O standards. The core programmable fabric consists of logic blocks and routing channels. Fig 2.1 shows a high-level view of an island-style FPGA, which represents a popular architecture framework and is also a widely accepted model in the FPGA community. Logic blocks also called

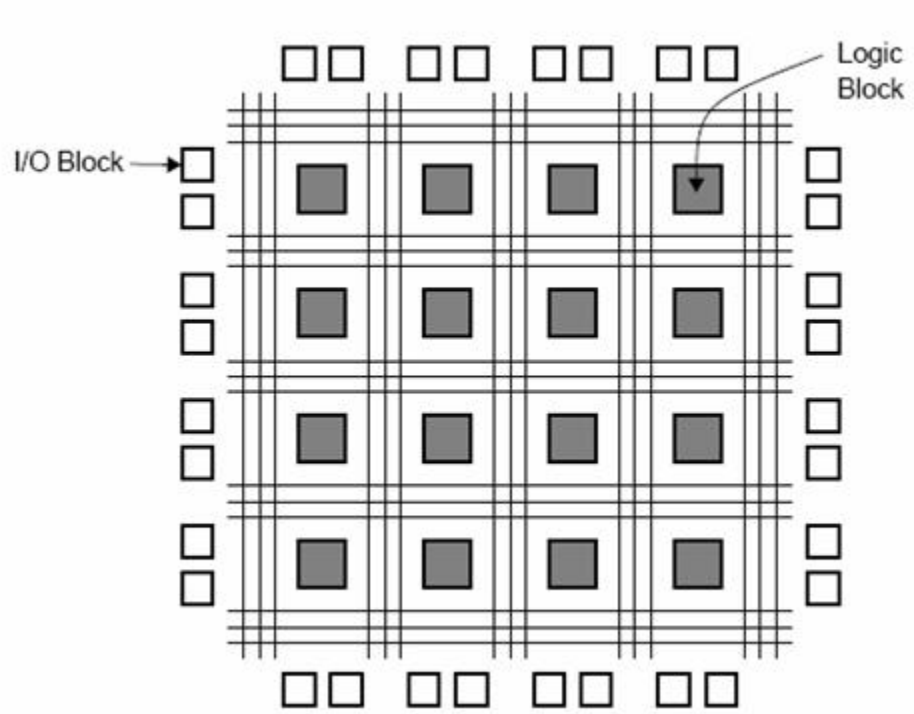

Figure 2.1: An island-style FPGA overview

configurable logic blocks (CLBs), are the main logic resources for implementing sequential as well as combinatorial circuits, surrounded by routing channels connected through *switch blocks* (SBs) and *connection blocks* (CBs), as illustrated in Fig 2.2. A *switch block* connects wires in adjacent channels through programmable switches such as pass-transistors or bi-directional buffers. A *connection block* connects the wire segments around a logic block to its inputs and outputs, also through programmable switches. Notice that all the switch blocks and connection blocks are identical. In Fig 2.2 it illustrates the different connecting situations in each SB and CB in certain electrical circuit configuration.

Routing architecture is defined by the parameters of channel width (W), switch block flexibility (F_s - the number of wires to which each incoming wire can connect in a switch block), connection block flexibility (F_c - the number of wires in each channel to which a logic block input or output pin can connect), and segmented wire lengths (the number of logic blocks a wire segment spans) [13].

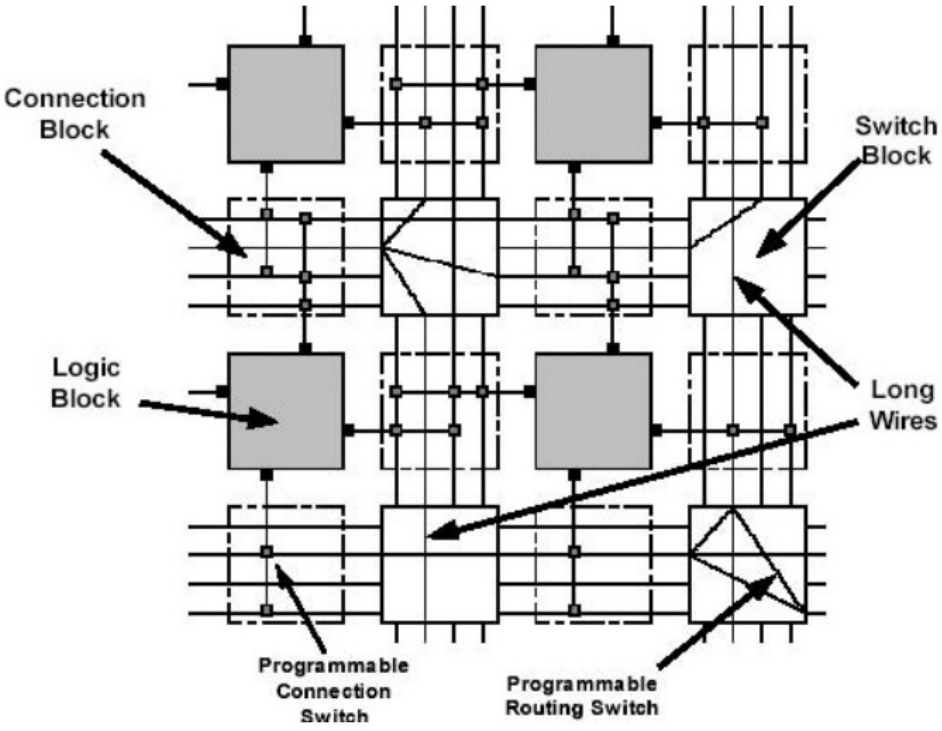

Figure 2.2: A detailed island-style FPGA under certain configuration

Further down the logic hierarchy, each logic block contains a group of basic logic elements (BLE), where each BLE contains a look-up table (LUT) and a register. The LUT is used to achieve a logic function. Fig 2.3 shows the architecture of a typical CLB.

In addition to regular logic fabric, clock distribution networks are another important issue in FPGA chips. Different distribution networks are used to make sure the clock delays from clock source to each storage element is identical, such as H-tree clock network. Besides this, global clock buffers and regional buffers are also considered to be clock network resources.

Nowadays, with the development of technology, modern FPGAs not only provide a large capacity of configurable logic gates but also integrate embedded IP cores, such as Block RAMs, DSP blocks, and specific processors, to facilitate the implementation of SoC design.

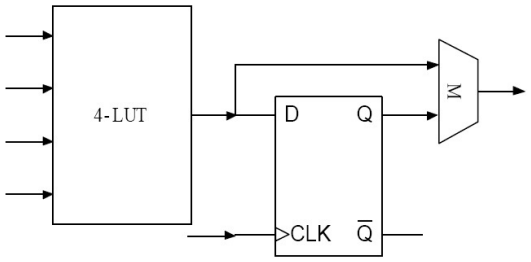

Figure 2.3: A typical CLB consist of a LUT and a register

2.1.1 The Xilinx Virtex-5 FPGA

There are two well-known FPGA companies, Xilinx and Altera, who are nowadays battling for the top position in the FPGA market. They are among the best makers of FPGAs, not only for universities, but also for big companies developing digital technology. The development tool kits provided by these two companies are very powerful to deal with the whole design flow of implementation on FPGAs. We focus on Xilinx FPGAs since they are our supporter. We will introduce Xilinx's Virtex-5 FPGA briefly to get familiar with the resources in the chip.

Xilinx Virtex-5 family provdes powerful features in the FPGA market. In addition to the advanced, high-performance logic fabric, Virtex-5 FPGAs contain many hard-IP system level blocks including powerful 36-Kbit block RAM/FIFOs, second generation 25×18 DSP slices, SelectIO technology, clock management tiles with integrated DCM (Digital Clock Managers) and phase-locked-loop clock generators [14]. Some FPGAs also feature PowerPC processors, Endpoint Blocks for PCI Express and Ethernet MACs. With these resources, many applications on different domains can be implemented on FPGAs efficiently and effectively. Table 2.1 shows the resources different devices have from Virtex-5 family. We use Xilinx Virtex-5 XC5VLX110T FPGA which is shown in bold line. As we can see that different FPGAs aim at different application fields. How to choose an FPGA to implement an application is an important issue nowadays to reduce the cost and achieve high usage rate. More details about the usage of Xilinx Virtex-5 FPGA can be found in [15].

Table 2.1: Resource table for members of the Virtex-5 FPGA Family. Selected FPGAs are similar in chip size.

Device	CLB Slices	Max Distributed RAM (Kb)	DSP48E Slices	BRAM 36Kb	BRAM Max (Kb)	PowerPC Processor Blocks	Ethernet MACs	Max User I/O
XC5VLX110	17,280	1,120	64	128	4,608	N/A	N/A	800
XC5VLX110T	**17,280**	**1,120**	**64**	**148**	**5,328**	**N/A**	**4**	**680**
XC5VSX95T	14,720	1,520	640	244	8,784	N/A	4	640
XC5VFX100T	16,000	1,240	256	228	8,208	2	4	680

Configurable Logic BLock

The configurable logic block is the basic logic element, which contains a pair of slices. Every slice contains four logic-function generators (or LUTs), four storage elements, wide-function multiplexers, and carry logic. These elements are used by all slices to provide logic, arithmetic and ROM functions.

Block RAM

The Block RAM on Virtex-5 FPGAs can store up to 36K bits of data and can be configured as either two independent 18Kb RAMs, or one 36Kb RAM. Each 36Kb block RAM can be configured as a $64K \times 1$ (when cascaded with an adjacent 36Kb block RAM), $32K \times 1$, $16K \times 2$, $8K \times 4$, $4K \times 8$, $2K \times 18$, or $1K \times 36$ memory. The Read and Write are synchronous operations and the two ports are totally independent, sharing only the stored data. Under this true dual-port mode, it can double the throughput of original RAMs and can be used as FIFOs working on two different clock domains. Block RAMs are useful and important resources in FPGAs.

2.2 FPGA Design Flow

As the FPGA architecture evolves and its complexity increases, CAD software has become more mature as well. Today, most FPGA vendors provide a fairly complete set of design tools that allow automatic synthesis and compilation from design specifications in hardware specification languages, such as Verilog HDL or VHDL, all the way down to a bit-steam to program FPGA chips. A typical FPGA design flow includes the steps shown in Figure 2.4.

We will briefly elaborate on these steps in the following:

- Specification. The specifications of a product includes a lot of aspects. For instant, function specification of a product is most important. It describes what the product will be. The more detailed specification, the better product it will be. Besides, design constraints typically include the expected operating frequencies of different clocks, the delay bounds of I/O delay, setup time and clock-to-output delay. Design language can be specified to Verilog HDL or VHDL at the register transfer level (RTL).

- Write RTL Source Code. From specifications we can write RTL HDL codes to realize the product. The RTL codes specify the operations at each clock cycle. It is most commonly used description language to describe the behavior of hardware. There is a general tread (though very slow) toward moving to specification at a higher level of abstraction, using general-purpose behavior description languages like C or System C.

- Functional RTL Simulations. This step is to verify the correctness of the written RTL codes. It plays an important role in the design flow because an error detected in this step costs less than in the following steps. Often self-test test files are used to avoid the errors made by humans. Several CAD softwares are provided for functional simulation like ModelSim and ISE Simulator.

- Synthesis. It is to break down the high-level language description to low-level implementations.

- Place-and-Route to Target Device Family. In this step, all the functional logics are mapped to the target FPGAs and are routed to realize the functions. Integrated embedded IP cores in modern FPGAs can be used for fast implementation. Modern CAD tools provides some options in this step such as size-based or performance-based options.

- Gate-Level Simulations. After placement and routing, the design can be implemented in FPGAs. But there is still no guarantee the design will work correctly. Gate-level simulations provide gate level information to produce a reliable design.

- Static Timing Analysis. It is a method of computing the expected timing of a digital circuit without requiring simulation.

- ECO. If the design has some flaws, then it needs to be modified and returned to previous steps to improve.

- Prototype. The design is completed. It is now ready for testing on devices.

- Device Level Testing. The final design are tested on the real FPGA devices to make sure that it works.

- System Level Test. The final design are tested at the system level.

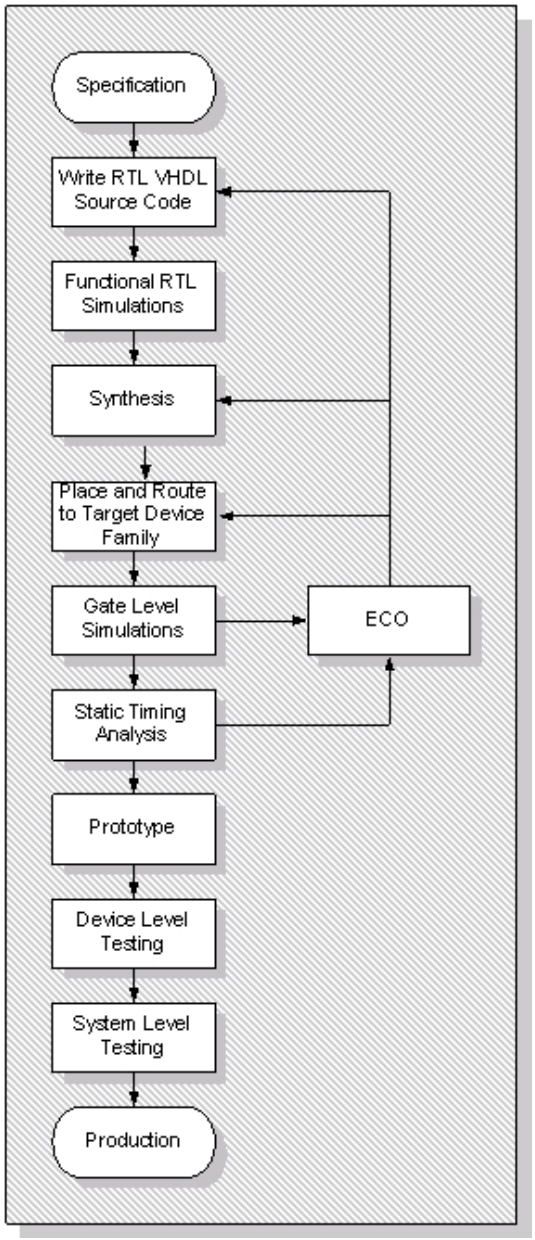

Figure 2.4: A typical FPGA design flow starting from specifications

Chapter 3

Auction Algorithms Background

In Chapter 2, we examined FPGA architecture which is suitable for parallel computing. In particular, we introduced the Xilinx Virtex-5 FPGA with CLBs and BRAMs. However, not all the existing algorithms are suitable in a FPGA's parallel computation environment.

In this chapter, we explore in details of the Auction Algorithm for solving assignment problems. The key of this algorithm is to use an iterative approach to reduce the complexity of control and increase the parallelism. In Chapter 4, we will utilize this parallel nature to develop a novel FPGA architecture for implementation.

3.1 Details of the Auction

The Auction algorithm is one of the classical methods to optimally solve the assignment problem [6] [8] [7]. It is different from primal-dual algorithms such as Hungarian method or primal algorithms. It departs from the cost improvement idea but it may deteriorate the cost at every iteration although it find an optimal primal solution in the end. The auction algorithm is also highly intuitive and easy to understand. It can be explained in terms of economic competition concepts like real auction in our daily life. In this section, we review the auction concept in economic terms and introduce three variations of auction algorithm for assignment problem.

3.1.1 The Forward Auction Algorithm

Recall from Chapter 1 Section 1.2 that there are n agents and m objects that we have to match them on a one-to-one basis. There is a benefit a_{ij} for matching agent i to object j and we want to assign agents to objects in order to maximize the total benefit. In order to develop an intuitive understanding of the auction algorithm, it is helpful to introduce an economic equilibrium problem that turns out to be equivalent to the assignment problem.

Consider a real auction where n agents are going to bid for m objects according to their own best interest. Suppose that object j has price p_j and that the agent who receives the object must pay the price p_j. Then the net value of object j for agent i is $a_{ij} - p_j$, and each agent i would logically want to be assigned to an object j_i with maximal net value, that is, with

$$a_{ij_i} - p_j = \max_{j \in A(i)} \{a_{ij} - p_j\} \tag{3.1}$$

where $A(i)$ is the set of objects that agent i can be assigned to. In reality agent i may not get the best object j due to others' competition. When other agents bid for the object j, the price p_j may increase to such an certain extent that agent i loses the interest of object j. In other words, agent i will bid for other objects for its best interests.

Equilibrium assignments and prices are of great interest to economists, but there is also a fundamental relation with the assignment problem. It turns out that an equilibrium assignment offers maximum total benefit, while the corresponding set of prices solves an associated dual problem.

As we have already had an overview the auction concept, now we will introduce the algorithm in details. Basically, the auction algorithm is composed of two phases in each iteration, a bidding phase and an assignment phase. Then it proceeds in iterations and generates a sequence of price vectors and assignments.

Typical Iteration of Auction Algorithm

Two phases are depicted as followings:

Bidding Phase: Each agent $i \in I$, where I is the set of unassigned agents, finds an object j_i with the best net value V_{ij_i}, that is,

$$V_{ij_i} = a_{ij_i} - p_{j_i} = \max_{j \in A(i)} \{a_{ij} - p_j\} \tag{3.2}$$

and also the second best net value W_{ij_i} where

$$W_{ij_i} = \max_{j \in A(i), j \neq j_i} \{a_{ij} - p_j\} \tag{3.3}$$

and then gives its reasonable bid b_{ij_i}, that is,

$$b_{ij_i} = p_{j_i} + V_{ij_i} - W_{ij_i} + \epsilon \tag{3.4}$$

and the bidding increment γ_i is

$$\gamma_i = V_{ij_i} - W_{ij_i} + \epsilon \tag{3.5}$$

where ϵ is a parameter.

Assignment Phase: Each object j that is selected as best bid by the agents in I, determines the highest bidder i_j, raises its price to b_{ij_j}, and gets assigned to the highest bidder i_j; The agent that is assigned to j at the beginning of the iteration (if any) get unassigned.

The algorithm continues with a sequence of iterations until all agents are assigned to an object.

ϵ-Complementary Slackness and ϵ-Scaling

The main method of the auction algorithm, like Hunagrian Algorithm, is to introduce another set of variables, prices in this case, so as to get the optimal assignment. During the computation, each agent has to raise the price of his interested object to defeat others in the competition. Unfortunately, without ϵ, the bidding increment γ_i is zero when more than one object offers maximum value for the bidder i. As a result, a situation may be created where several agents contest a smaller number of equally desirable objects without raising their prices, thereby creaing a never ending cycles. To break such cycles, ϵ works like a real auction where each bid for an

object must raise the object's price by a minimum positive increment, and bidders must on occasion take risk to win their preferred objects. In order to derive an optimal solution, the parameter ϵ must be set $\epsilon < \frac{1}{n}$, which is proven in [6] and [7]. However, it is not necessary to fix the ϵ during the overall computation, which is called ϵ-scaling. At first, we can run the auction algorithm where ϵ can be set as a much bigger constant to make the price vector close to the ultimate one. Then we derease the ϵ value and run the algorithm again until ϵ is smaller than $\frac{1}{n}$. It can achieve the optimal solution more promptly in major cases but it doesn't affect the FPGA architecture design while it introduces more control.

For greater clarity, we write the algorithm in the pseudo code shown in Table 3.1. The input of the algorithm is the number of agents and objects, and associated benefit matrix as well. The output is the assignment which yields the maximal benefit in the end. The algorithm runs in iteration, switching from bidding phase to assignment phase.

The algorithm has several characteristics as follows:

- Comparably fast. Compared to the Hungarian algorithm (HA), in which a maximal alternating forest is constructed in each iteration that has computational complexity $O(n^3)$, auction algorithm is still comparably fast with complexity $O(nm\log(nC))$, where $C = \max\{a_{ij}\}$. In some cases, the auction algorithm is even faster than HA. An interesting analysis in paper [16] reveals that the real running time for dense random instances of the linear sum assignment problem is only $O(m\log(n))$. The author adopted the assumption of randomness to estimate the real computation load of the auction algorithm. He derived the above computation complexity based on the possibility theory.

- Nature of parallelism and simplicity. The computational operations are simple with much fewer controls than HA. Parallelism is easy to understand because all the unassigned agents in set I can bid simultaneously and each unassigned agent does the same operation to compute the best bid for ideal object.

Table 3.1: Pseudo-code of the auction algorithm

```
Input:    agents n, objects m; benefit matrix A
Output:   assignment Φ
1         BEGIN
2           INIT p_j=0, Φ_j=-1 for all objects j;
              put all unassigned agents in a list I;
              set the number of assigned agents, nAssi=0
3           WHILE nAssi != n
              /* Bidding Phase */
4             Get some unassigned agents from I
5             FOR all the objects j
6               Compute the net benefit, a_ij − p_j
7             GET V_ij_i, W_ij_i and j_i
              /* Assignment Phase */
8             COMPUTE the bid of agent i for object j_i
9             IF Φ_j_i != -1
10              ADD Φ_j_i to the list I
11            ELSE
12              nAssi++
13            Φ_j_i=i;
14            p_j_i=bid
15          RETURN Φ
16        END
```

This algorithm was first proposed by Bertsekas in [6]. It was named by Forward Auction Algorithm (FAA) since it acts like a real auction. Later on, the algorithm was further developed in the literature [8] [17] leading to the Reverse Auction Algorithm (RAA) and the Forward/Reverse Auction Algorithm (FRAA). We will introduce them in the next section and discuss about their characteristics.

3.1.2 Reverse Auction Algorithm

In the previous section, we know that each object has its own price to attract certain agents while it is possible to exchange the roles of agents and objects. This leads to the idea of reverse auction where the objects compete for agents by essentially offering discounts (lowering their prices). Roughly, given a price vector p, we can

view the net value of the best object for agent i

$$\max_{(i,j)\in A(i)} \{a_{ij} - p_j\} \tag{3.6}$$

as a profit π_j for agent i. When objects lower their prices they tend to increase the profits of the agents. Therefore, profits play for agents a role analogous to the role prices play for objects. Forward and reverse auctions can be seen to be mathematically equivalent.

Typical Iteration of Reverse Auction

The reverse auction is mathematically equivalent to the forward auction. Two phases are depicted as followings:

Bidding Phase: Each object $j \in J$, where J is the set of unassigned objects, finds an agent i_j with the best net value β_{ji_j}, that is,

$$\beta_{ji_j} = a_{ij_i} - \pi_{i_j} = \max_{i\in B(j)} \{a_{ij} - \pi_i\} \tag{3.7}$$

and also the second best net value ω_{j_i} where

$$\omega_{ji_j} = \max_{i\in B(j), i\neq i_j} \{a_{ij} - \pi_i\} \tag{3.8}$$

and then gives its reasonable bid r_{ji_j}, that is,

$$r_{ji_j} = \pi_{j_i} + \beta_{ji_j} - \omega_{ji_j} + \epsilon \tag{3.9}$$

where ϵ is a parameter.

Assignment Phase: Each agent i that is selected as best by the objects in J, determines the highest bidder j_i, raises its profit π_j to r_{ji_j}, and gets assigned to the highest bidder j_i; The object that is assigned to i at the beginning of iteration (if any) get unassigned.

The algorithm continues with a sequence of iterations until all agents are assigned to an object.

Since the reverse auction algorithm is mathematically equivalent to the forward auction algorithm, there is no improvement on performance. However, to combine

forward and reverse auction algorithm, we can construct a new algorithm, that is FRAA, which switches from forward to reverse auction and back. It simultaneously maintains a price vector p and a profit vector π. More details of FRAA are described in paper [17]. In this section we describe one easily implementable possibility of FRAA.

3.1.3 Forward/Reverse Auction Algorithm

FRAA is a combined algorithm which implements FAA and RAA within steps until all the agents or objects are assigned. There are a lot of choices one can choose to determine when a switch from FAA to RAA will occur. One easily implementable strategy is depicted as follows:

Step 1: (Run FAA) Run forward auction algorithm serveral iterations. At the end of each iteration, at least one agent-object pair is added into the assignment. Then set

$$\pi_j = a_{ij_i} - p_{j_i} \tag{3.10}$$

for every agent-object pair that entered the assignment during the iteration. Go to Step 2.

Step 2: (Run RAA) Run the reverse auction algorithm for several iterations. At the end of each iteration, at least one agent-object pair is added into the assignment, then set

$$p_j = a_{i_j j} - \pi_{i_j} \tag{3.11}$$

for every agent-object pair that entered the assignemtn during the iteration. Go to Step 1.

The algorithm terminates when all the agents or objects are assigned.

3.2 Comparison of Auction Algorithms

With the introduction of the profit vector used in RAA, the combined algorithm FRAA outperforms the FAA in some cases where fierce competition occurs. In other words, if several agents have the similar interests on a few objects, FAA leads

to a large number of iterations so that the price vector converges to the optimal one slowly, because it only increase the price vector bid by bid. FRAA use the profit vector, which can increase promptly after each switch to reach the optimal one. However, FAA still outperforms FRAA in other cases, and moreover, FRAA often requires additional overhead over FAA or RAA. Last but not least, FAA is also the core of auction-based variations to solve other network problems, such as shortest path problems. So we focus on the forward auction algorithm in this thesis. We will explore the architecture design of FAA in Chapter 4. Before that we will review some existing work on hardware implementations of the auction algorithm in the literature.

Chapter 4

FPGA Architecture Exploration

The key point of using an FPGA for acceleration lies in extracting and exploiting the parallelism in a target problem. Consider the forward auction algorithm: there are two potential hardware architectures, namely, Jacobi and Gauss-Seidel, to expose the parallel computing flow. In this chapter, we compare these two architectures and argue that the latter one is better for acceleration. Then we describe the details of the FPGA design to solve any assignment problem.

4.1 Jacobi Parallelism and Gauss-Seidel Parallelism

In existing work, these two parallelism strategies were both implemented on different hardware. In this section, we analyze the implementation on FPGAs.

4.1.1 Jacobi Parallelization Architecture

Jacobi parallelization regards each agent as an individual processing element and allows several unassigned agents to bid for objects simultaneously. The problem can be understood in an easier way, shown in the left part of Figure 4.1. For the n-to-m assignment in a benefit matrix A, each row represents an agent and each column represents an object is introduced. The element of the benefit matrix a_{ij} represents the benefit value that agent i has for object j. The prices of all the objects are stored in a separate vector. At the beginning of each iteration, the PE array (1-D array of PEs) computes the net values for the first object (first column). The corresponding price is provided for all the PEs. Then it goes through all the

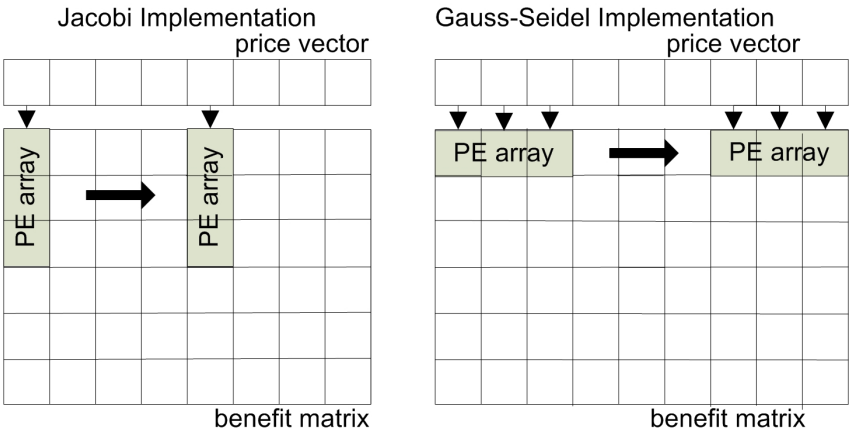

Figure 4.1: Jacobi implementation (left) and Gauss-Seidel implementation (right) of auction algorithm

columns to compute the best net values, second best net values, and the bids of the unassigned rows.

4.1.2 Gauss-Seidel Parallelization Architecture

In contrast with the Jacobi method, the Gauss-Seidel method regards each object as an individual PE, shown in the right part of Figure 4.1. Each PE is responsible for computing the net value of one object. At first, the PE array computes the net values for the first several columns at the same time. After finishing the computation of the first block, it moves forward a block to calculate the next set of net values. By several steps of computation, it will go through all the columns, to find the best object of the computed agent. Different from the Jacobi method, it computes one bid per iteration.

4.1.3 Jacobi method vs. Gauss-Seidel method

The Jacobi and Gauss-Seidel architectures result in a performance difference and we find that Gauss-Seidel leads to better performance when implemented on FPGAs.

The major reason for the difference is due to the fact that during the auction process, the number of assigned agents increases quickly with the problem size.

Table 4.1: Statistics on the number of unassigned agents

Number of unassigned agents	Fraction of iterations
1	89.64%
1-10	99.50%
1-20	99.81%
1-50	99.97%

In other words, the number of unassigned agents decreases quickly. Table 4.1 illustrates the iteration distribution with respect to the number of unassigned agents, using randomly generated test benches. When the number of unassigned agents is less than fifty, the number of iterations takes up to 99.97%. When only one agent is not assigned, it takes 89.64% of the total iteration to derive the optimal solution. Figure 4.2 illustrates the utilization of 16 PEs implemented by Jacobi method. As is clearly shown, the utilization of 16 PEs decreases quickly to below 10%. In this case, the Jacobi architecture, whose processing granularity is the agent, leads to a large number of idle PEs as there are few unassigned agents left.

Different from the Jacobi architecture, the Gauss-Seidel architecture handles one agent at a time, and the bids for different objects are computed in parallel. As each agent will always need to compare all objects, the hardware usage does not drop. Therefore it achieves better acceleration using the Gauss-Seidel architecture.

Another important factor to consider is that Gauss-Seidel architecture can introduce pipeline technology to increase the throughput. The bidding phase and the assignment phase can be divided into three steps, so there is no need to separate these two in the Gauss-Seidel architecture. However, the Jacobi implementation derives the bid at the end of iteration simultaneously, it has to take additional time for assignment in this case. More implementation details are shown in the next section.

To sum up, the Gauss-Seidel parallelization is more suitable for FPGA implementation. Experiments show that the Gauss-Seidel architecture achieves a 6X speedup on average compared to the Jacobi architecture.

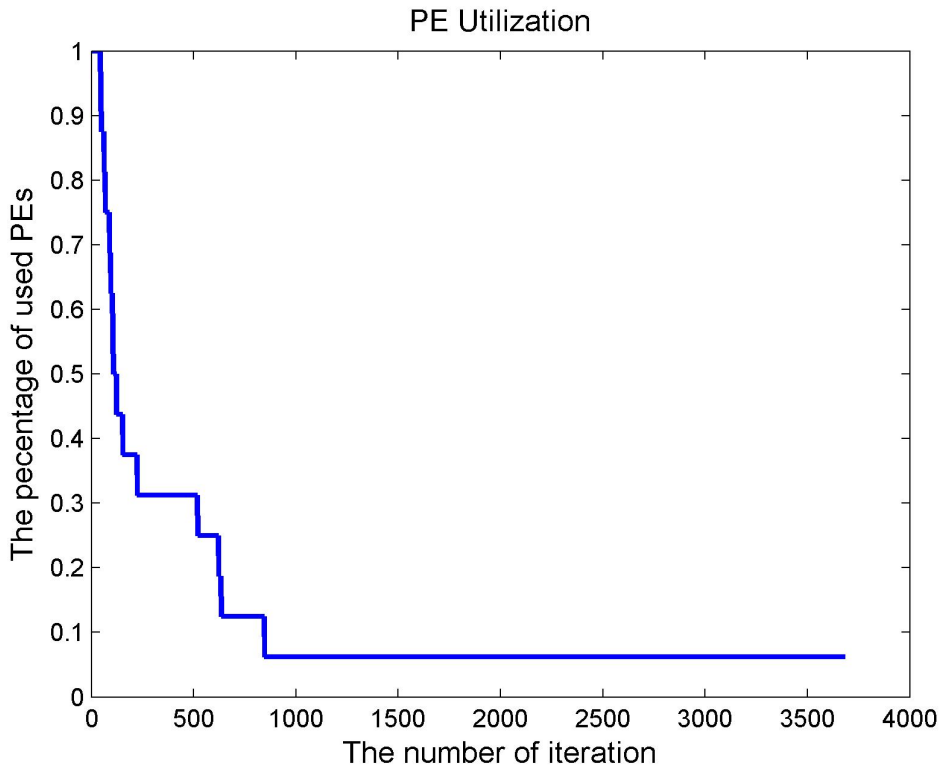

Figure 4.2: The utilization of 16 PEs in Jacobi implementation, using uniform random class with problem size 500.

4.2 Related Work of FAA Implementation on Hardware

There are several proposed hardware implementation of auction algorithms in the literature.

In paper [18], Wein implemented the forward auction algorithm on a connection machine CM2, which is a computer with up to 65536 single-bit processors in SIMD mode. A mapping of virtual processors in a grid to the physical processors of the CM is used. Processor (i, j) stores the corresponding a_{ij} of the benefit matrix, and local variables applicable to agent i such as the most profitable object to that agent, and local variables applicable to object j, such as price. He implemented the Jacobi, Gauss-Seidel and a hybrid version (where a switch from Jacobi to Gauss-Seidel when the number of unassigned agents is below a threshold), and the hybrid

implementation achieved good speedups varying from 3X to 5X.

Bertsekas discussed the parallel implementation in parallel synchronous and asynchronous mode in [19]. Whether in synchronous Jocobi or Gauss-Seidel implementation, there are several barriers within four steps such as selecting unassigned agents, computing their bids, merging the bids and doing the assignment, which is one iteration. Each step has to wait for the completion of its previous step, and then it starts its task. So the asynchronous version is only to introduce the pipeline but the computation has to across iteration which can be wasteful computation.

In paper [20] Bagherzadeh implemented the synchronous and asynchronous computation models on an hypercube multiprocessor. Each processing node was an i860 microprocessor and 16 Mbytes of RAM. It was a MIMD message implementation. While it focused on the convergence problem of auction algorithm. It scarified the optimal results to achieve the fast computation.

In paper [21] Naiem implemented the Gauss-Seidel auction algorithm using the message passing interface on a computer cluster, aiming at solving the scalability. Through this system, low speedup ratio was achieved.

More recently, Vasconcelos and Rosenhahn [12] used a GPU to accelerate auction algorithm. They implemented Jacobi version and streamed data into the GPU and achieved 7X with problems ranging in size 400 to 4000.

Since FPGAs are more flexible and have better data processing granularity. They are essentially high density arrays of uncommitted logic so that developers can directly build up module-to-module hardware infrastructure and trade-off performance by partitioning implementation of applications differently. FPGAs are more flexible to implement auction algorithms. In the next section, we will focus on the implementation of Gauss-Seidel version, which we call the Gauss-Seidel architecture.

4.3 Platform Implementation

In this section, we discuss in detail the implementation of Gauss-Seidel architecture. We develop a flexible hardware platform which can automatically generate

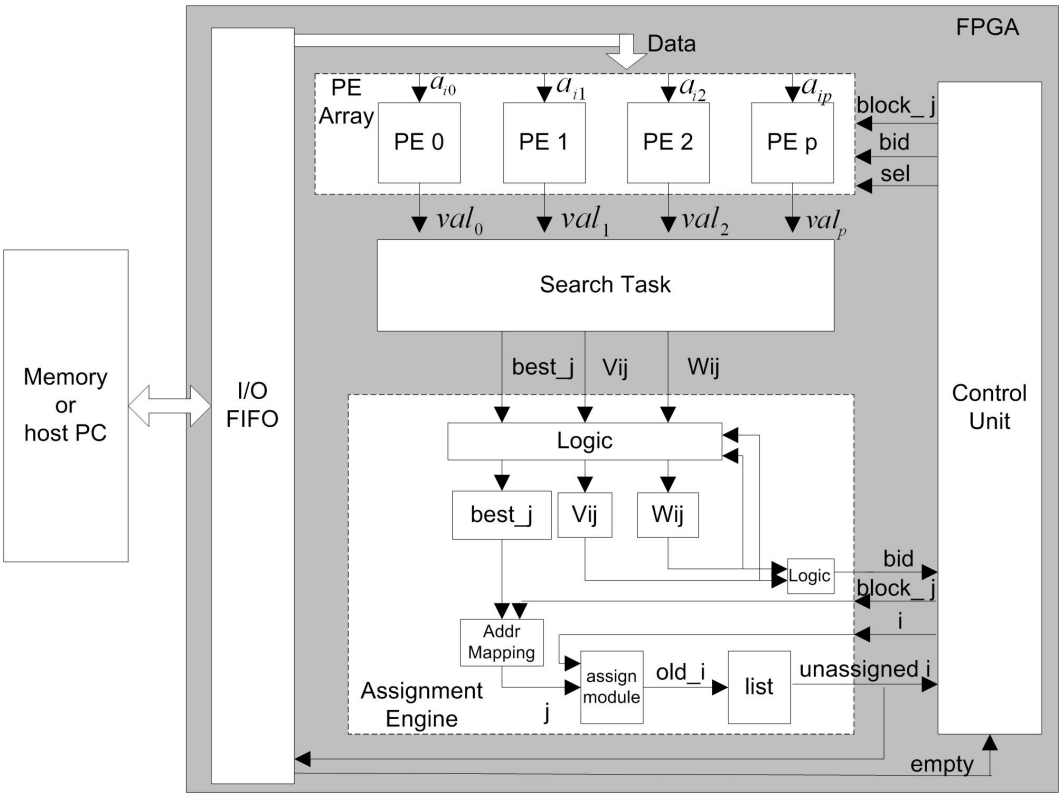

Figure 4.3: Gauss-Seidel Architecture with p PEs for parallel computation.

the architecture of user-specified size to solve assignment problems of arbitrary size efficiently. It ensures that no PE will be idle during the bidding phase throughout the whole computation progress. We leverage the Block RAMs (BRAMs) in modern FPGAs to keep the instances data so as to save the energy dealing with the interface to the external memory or host machine. Figure 4.3 illustrates the detailed architecture implemented in FPGA.

4.3.1 Gauss-Seidel Architecture

The architecture is composed of five modules: I/O FIFOs, PE Array, Search Task, Assignment Engine, and Control Unit. Among them, PE Array and Search Task modules are two data-intensive computing units. The basic units to compose these two modules have regularity in structure, which brings the benefit of highly extensible architecture. I/O FIFOs offers an effective method to reduce the overhead of

data transmission through interface. Other two modules are independent of architecture size.

I/O FIFOs

I/O FIFOs send requests of data and receive benefit matrix to and from memory. In order to minimize the communication cost, two input FIFOs work alternatively to receive data from memory. As long as the unused input FIFO is not full and there is a queue of unassigned agents, the data will always be received from memory to fill the FIFO.

PE Array

The PE array is composed of a number of identical PEs in one dimension. Each PE aims at computing a net value in the bidding phase (corresponding to line 6 in Table 3.1), shown in Figure 4.4. Inside a PE, one distributed RAM is used as a cache to store a group of prices in a specific order when the problem size is larger than the number of PEs. Since the number of prices grows linearly with problem size, it is costless to keep the prices in an FPGA. All the PEs are identical in data-processing behavior so that they can share the same control signal.

To update a specified price, we use a decoder to select the target PE and provide the address of the local RAM at the same time, so that we can access any price in the PE array.

More details are described in the appendix.

Search Task

After an array of net values are calculated, search task computes the best net value, the second best net value, and the best value's location in the PE array in this block.

To realize the search function, two stages are employed to get the final results, shown in Figure 4.5. The first stage is to compute the best net value and its location in the PE array by a comparison tree. In the second stage, the known best value is eliminated from the input data so that the second best net value can be derived.

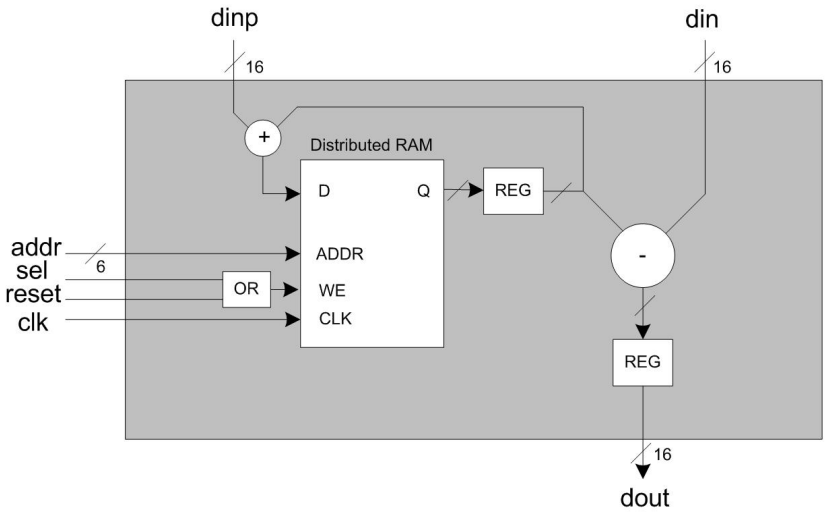

Figure 4.4: The structure of a PE module.

The comparison tree consists of many regular two-input comparators, bringing the benefit of extensibility. And it is pipelined to prevent clock frequency from dropping dramatically when architecture size increases. Furthermore, it is pipelined partially instead of fully because a highly-pipelined search introduces high latency which influences the overall performance.

Assignment Engine

The goal of the Assignment Engine is to derive all the required values to compute the bid and to complete the functions in the assignment phase (corresponding to lines 7-14 in Table 3.1).We map those lines into three steps and each step takes one clock cycle. In the first step, the best net value V_{ij_i}, its location in the PE array j_i, and the second best bid W_{ij_i}, are computed (line 7). Based on the derived data, the bid b_{ij} can be computed at this time (line 8). Meanwhile, the assignment status of object j_i is checked (line 9). Finally, it completes the three functions in assignment phase (line 10, 13, 14 in Table 3.1).

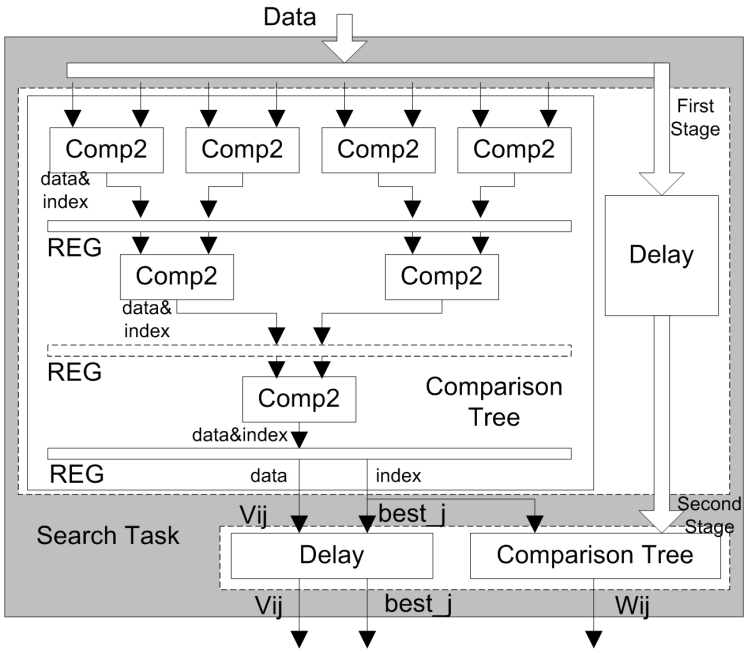

Figure 4.5: Details of the Search Task module. The total latency of the 8-input architecture is 4, introduced by registers (solid line) in two stages. The dotted-line register is eliminated to realize a partial pipeline.

Control Unit

Control Unit module is to control all the other modules to work together. There are six main states in the module, and they are idle, loading, computing, updating, output and completion, and the relationship between them are shown in Figure 4.6. In each main state, there is one or more sub-states to deliver the detailed control signals, which is called factored state machine. Using factored state machine is much easier to design a complex system-level finite state machine. Now, we will explain the main functions that are realized in each main state.

- idle: the system is idle when reset button is pushed.
- loading: in the loading state, we load the necessary parameters into registers, and initialize the price vector and assignment vector as well in this period.
- computing: in this state, all the data processing modules are waiting for valid data arriving, including PEs, Search Task and Assignment Engine. Control

Table 4.2: Different architecture size (the number of PEs) introduces different latency.

Number of PEs	16	64	128	256
Latency (cycles)	4	6	8	8

Unit sends read signals to retrieve data from memories.

- updating: in this state, assignment is being implemented.
- output: when all the computation of the algorithm is complete, the assignment results are output onto the PC-FPGA interface, such as JTAG. Using Xilinx's ChipScope tools, we can monitor the experimental results and do the verification on board.
- completion: the algorithm is complete, waiting for another reset signal for new computation.

It is worth mentioning that the termination of the algorithm can be easily realized. Through careful observation, we figure out that once an unassigned object is assigned to an unassigned agent, the number of assigned pairs will increase by one and won't decrease until the end of the assignment progress. In other words, the termination can be simply implemented by a counter. When the value of the counter is the same as the value of problem size (i.e. the number of agents needed to be assigned), the termination is met and the whole progress ends.

4.3.2 Influence of comparison tree on performance

There is no doubt that better performance can be achieved by a parallel architecture with more processing elements. However, the latency introduced by the search task has some negative impact on the overall performance. In each assignment phase the price of the bid object requires updating; it has to wait until search task finishes the latest computation.

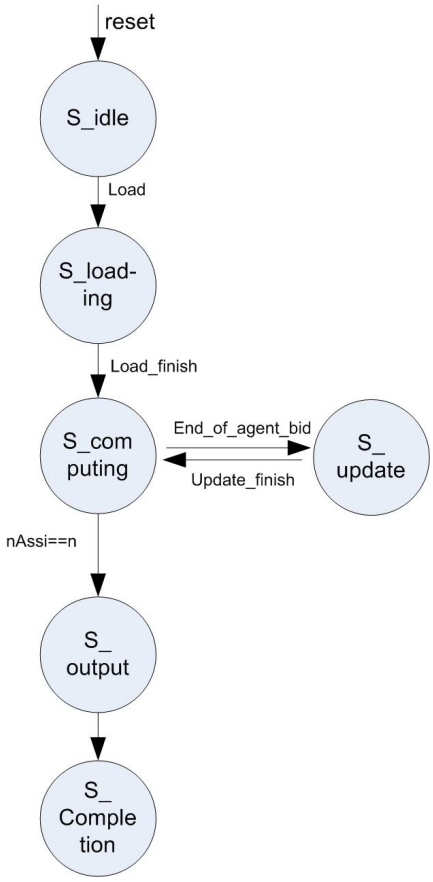

Figure 4.6: Details of the main Finite State Machine (FSM). There are six main states in the FSM.

Since a full pipeline in the search task module introduces high latency, which heavily affects the performance, we shorten the pipeline by half. The reduced registers is depicted by dotted line in comparison tree in Figure 4.5. We make sure that search task module does not become the longest path in the whole design. The latency introduced in search task is 4, 6 and 8 cycles when the number of PEs is 16, 64 and 128, respectively, as Table 4.2. The influence on performance based on different architecture size is depicted in Figure 4.7. From the figure, we can see that given a certain architecture size, the percentage of latency decreases and more cycles will be spent on effective computation when the problem size increases. Moreover, for the problems of small size, it is wise to implement a relatively small architecture, to avoid the influence of high latency.

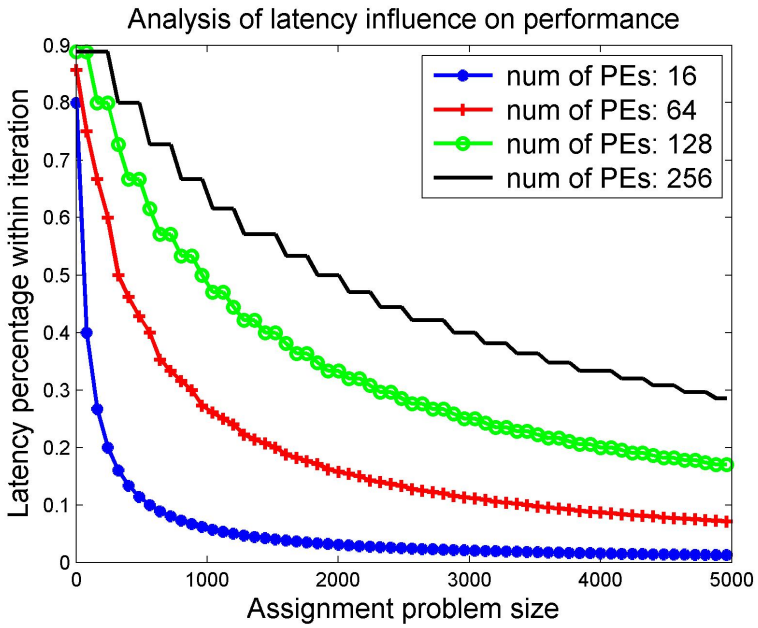

Figure 4.7: The percentage of cycles consumed by the latency within iteration.

The implementation can be further improved by adjusting the schedule of assignment phase from one after each bidding phase to one after several bidding phases, so as to save the latency cycles within iteration. The validation of this implementation is shown in [22]. It can efficiently mitigate the influence of the latency. However, when problems involve several unassigned agents (often 1-2 agents), the latency influence is unavoidable.

Chapter 5

Experiments and Results

In this chapter, we analyze the experiments that we performed on our FPGA. The experiments were conducted using the most commonly used class as our benchmarks. The analysis shows that an FPGA implementation of the auction algorithm can perform better than a comparable CPU.

5.1 Experiment Prerequisite

The proposed Gauss-Seidel architecture for the assignment problem is implemented on Xilinx Virtex-5 LX110T FPGA with a speed grade of -1. A CAD flow is employed to generate the architecture of user-specified size in Verilog HDL. We use ModelSim SE 10.0a for simulation, and Xilinx ISE 13.1 to synthesize and place-and-route our design.

To test the performance of the proposed architecture, we have used the most common classes of instances used in the literature [23] to test AP algorithms, which is called uniform random class. This class includes that the entries of the benefit matrix are integers uniformly randomly generated in $[0, K]$, with $K = 1000$. Although we used symmetric problems as examples to test, asymmetric problems also can be computed.

Table 5.1: Resource Utilization based the on the architecture of four selected sizes and their clock frequency

Resources	Size							
	16		32		64		128	
	Used	Uti.	Used	Uti.	Used	Uti.	Used	Uti.
Registers	1229	1%	2532	3%	5139	7%	11369	13%
LUT as logic	2163	3%	3975	5%	7721	11%	15046	21%
LUT as mem.	540	3%	1053	5%	2078	11%	4127	23%
BRAMs	124	83%	130	87%	130	87%	130	87%
Freq. (MHz)	220		193		163		139	

5.2 FPGA Implementation Analysis

5.2.1 Complexity and Cost of Different Architecture Size

The architectures of several sizes have been mapped into hardware. For a single Virtex-5 LX110T FPGA, we can implement more than 400 PEs for parallel computation. However, we did not implement the architecture of 400 PEs in our experiments. The reason is that we leverage the on-chip BRAMs to store the test instances. Because of the limited amount of BRAMs, 500×500 elements can be stored. In this situation, the architecture with large size only introduces more latency within each iteration, so we implement 16 PEs, 32PEs, 64PEs and 128 PEs in our experiments. Table 5.1 shows the resource utilization for the four selected architectures. From the table we can see that LUTs are heavily consumed when the architecture size is 128. This is because the search task module consumes a large amount of resources, nearly 95% of the total consumed resources. Overall, the cost of hardware resources increases linearly with the architecture size. However, the consumption of BRAMs is constant high, around 87% of the total number of BRAMs. It is because we also use BRAMs to store our test instance data.

The timing report shows that the clock frequency decreases with the increasing architecture size. This is no surprise since the large usage of FPGA routing resource results in the difficulty in routing task, which increases the time consumed by routes. However, the clock frequency can be further improved by manual floor planning.

5.2.2 Speedup Analysis

For comparison, we implement a C program based on the public FORTRAN codes originally written by the author of [7]. The program was run on a PC with a Intel Dual Core CPU running at 2.7GHz with 4.00GB RAM. Both the CPU and FPGA implementations eliminate the epsilon-scaling phase and set the epsilon as a constant, to eliminate the implementation complexity. Symmetric assignment problems of size from 200 to 500 were tested due to the limited number of BRAM resources. We use 5 instances of uniform random class per problem size to test. All the assignment results are optimal verified by public codes of Hungarian algorithm.

The speedup is computed as follows:

$$Speedup = \frac{Computation\ time\ by\ CPU}{Computation\ time\ by\ FPGA}$$

The computation time is core calculation time to execute the algorithm without instance preparation.The speedup of 16-PEs and 128-PEs architectures on four sets of assignment problems is shown in Figure 5.1. There is an increasing trend of speedup when the problem size grows. It is because the latency influence degrades and more cycles are spent on effective computation, so the proposed architecture can achieve a higher speedup ratio for large problems. Currently, the speedup is more than 10X when the problem size reaches 500.

5.2.3 System Bottleneck

As we compared previously, the core computation time is the main part for parallel computation. Table 5.2 shows the core computation time used by CPU and FPGAs.

Table 5.2: The core calculation time by the CPU and the FPGA and the data-transferring time from PCs to FPGAs through on-board PCIe x1.

Problem Size	200	300	400	500
Core Computation time by CPU (msec)	1.67	3.20	8.00	11.25
Core Computation time by FPGA (msec)	0.27	0.57	1.16	1.48
Transfer time (msec)	0.32	0.72	1.28	2.00

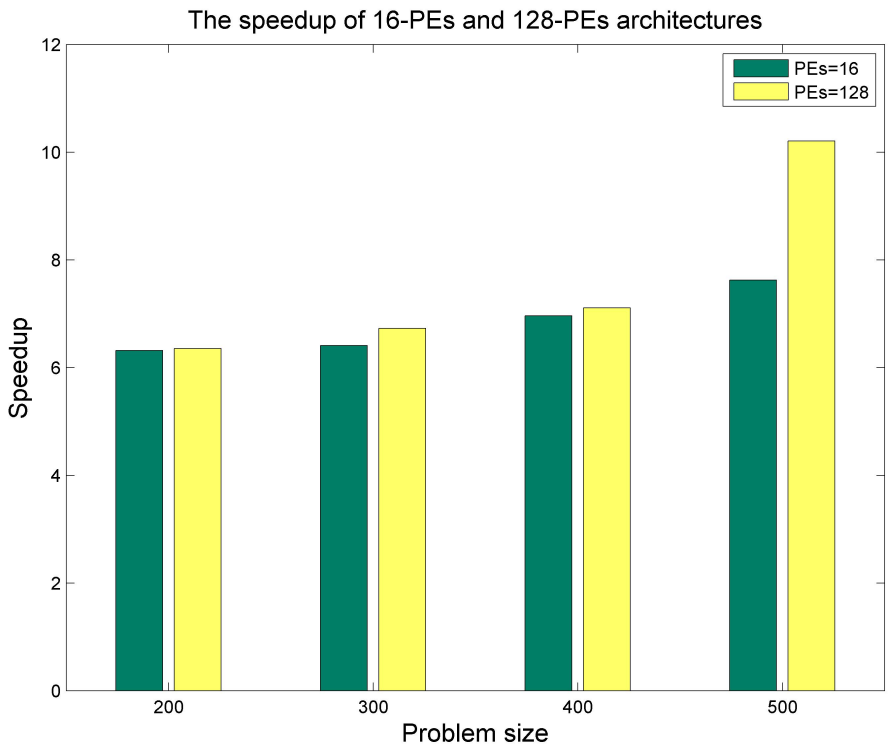

Figure 5.1: The speedup is averaged based on 5 random generated instances. The architecture contains 16 PEs and 128 PEs for parallel computing respectively. The problem size ranges from 200 to 500.

However, except the core computation time, we still want to estimate the overall time consumption by the system, including data generation and data transfer between CPUs and FPGAs through interfaces. The developing board we used is equipped with PCIe x1 slot, thus providing the opportunity of fast-transferring data. In the current experiment, since we use on-chip BRAMs to store all the test data, downloading data from PCs to FPGAs only require once. The standard bandwidth for PCIe x1 is $2.5Gb/s$. The effective bandwidth per direction due to the overhead of the 8B/10B encoding and decoding used by the protocol is $2Gb/s$. Therefore the data-transferring time would be what is shown in Table 5.2. As we can see that data-transferring time counts for as much as the core computation time by FPGAs. Therefore, we should pay attention to the data transfer between CPUs and FPGAs, and should reduce the frequency of this kind of transfer.

Chapter 6

Conclusions and Future Work

In this chapter we review the accomplishments of this work and briefly discuss future directions of our research.

6.1 Conclusions

In Chapter 2, we briefly talked about the FPGA chip architecture with plenty of CLBs, BRAMs, integrated IPs and other resources for different-domain applications. With the development of high-density FPGAs, and entire SoC design can be built on a single FPGA. FPGA design has the early time-to-market advantages compared to ASIC design, which is supported by plenty of sophisticated CAD tools.

Then we discussed the details of auction algorithms: Forward Auction Algorithm, Reverse Auction Algorithm and Forward/Reverse Auction Algorithm in Chapter 3. The FAA acts like a real auction in an economic market. It has the nature of parallelism. Even in serial computing environment, it is fast enough to solve the assignment problem with time complexity $O(n^2 \log(nC))$. RAA is mathematically equivalent to FAA. FRAA, combined with FAA and RAA, overcomes the situation where fierce competitions occurs while it introduces some overhead in switches between FAA and RAA. FAA has the simplicity and extensibility to solve other network problems so that we focus on FAA implementation on FPGAs.

In Chapter 4 we explored two classes of auction-based parallel architecture for solving the assignment problem and figured out that Gauss-Seidel method is more suitable for parallel implementation. It ensures that no PE is going to be

idle throughout the bidding phase, even when the number of unassigned agents converges to one. Then we set up a general platform which generates extensible and application-specific hardware architecture. It can efficiently deal with assignment problem of any size. The proposed architecture can achieve more than 10X speedup with 128 PEs on the problem at size 500. As the problem size grows, a higher speedup ratio can be achieved.

6.2 Future Work

Although our implementations of Auction Algorithms on an FPGA outperform a conventional CPU implementation, there are many directions to be considered for future research. In this section, we describe some of these possible directions.

6.2.1 Interface with FPGAs

The efficient interface with FPGAs is very important in the near future when FPGAs are used as co-processors to compute large amounts of data. The PCIe interface between FPGA and CPU or on-board memory can be utilized for data transmission. Modern FPGA platforms are available that integrate these features.

6.2.2 Improvement on Data Reuse

Recall that in the bidding phase of FAA, we have to compute the best net value V_{ij} and the second best net value W_{ij} so as to compute the bid for each unassigned agent. It is computation-intensive and time-consuming process. The first approach to reusing bids was described in [7]. The author mentioned to use the third highest net value as well. It is quite useful to compute the bids only by several objects with highest net values in last iteration, instead of by the whole object set. A set of BRAMs can be used as Cache to store a list of objects to speedup the computation.

6.2.3 Architecture Design for Other Problems

Currently, we concentrate on one of the auction algorithms application solving the linear network problems. In the future research, we plan to extend our hardware

architecture to other problems in linear network field as well. For instant, shortest path problem can be converted to assignment problem so that our architecture of FAA with minor changes can be deployed to solve the problem efficiently. Besides network problems, we will also deploy our module as a building block for data mining in large datasets.

Bibliography

[1] J. Munkres, "Algorithms for the assignment and transportation problems," *Journal of the Society for Industrial and Applied Mathematics*, vol. 5, no. 1, pp. 32–38, 1957.

[2] M. Hung, "A polynomial simplex method for the assignment problem," *Operations Research*, vol. 31, no. 3, pp. 595–600, 1983.

[3] A. Sahu and R. Tapadar, "Solving the Assignment problem using Genetic Algorithm and Simulated Annealing," *IAENG International Journal of Applied Mathematics*, vol. 36, no. 1, pp. 37–40, 2007.

[4] J. Wang, "Analogue neural network for solving the assignment problem," *Electronics Letters*, vol. 28, no. 11, pp. 1047–1050, 1992.

[5] H. Kuhn, "The Hungarian method for the assignment problem," *Naval research logistics quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955.

[6] D. Bertsekas, "A distributed asynchronous relaxation algorithm for the assignment problem," in *Decision and Control, 1985 24th IEEE Conference on*, vol. 24. IEEE, 1985, pp. 1703–1704.

[7] D. Bertsekas, "Auction algorithms for network flow problems: A tutorial introduction," *Computational Optimization and Applications*, vol. 1, no. 1, pp. 7–66, 1992.

[8] D. Bertsekas, "The auction algorithm: A distributed relaxation method for the assignment problem," *Annals of Operations Research*, vol. 14, no. 1, pp. 105–123, 1988.

[9] S. Che, J. Li, J. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute-intensive applications with gpus and fpgas," 2008.

[10] D. Hung and J. Wang, "A FPGA-based custom computing system for solving the assignment problem," in *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*. IEEE, 1998, pp. 298–299.

[11] D. Hung and J. Wang, "Digital hardware realization of a recurrent neural network for solving the assignment problem," *Neurocomputing*, vol. 51, pp. 447–461, 2003.

[12] C. Vasconcelos and B. Rosenhahn, "Bipartite graph matching computation on GPU," in *Energy Minimization Methods in Computer Vision and Pattern Recognition*. Springer, 2009, pp. 42–55.

[13] D. Chen, J. Cong, and P. Pan, "Fpga design automation: A survey," *Foundations and Trends in Electronic Design Automation*, vol. 1, no. 3, pp. 139–169, 2006.

[14] Xinlix, "Virtex-5 family overview," http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf.

[15] Xinlix, "Virtex-5 user guide," http://www.xilinx.com/support/documentation/user_guides/ug190.pdf.

[16] B. Schwartz, "A computational analysis of the auction algorithm," *European journal of operational research*, vol. 74, no. 1, pp. 161–169, 1994.

[17] D. Bertsekas and D. Castañon, "A forward/reverse auction algorithm for asymmetric assignment problems," *Computational Optimization and Applications*, vol. 1, no. 3, pp. 277–297, 1992.

[18] J. Wein and S. Zenios, "Massively parallel auction algorithms for the assignment problem," in *Frontiers of Massively Parallel Computation, 1990. Proceedings., 3rd Symposium on the*. IEEE, 1990, pp. 90–99.

[19] D. Bertsekas and D. Castanon, "Parallel synchronous and asynchronous implementations of the auction algorithm*," *Parallel Computing*, vol. 17, no. 6-7, pp. 707–732, 1991.

[20] N. Bagherzadeh and K. Hawk, "Parallel implementation of the auction algorithm on the intel hypercube," in *Parallel Processing Symposium, 1992. Proceedings., Sixth International*. IEEE, 1992, pp. 443–447.

[21] A. Naiem, M. El-Beltagy, and M. Rasmy, "A centrally coordinated parallel auction algorithm for large scale assignment problems," in *Informatics and Systems (INFOS), 2010 The 7th International Conference on*. IEEE, 2010, pp. 1–4.

[22] D. Bertsekas and D. Castañon, "Parallel synchronous and asynchronous implementations of the auction algorithm*," *Parallel Computing*, vol. 17, no. 6-7, pp. 707–732, 1991.

[23] M. Dell'Amico and P. Toth, "Algorithms and codes for dense assignment problems: the state of the art," *Discrete Applied Mathematics*, vol. 100, no. 1-2, pp. 17–48, 2000.

Appendix A

Details and codes for the Processing Element module

The processing elements play an important role as common CPUs in the design. It is to compute the net value, $a_{ij} - p_j$. All the PEs have identical computation task in the bidding phase, so they share the same control signals. Since the regularity of a PE, we can duplicate it into one dimension array to parallel process data.

The following codes describe the functions it implements and its structure:

module PEunit(dout, clk, reset, sel, addr, din, dinp);
parameter data_width=16, addr_width=6; // Each PEunit has 2$\hat{6}$=64 memory size to store price
output [data_width-1:0] dout;
input [data_width-1:0] din, dinp;
input clk, // clock
reset, // reset LUT memory to zero when assignment is complete, '1' is effective
sel; // selected PEunit is to update price[addr]
input [addr_width-1:0] addr; // used to read or write price[addr]

wire [data_width-1:0] lut_dout; // lut_dout = price[j]
wire [data_width-1:0] lut_din; // update price[] or reset price[] to zero
wire lut_we; // lut_we: update or reset
wire [data_width-1:0] bid_val; // bid_val=pj+vij-wij+epsilon

```
reg [data_width-1:0] lut_dout_reg;

assign bid_val = dinp + lut_dout_reg;
assign lut_din = reset ? dinp : bid_val; // When PEunit is selected, update price[]
assign lut_we = sel — reset;
ram64x16s
ram64x16s(.dout(lut_dout),
.clk(clk),
.we(lut_we),
.din(lut_din),
.addr(addr)
);
reg [data_width-1:0] bid;

always @(posedge clk)
begin
bid ¡= din - lut_dout ; // bid = aij - pj
lut_dout_reg ¡= lut_dout;
end
assign dout=bid;

endmodule
```

Appendix B

Details and codes for the Search Task module

Search Task module is a two-stage module. Each stage uses the same comparison tree except different inputs. So to build a comparison tree is important and also tricky. We do not want that the comparison tree is not pipelined or fully pipeline, which leads to decreasing clock frequency or high latency. Instead, we design the comparison tree that is half pipelined with scalability.

The following codes describe the functions it implements and its structure:

```
module comparisontreepipeline(best_bid,block_inner_j,clk,data_in);

parameter data_width=16, bits_num_PEs=16'd4, num_PEs=2**bits_num_PEs;
parameter levels_delay=bits_num_PEs/2+bits_num_PEs%2; // If comparison tree is fully pipelined, then the levels of delays are bits_num_PEs. If halfly pipelined, then it is.

output [data_width-1:0] best_bid; // compute the best bid in parallel
output [data_width-1:0] block_inner_j; // the index of PEunit
input [data_width*num_PEs-1:0] data_in; // num_PEs input
input clk;

// comparison tree register declaration
```

```
wire [data_width*(2*num_PEs-1)-1:0] cmp_out;
reg [data_width*(2*num_PEs-1)-1:0] cmp_out_reg;
wire [data_width*(2*num_PEs-1)-1:0] index_out;
reg [data_width*(2*num_PEs-1)-1:0] index_out_reg;

genvar j,i;
generate
begin
for(j=0;j¡bits_num_PEs;j=j+1)
begin: outer_loop
if(j%2==1&&j!=bits_num_PEs-1)
begin
for(i=0;i<2**j;i=i+1)
begin: inner_loop_odd
comparator2 #(
.data_width(data_width)
)
cmp(
.dout(cmp_out[data_width*((2**j)+i)-1:data_width*((2**j)-1+i)]),
.index_out(index_out[data_width*((2**j)+i)-1:data_width*((2**j)-1+i)]),
.din1(cmp_out_reg[data_width*((2**j)*2+i*2)-1:data_width*((2**j)*2+i*2-1)]),
.din2(cmp_out_reg[data_width*((2**j)*2+i*2+1)-1:data_width*((2**j)*2+i*2)]),
.indexin1(index_out_reg[data_width*((2**j)*2+i*2)-1:data_width*((2**j)*2+i*2-1)]),
.indexin2(index_out_reg[data_width*((2**j)*2+i*2+1)-1:data_width*((2**j)*2+i*2)])
);
end
end
else
begin
for(i=0;i<2**j;i=i+1)
```

```
begin: inner_loop_even
comparator2 #(
.data_width(data_width)
)
cmp(
.dout(cmp_out[data_width*((2**j)+i)-1:data_width*((2**j)-1+i)]),
.index_out(index_out[data_width*((2**j)+i)-1:data_width*((2**j)-1+i)]),
.din1(cmp_out[data_width*((2**j)*2+i*2)-1:data_width*((2**j)*2+i*2-1)]),
.din2(cmp_out[data_width*((2**j)*2+i*2+1)-1:data_width*((2**j)*2+i*2)]),
.indexin1(index_out[data_width*((2**j)*2+i*2)-1:data_width*((2**j)*2+i*2-1)]),
.indexin2(index_out[data_width*((2**j)*2+i*2+1)-1:data_width*((2**j)*2+i*2)])
);
end
end
end
end
endgenerate

// The outputs of comparator2 is stored in regiesters
always @(posedge clk)
begin
cmp_out_reg <= cmp_out;
index_out_reg <= index_out;
end assign cmp_out[data_width*(2*num_PEs-1)-1:data_width*(num_PEs-1)] = data_in;

generate
begin
for(j=0;j<16'h0001<<(bits_num_PEs);j=j+1)
begin: assign_index
assign index_out[data_width*(num_PEs+j)-1:data_width*(num_PEs-1+j)] = j;
```

```
end
end
endgenerate
assign best_bid = cmp_out_reg[data_width-1:0];
assign block_inner_j = index_out_reg[data_width-1:0];


    endmodule


    // comparator module
module comparator2(dout,index_out,din1,din2,indexin1,indexin2);
parameter data_width=16;
output [data_width-1:0] dout;
output [data_width-1:0] index_out;
input signed [data_width-1:0] din1,din2;
input [data_width-1:0] indexin1, indexin2;

wire sel_out=din1>=din2;
reg [data_width-1:0] dout;
always @(din1 or din2 or sel_out)
begin
if(sel_out)
dout = din1;
else
dout = din2;
end
assign index_out = sel_out ? indexin1 : indexin2;
```

```
endmodule
```