Arquitectura Hardware en FPGA para el Minado de Conjuntos Frecuentes en Conjuntos de Datos Usando una Estrategia de Segmentación

Por:

Mauro Martín Letras Luna

Tesis sometida como requisito parcial
para obtener el grado de:

MAESTRÍA EN CIENCIAS EN EL ÁREA DE CIENCIAS COMPUTACIONALES

en el

Instituto Nacional de Astrofísica, Óptica y Electrónica

Tonantzintla, Puebla

Dirigida por:

Dr. René Armando Cumplido Parra
Dr. Raudel Hernández León

HARDWARE ARCHITECTURE FOR FREQUENT ITEMSET MINING IN STATIC DATASETS USING A SEGMENTATION STRATEGY

Mauro Martin Letras Luna

Thesis submitted in partial fulfillment of a
M.Sc. degree in Computer Sciences

Advisors
PhD. René Cumplido Parra, National Institute of Astrophysics, Optics and Electronics, INAOE, México
PhD. Raudel Hernández León, Advanced Technologies Application Center, CENATAV, Cuba

M.Sc. committee
PhD. José Francisco Martínez Trinidad
PhD. Leopoldo Altamirano Robles
PhD. Miguel Octavio Arias Estrada

Computer Sciences Department
National Institute of Astrophysics, Optics and Electronics
INAOE
Sta. María Tonantzintla, November 2015

Hardware Architecture for Frequent Itemset Mining in Static Datasets using a Segmentation Strategy

Thesis submitted in partial fulfillment of a M.Sc. degree in Computer Sciences

Computer Sciences Department
National Institute of Astrophysics, Optics and Electronics
INAOE
Luis Enrique Erro No. 1, Sta. María Tonantzintla
72840, San Andrés Cholula,
México

Telephone: (222) 266.31.00 Ext 8302/8308

Bibliographic information:
Mauro Martin Letras Luna, 2015, Hardware Architecture for Frequent Itemset Mining in Static Datasets using a Segmentation Strategy, M.Sc. thesis, Computer Sciences Department, National Institute of Astrophysics, Optics and Electronics, INAOE.

Sta. María Tonantzintla, Puebla, México, November 2015

I am not fully known to myself because
part of what I am is the enigmatic traces of others

Abstract

In recent years there has been a significant increase in the information generated from distinct domains and the size of datasets overwhelm the human capacity to process them and obtain valuable information. Because of this, Data Mining has emerged as a set of techniques and algorithms dedicated to finding patterns in datasets, and then these patterns are used to classify or predict the behavior of some phenomena related to the data. Association Rules Mining is an important branch inside Data Mining, and it consists in finding relationships among the data in the form of implication rules. The problem is usually decomposed into two subproblems. One is to find those itemsets whose occurrences exceed a predefined threshold in the database; those itemsets are called frequent itemsets. The second problem is to generate association rules from those frequent itemsets.

In this research, Frequent Itemset Mining is explored, because the huge amount of data in some cases makes difficult to obtain a response in an acceptable time according to the application requirements, due to the exhaustive nature of the problem. There are many algorithms dedicated to searching frequent itemsets, the most widely used are: Apriori, FP-Growth, and Eclat. They use strategies like breadth-first search and depth-first search to go over to the search space. They have to do a search in datasets, some of them like Apriori, have to access many times the dataset. FP-Growth reads the dataset twice, but it must keep in memory large amounts of data. Frequent Itemset Mining is an exhaustive task since the database must be read many times independently of the way in which the data is stored (in main memory or hard disk). In the literature, there have been reported two ways to accelerate Frequent Itemset Mining: the first one consists in improving the existing software algorithms through proposing new heuristics to save time, and the second one consists in developing hardware architectures dedicated to this task.

The main goal of this research is to design a Hardware Architecture to accelerate the Frequent Itemsets Mining process. A segmentation strategy is proposed using equivalence classes to guarantee that all the frequent itemsets will be found independently of the available hardware resources. An implementation in FPGA will

be carried out to validate the proposed architecture and compare it with software only implementations.

Keywords

FPGA, Hardware Architecture, Data Mining, Frequent Itemsets, Eclat

Resumen

En años recientes ha habido un incremento significativo de la información generada en distintos dominios y el tamaño de dichos conjuntos de datos sobrepasa la capacidad humana para procesarlos y obtener información útil. En consecuencia, la Minería de Datos ha emergido como un conjunto de técnicas y algoritmos dedicados a encontrar patrones en grandes conjuntos de datos, después estos patrones son utilizados para clasificar o predecir algún fenómeno relacionado con los datos. La Minería de Reglas de Asociación es un área importante dentro de la Minería de Datos y se divide en: Minería de Conjuntos Frecuentes y Generación de Reglas de Asociación.

En este trabajo de investigación se aborda el problema de la Minería de Conjuntos Frecuentes, porque en algunos casos los algoritmos no generan una respuesta en un tiempo aceptable de acuerdo a los requerimientos de una aplicación específica. Existen varios algoritmos reportados en la literatura, entre los cuales se destacan: Apriori, Eclat y FP-Growth. Estos algoritmos utilizan dos métodos para la exploración del espacio de búsqueda: (1) primero en profundidad y (2) primero en anchura. Algoritmos como Apriori tiene que realizar varias lecturas del conjunto de datos. Estrategias basadas en FP-Growth leen el conjunto de datos sólo en dos ocasiones pero tienen que almacenar grandes cantidades de información en memoria. La minería de conjuntos frecuentes se vuelve una tarea exhaustiva debido a que el conjunto de datos tiene que ser leído en repetidas ocasiones independientemente de donde sea almacenado (en memoria o en disco duro). En la literatura se han reportado dos maneras para acelerar esta tarea. La primera consiste en mejorar los algoritmos propuestos mediante nuevas heurísticas y la segunda consiste en el desarrollo de arquitecturas hardware de propósito específico.

El principal objetivo de esta investigación es diseñar una Arquitectura Hardware para acelerar la Minería de Conjuntos Frecuentes. Se propone una estrategia de segmentación basada en clases de equivalencia que garantiza encontrar todos los conjuntos frecuentes independientemente de los recursos de hardware disponibles. Se realiza una implementación en FPGA para validar y comparar a arquitectura propuesta contra implementaciones en software reportadas en la literatura.

Palabras Clave:
FPGA, Arquitectura Hardware, Minería de Datos, Minería de Conjuntos Frecuentes, Eclat

Contents

List of Figures

List of Tables

Abbreviations

Poset: Partial Ordered Set

FPGA: Field Programmable Gate Array

GPU: Graphic Processor Unit

KDD: Knowledge Discovery in Databases

SVM: Support Vector Machine

ECLAT: Equivalence Class Transformation Algorithm

HDL: Hardware Description Language

HLS: High Level Synthesis

LUT: Look Up Table

RTL: Register Transfer Level

DHP: Direct Hashing and Pruning

1

Introduction

In this chapter, we present the motivation for this research work, the context, and the importance to investigate the Frequent Itemset Mining. Frequent Itemset Mining is an elemental part of the Association Rules Mining, and it is a time demanding process due to the exhaustive search of itemsets in datasets. In this research, acceleration by hardware is proposed. In the literature, several hardware architectures have been proposed but most of them have limitations when dealing with many items and transactions, due to the physical limitations of the hardware platform used to implement them and memory constraints. The objective of this dissertation is to design and implement a hardware architecture able to deal with different datasets, no matter how many transactions or items they have. In the next lines, the hypothesis and research questions are shown. The main objective, specific objectives, and the contribution are explained in a more detailed form.

Nowadays, information technology is present in every activity that we perform: in smartphones, personal computers, and even household appliances connected to the Internet that interchange and generate big amounts of data. It has been reported that

2.5 exabytes of new information are generated per day [38]. This data comes from bills, medical reports, tax declarations, scientific observations, entertainment, social networks, and others. In the last two years, 90 % of the information in the world has been generated, and it is expected that this amount will increase because of the massive proliferation of smartphones and mobile devices [45]. For this reason, it is necessary to develop better strategies to work with big amounts of data to obtain non-trivial useful information [47]. Every day, Google alone processes around 24 petabytes (or 24,000 terabytes) of data [20], and the social network Twitter generates around 90 millions of twits per day in all the world, where each twit might contain 140 characters [48]. Another example out of internet context is Walmart [38]; this supermarket chain produces 2.5 petabytes per hour from the transactions generated by their customers. This amount of data and the diversity of the information exceed the human capacity to process and obtain rules that describe the relationship among the data. Traditional methods do not get the expected results in an acceptable time. As consequence, it is necessary to develop strategies capable of obtaining information not seen at first sight on those big amounts of data [25].

Data Mining solves this problem using automatic or semi-automatic processes to analyze huge datasets to find patterns and then performs classification or prediction tasks [61]. There are Data Mining algorithms used to determine Association Rules in the form of implications among the items of the datasets [3]. A crucial step in the Association Rules Generation is to count the frequency of items and itemsets to know their relevance; this process is known as Frequent Itemset Mining. The frequent itemsets are those set of items that are frequent in the whole dataset; in other words, their frequency is higher than certain support threshold (henceforth s_{min}) [3].

Nevertheless, looking for frequent itemsets may become an expensive task in time due to the big amount of data, sparse datasets, and low s_{min} value. For these reasons, sometimes the implementations of these algorithms cannot return a solution in an acceptable time. When a big amount of data is mentioned, it refers to datasets with thousands of different items and millions of transactions. One way to deal with this problem is improving existing algorithms by reducing execution time and proposing new heuristics to explore the search space or using different data representations. Another alternative is to use supercomputer or clusters to perform Frequent Itemset Mining in a concurrent or parallel manner. Although, a speed up execution time of algorithms is reached, a disadvantage is the power consumption of one supercomputer

or cluster.

In recent years, there is a trend to develop specialized hardware architectures to reduce the execution time because, a hardware implementation of one algorithm is faster than its software counterpart, and the power consumption of the employed devices is lower compared to a supercomputer. In literature, there are several hardware architectures based on FPGA (Field Programmable Gate Arrays) and GPUs (Graphic Processor Units) used as co-processors to be in charge of specific tasks such as counting s_{min} and, other more sophisticated ones that perform a full implementation of Frequent Itemset Mining algorithms. They take advantage of the inner parallelism of GPU and FPGA to accelerate the searching process [39, 51, 54, 55].
This thesis proposes to develop a Hardware Architecture for Frequent Itemsets Mining based on FPGA to take advantage of inner parallelism, and the main goal is to improve efficiency that it will be measured regarding the amount of data processed per unit area and overall processing time when compared with software-only implementations. The main contribution is to approach this problem in an incremental way; the idea is to develop a Hardware Architecture that is not dependent on the problem size (space of solutions). Most of the reported work in literature has been though for a fixed problem size, in others words they have a limit on the number of different items that are processed by the Architecture, limited by the resources of the device employed and by memory constraints. To achieve this goal, the Hardware Architecture must be able to segment the problem, then generate partial solutions for each partition, and finally combine all the partial solutions to construct a global solution. The architecture is mainly based on equivalence classes where the equivalence relationship is the prefix of each frequent itemset. All the equivalence classes may be processed in an independent way due to the *downward closure property* [3] of frequent itemsets. In consequence, parallel processors might be implemented to accelerate the processing of each equivalent class and increase the performance.

1.1 Problem Description

Frequent Itemset Mining in datasets becomes an expensive computational task due to the big amount of information that is stored in those datasets. It has been previously mentioned, that there is an increasing interest in developing new strategies to manage those datasets, and the trend indicates that those datasets will continue increasing due to the emerging technologies.
Another issue is how to go over the searching space to generate the frequent itemsets. Some well-known strategies like Apriori [3] perform Breadth First Search based in a heuristic to determine frequent itemsets candidates. In this particular case, Apriori performs several reads from the dataset to perform candidate generation, and this becomes the most demanding task in the algorithm. Strategies based on a depth-first search like Eclat [68] only need to read the entire dataset twice, but the intermediate structures to store the frequent itemsets become too large to be practically stored in memory. Strategies like FP-Growth [29] employ a tree structure in main memory to store the set of transactions, then the Frequent Itemset Mining is accelerated but sometimes the tree structure becomes impractical to be stored in memory. Although, there are many strategies for Frequent Itemset Mining, sometimes the algorithms do not return a response in an acceptable time due to the number of items or transactions in datasets, this becomes intractable because in some scenarios a response as early as possible is required. Another important issue is when the minimum support is low because, in theory, the number of candidate itemsets is 2^n, being n the number of different items in the dataset. Then the searching space could be, in the worst case, an exponential search space that includes all the possible frequent itemsets. For these reasons, it is necessary to develop strategies to speed up Frequent Itemsets Mining and guarantee that all the Frequent Itemsets will be found.

1.2 Research Questions

In this dissertation, the following questions are planned to be answered:

1. What is necessary to accelerate the Frequent Itemset Mining and obtain an efficient Architectural Design?

2. What are the advantages of using Equivalence Classes as segmentation strategy?

3. What control schemes are required to guarantee that all the frequent itemsets will be found?

1.3 Hypothesis

The partition of the search space of itemsets into equivalence classes will make possible to design a Hardware Architecture for Frequent Itemset Mining. Such Architecture will be able to parallelize Frequent Itemset Mining regardless of how many distinct items and transactions are present in the dataset.

1.4 Main Objective

Design an FPGA-based Hardware Architecture to accelerate Frequent Itemset Mining in datasets. The architecture must extract frequent itemsets faster that software only implementations and be able to process partitions of the search space based on equivalence classes.

1.5 Specific Objectives

- Propose a search strategy using equivalence classes to make partitions of the search space.

- Propose implementations based on standalone processing modules.

- Evaluate the proposed architectures in terms of processing time and used hardware resources.

1.6 Contribution

The first contribution of this research work is to propose a search strategy that can partition the search space of solutions and can deal with memory constraints because most of the works reported in the literature have been implemented for a fixed problem size, limited by the resources of the FPGA device employed and memory restrictions. The second contribution is to speed up the frequent itemset mining problem taking advantage of the inner parallelism of FPGA device. The proposed architecture must be faster than the software-only implementations reported in literature.

1.7 Summary

This chapter approached the motivation to develop this research and the context of the Frequent Itemset Mining task. An important reason to approach this problem is the big amount of data in data sets and the diversity of those data. The main contribution of this research is to set forth a Hardware Architecture that can deal with different datasets regardless of the number of transactions, the number of items or the resources available from the hardware device employed. In this chapter, it has been exposed the hypothesis and the specific objectives to aim this goal.

1.8 Document Organization

This thesis document is organized as follows. Chapter two describes the theoretical framework of Frequent Itemset Mining, the basic concepts, and terminology of Frequent Itemset Mining is disclosed, including a formal definition of frequent itemset and the

main algorithms found in the literature. Chapter three includes a revision of the state of the art and the implementations of Apriori, Eclat and FP-Growth in FPGA and GPU devices. In the same way, a review of advanced hardware techniques employed to approach this problem is presented like Systolic Arrays and Content Addressable Memories. In Chapter four, the details and design of the proposed architecture and the algorithm employed are shown; also the methodology employed is described in a set of steps. Chapter five includes a description of the employed metrics, the experimental results and the analysis of results showing the advantages of the proposed architecture. Finally in chapter six, the conclusions, and future work are presented.

2

Theoretical Framework

In this chapter, the basic concepts of Frequent Itemset Mining and the definitions needed to understand the theoretical framework are presented. A brief explanation of Data Mining and their main applications are shown to stand out the importance of Pattern Recognition Tasks in real world applications and the benefits that they imply. Then, a formal definition of the frequent item and frequent itemset are shown and concepts like minimum support, frequency and cover are described in a more detailed form. In the following section, the minimum support value is analyzed and how this value affects the number of frequent itemsets in the search space is explained. Later, a complete revision of Frequent Itemset Mining Algorithms is presented, and a classification is shown according to the way that algorithms explore the search space (breadth-first search or depth-first search) and, according to it if the algorithm is a serial one or a parallel one. Finally, a revision of the most important algorithms presents the benefits and disadvantages of each of these algorithms.

2.1 Data Mining

The information generated in our daily activities continues growing, and there is not an apparent end. Namely, the capacity of storing devices continues increasing, and they are becoming more affordable. Furthermore, there are a lot of on-line storage services that make easy to keep all the generated information. In the same way, devices connected to the internet like laptops, smartphones, tablets, and televisions record our decisions, our musical preferences, our choices in the supermarket, our financial habits and our academic records. In other words, as individuals, we generate a lot of information during our entire life. And all the previous examples are just information generated by personal choices, but the amount generated in business, commerce and industry is also very significant. We are witnessing the growing difference between data generation and the understanding of it. As the volume of data increases, inexorably, the proportion of it that people understand decreases alarmingly [61]. Consequently, it becomes a challenge to the human capacity to deal with those big amounts of data. For this reason, automatic or semi-automatic algorithms executed in computers have been proposed to extract non-trivial useful information (knowledge) from datasets. Data Mining and Knowledge Discovery in Datasets are branches in Computer Sciences dedicated to finding non-trivial information from big amounts of data. There are several definitions of Data Mining for example:

Witten et. al. [61] *Data mining is defined as the process of discovering patterns in data. The process must be automatic or (more usually) semiautomatic. The patterns discovered must be meaningful in that they lead to some advantage, usually an economic one. The data is invariably present in substantial quantities. Data Mining is about solving problems by analyzing data already present in datasets.*

Likewise, Olson and Delen [40], *Data mining has been called exploratory data analysis, among other things. Masses of data generated from cash registers, from scanning, from topic specific datasets throughout the company, are explored, analyzed, reduced, and reused. Searches are performed on different models proposed for predicting sales,*

marketing response, and profit. Classical statistical approaches are fundamental to Data Mining.

In Berry and Linoff [8], *Data Mining is the process of discovering meaningful correlations, patterns, and trends by sifting through large amounts of data stored in repositories. Data Mining employs pattern recognition technologies, as well as statistical and mathematical techniques.*

All the previous definitions agree that Data Mining focuses on extracting knowledge from big data repositories, based on the premise that we are rich in data but poor in information. The information could be used to take a decision and to obtain a benefit (in most of the cases an economical one). For this reason, a set of algorithms and automatic or semi-automatic methods are proposed to approach this challenge. Data Mining is used to take decisions, classify and predict a trend based on the information discovered. Data Mining is an inner part of Knowledge Discovery in Datasets. The steps involved in Knowledge Discovery in Datasets are shown in figure 2.1. The first step consists in understanding the context and the application domain and the prior information to be conscious of the final target of the application.

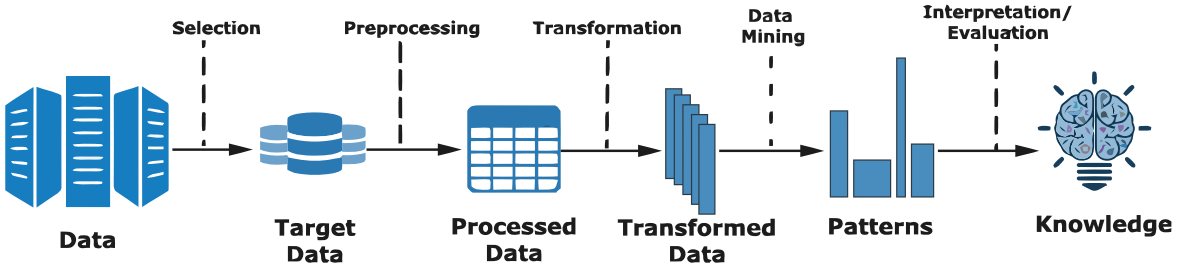


Figure 2.1: Steps involved in knowledge discovery in datasets

The first step also concerns about create a dataset, collecting instances, or selecting those representative instances or attributes to construct a dataset, on which information discovery will be performed. The second step is to perform a pre-processing, the most common task performed at this stage is to remove the noise, and choose a strategy to deal with missing values or use a data representation according to the re-

quirements of the algorithms. The third step is data reduction and projection: finding useful features to represent the data depending on the goal of the task with dimensionality reduction or transformation. The fourht step is to carry out a Data Mining method. For example *decision trees, clustering, association rules generation, SVM, among others.* This stage also includes deciding which models and parameters are the most appropriated. The fifth step consists of interpreting the mined patterns and taking a decision based on the discovered knowledge, incorporating the knowledge to another system[24].
Although, Data Mining is an important part of KDD, there are different types of Data Mining systems according to the context and the specific analysis that they perform. The following section focuses on Frequent Itemset Mining, its applications in the real world, and the impact that Frequent Itemset Mining have in distinct domains are discussed.

2.2 Frequent Itemset Mining

Frequent Itemset Mining is a method for Market Basket Analysis, and was introduced in [2] by Agrawal. In Market Basket Analysis the main goal is finding patterns in the shopping behavior of customers; in other words, finding the set of products that are frequently bought together. The obtained patterns are used in the Association Rules Generation. An Association Rule is expressed in the form of implication. For example, if a customer purchases tires and auto accessories then he probably gets automotive service done [3]. Finding frequent itemsets and associations rules is essential for marketing applications, improving the arrangement of products on shelves and suggesting other products. The datasets involved in these applications are usually large. Therefore, it is important to develop fast algorithms for this task due to a large amount of information in datasets. The first algorithm for Frequent Itemset Mining was formalized by Agrawal in the 90's [2, 3], and it is used to find patterns in datasets. These datasets are represented by transactions; each transaction is labeled with a unique identifier. Frequent Itemset Mining can be defined as follows: Formally, let $I = \{i_1, \ldots, i_n\}$ be a set of items. Let D be a set of transactions, where each transaction T is a set of items such as $T \subseteq I$. And let X be an itemset such as $X \subseteq I$; without loss of generality, we

will assume that all items in each transaction are sorted in lexicographic order. The support value of the itemset X is the number of transactions over D containing X. An itemset is called frequent if its support is greater than or equal to a given support threshold (S_{min}). For example in table 2.1. If s_{min} = 50 % (0.5*6 = 3 occurrences)

Table 2.1: Example of transactional dataset.

ID	Items
1	Milk, Bread
2	Butter, Bread
3	Butter
4	Milk, Bread, Butter
5	Bread
6	Milk, Bread, Butter

Table 2.2: Itemsets and their support and frequency in the dataset.

Itemset	Cover	Support	Frequency
{ }	1,2,3,4,5,6	6	100.00%
{Bread}	1,2,4,5,6	5	83.00%
{Butter}	2,3,4,6	4	66.67%
{Milk}	1,4,6	3	50.00%
{Bread, Milk}	1,4,6	3	50.00%
{Butter, Bread}	2,4,6	3	50.00%
{Butter, Milk}	4,6	2	33.33%
{Bread, Butter, Milk}	4,6	2	33.33%

the Frequent Itemsets are: {Bread}, {Butter}, {Milk}, {Bread, Milk}, and {Bread, Butter} because they are present at least three times in the dataset. The cardinality of an itemset defines its size, an itemset of size k is called a $k-itemset$. As consequence, a brute force approach that traverses all the possible itemsets calculating their support and removing infrequent itemsets is inefficient. The number of itemsets and operations grows exponentially according to the number of different items in the dataset and the s_{min} value. For example, the catalog of a supermarket is around thousands of different products. For example, in figure 2.2 a universe of five items is shown, and a lattice represents the search space of itemsets. Table 2.3 shows the employed transactions in this example. The $S_{min} = 3$, and the gray boxes represent frequent itemsets. The figure depicts that it is not necessary to traverse all the possible itemsets and that a brute

Table 2.3: Transaction dataset example.

TID	Itemset
1	{a,d,e}
2	{b,c,d}
3	{a,c,e}
4	{a,c,d,e}
5	{a,e}
6	{a,c,d}
7	{b,c}
8	{a,c,d,e}
9	{b,c,e}
10	{a,d,e}

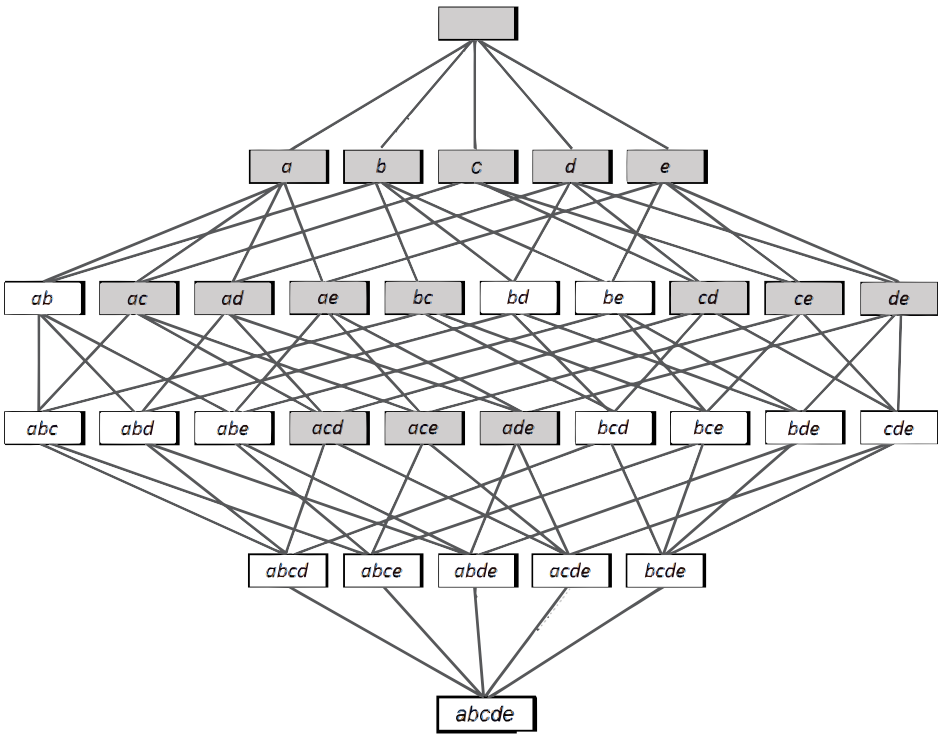


Figure 2.2: Complexity of search space of dataset in table 2.3 represented in a Hasse diagram, the $S_{min} = 3$ and the gray ones represent frequent itemsets

force approach is unnecessary. Agrawal in his article [3] notes certain characteristics of frequent itemsets and their support that are useful in the generation of frequent itemsets and it avoids to traverse all the possible itemsets. The downward clousure says that if an itemset is extended, its support cannot increase [3]. For example, if

the itemset $A = \{a, b, h\}$, and its $s_T(A) = 10$, and there is an itemset $B = \{a, b, h, j\}$. In consequence, the value of $s_T(B)$ at most will be 10. With the previous property is inferred that no superset of an infrequent itemset can be frequent. This property is also called the **Apriori Property** [3], this property is very helpful, for example, all the items that are nor frequent could be removed because they never will be a frequent itemset. As its name suggests, with previous information of the support value of the items, it is possible to reduce the search space.

2.3 Data Representation

The Frequent Itemset Mining Algorithms could be classified according to the way they explore the search space. Some explore the search space using a depth-first search while others use breadth-first search. According to the employed strategy, there are four types of representing the datasets like a matrix.

Horizontal Items Vector (HIV): The transactions are organized in a set of rows, each row stores a transaction identifier (TID) and a binary vector, where 1 represents the occurrence of an item and 0 represents the absence of an item in the dataset.

Horizontal Items List (HIL): this representation is similar to HIV the only difference is that each row stores a sorted list of item identifiers (ID), holding only the items that belong to the transaction.

Vertical ID Vector (VIV): the transactions are represented as a set of columns associated with the items, each column stores a transaction ID and a binary vector which represent the presence or absence of an item in each transaction.

Vertical Transaction List (VTL): this representation is similar to VIV, the only difference is that each column stores a sorted list of identifiers (idList), holding only the transactions which the item is present.

The main advantage of vertical representation is that a transaction list for a pair of items could be calculated by intersecting the transaction lists of the individual items.

TID	Itemset
1	a, d, e
2	b, c, d
3	a, c, e
4	a, c, d, e
5	a, e
6	a, c, d
7	b, c
8	a, c, d, e
9	b, c, e
10	a, d, e

(a) Horizontal list vector

a	b	c	d	e
1	2	2	1	1
3	7	3	2	3
4	9	4	4	4
5		6	6	5
6		7	8	8
8		8	10	9
10		9		10

(b) Vertical ID vector

Figure 2.3: Horizontal and vertical transaction ID representation.

	a	b	c	d	e
1	1	0	0	1	1
2	0	1	1	1	0
3	1	0	1	0	1
4	1	0	1	1	1
5	1	0	0	0	1
6	1	0	1	1	0
7	0	1	1	0	0
8	1	0	1	1	1
9	0	1	1	0	1
10	1	0	0	1	1

(a) Vertical items vector

	1	2	3	4	5	6	7	8	9	10
a	1	0	1	1	1	1	0	1	0	1
b	0	1	0	0	0	0	1	0	1	0
c	0	1	1	1	0	1	1	1	1	0
d	1	1	0	1	0	1	0	1	0	1
e	1	0	1	1	1	0	0	1	1	1

(b) Horizontal items vector

Figure 2.4: Horizontal and vertical binary representation.

A vertical transaction representation exploits the next property:

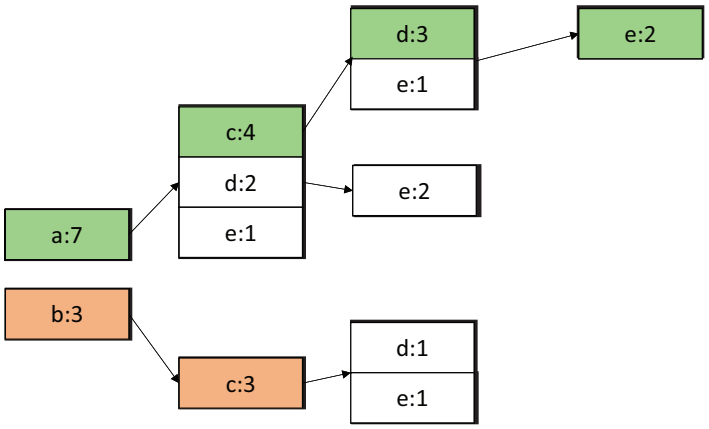
$$\forall I, \forall J, J \subseteq B : K_T(I \cup J) = K_T(I) \cap K_T(J). \tag{2.1}$$

For example, for set $A = \{a, b, c, f\}$ and set $B = \{a, b, c, f, h\}$. The cover is $K_T(A \cup B) = K_T(A) \cap K_T(B)$. So, the $K_T(A \cup B) = \{a, b, c, f\}$.

There is an alternative and compact representation used by the FP-Growth algorithm[29].

TID	Itemset
6	a, c, d
4	a, c, d, e
8	a, c, d, e
3	a, c, e
1	a, d, e
10	a, d, e
5	a, e
7	b, c
2	b, c, d
9	b, c, e

(a) Lexicographically sorted dataset



(b) Prefix tree representation

Figure 2.5: Prefix tree representation.

The prefix tree representation is a compressed horizontal representation, and it follows the principle that equal prefixes of transactions are merged. For example, in Figure 2.5 b), the itemset $A = \{a, c, d, e\}$ has a $s_T(A) = 2$, the support value is computed by following the path from a to e, the last item in the path contains the support value for the itemset. Then to calculate the value for $B = \{b, c\}$ the support value is $s_T(B) = 3$. This representation is most effective if the items are sorted in ascending order according to their support value[12].

2.4 Frequent Itemset Mining Algorithms

According to the data representation and the way that the search space is explored, different Frequent Itemset Mining algorithms have been proposed in the literature. Two main methodologies have been proposed to reduce the computational time cost. The first methodology proposes the generation and pruning of candidate frequent itemset in the search space while the second considers reducing the number of comparisons required to determine the itemsets support.

2.4.1 Candidate Generation

Previously, it has been mentioned that the first alternative for Frequent Itemset Mining is a brute-force approach for computing the support for every possible itemset. Given the set of items B and a partial order with respect to the subset operator, all possible candidate itemsets can be denoted by a Hasse diagram, as it has been presented previously in Figure 2.2. The brute force approach compares each candidate itemset with every transaction $t \in T$ to check for containment. An approach like this would require $O(|T| \cdot L \cdot |I|)$ item comparisons, where the number of non-empty itemsets in the Hasse diagram is $L = 2^n - 1$.

This type of computation becomes prohibitively expensive and sometimes it is not necessary to explore all the search space. One way to reduce computational complexity and reduce operations is reducing the number of candidate itemsets tested for support. In consequence, algorithms rely on the observation that every candidate itemset of size k is the union of two candidate itemsets of size $(k-1)$, this observation is a consequence of the **downward closure**. Then, the supersets of an infrequent itemset must be infrequent. Thus, given a minimum support value, there are itemsets in the Hasse diagram that they do not need to be explored "if and only if" their support value is lower than the minimum support value (S_{min}).

This technique is often referred as support-based pruning and was first introduced in the Apriori algorithm by Agrawal and Srikant [3].

Algorithm 1 shows the pseudo-code for Apriori-based Frequent Itemset Mining. Starting with each item (1-itemsets) as an itemset, the support for each itemset is calculated, and itemsets that do not meet the minimum support threshold s_{min} are remove.

The next iteration consists in two phases. In the first one, the large itemsets L_{k-1} found in the (k-1)th iteration are used to generate the candidate set C_k using the $apriori-gen$ function. The $apriori-gen$ performs the next operations shown in algorithm 2. For example, let $L_3 = \{\{1,2,3\},\{1,2,4\},\{1,3,4\},\{1,3,5\},\{2,3,4\}\}$, following algorithm 2, the itemsets $\{1,2,3\}$ and $\{1,2,4\}$ generate the candidate $C_1 = \{1,2,3,4\}$.

Table 2.4: Sets involved in Apriori Algorithm.

k-itemset	An itemset having *k* items.
L_k	Set of large *k-itemsets* (those with minimum support). Each member of this set has two fields: i)itemset and ii)support count
C_k	Set of candidates *k-itemsets* (potentially large itemsets). Each member of this set has two fields: i)itemset and ii)support count
C'_k	Set of candidates *k-itemsets* when the TIDs of the generating transaction are kept associated with the candidates.

Algorithm 1 Apriori Algorithm.

1: $L_1 = \{\text{Large 1-itemsets}\}$;
2: **for** $k = 2$ to $F_{k-1} \neq \emptyset$ **do**
3: $C_k = \text{apriori-gen}(L_{k-1})$; // New Candidates
4: **for all** transactions $t \in D$ **do**
5: $C_t = \text{subset}(C_k, \text{t})$; // Candidates contained in t
6: **for all** candidates $c \in C_t$ **do**
7: c.count++;
8: **end for**
9: **end for**$L_k = \{c \in C_k | c.count \geq s_{min}\}$
10: **end for**
11: **return** $\bigcup_k L_k$;

The itemsets $\{1, 3, 4\}$ and $\{1, 3, 5\}$ generate the candidate $C_2 = \{1, 3, 4, 5\}$. So, the candidates generated are C_1 and C_2, these candidates were generated following the **downward closure** but it does not warranty that the candidates are also frequent itemsets. The second step is the candidate pruning to obtain the frequent itemsets.

In algorithm 3, the candidate pruning is performed. In the previous example, C_1 and C_2 were selected as candidates. For C_1, the algorithm returns that C_1 is a frequent itemset because all the subsets of size 3 belongs to L_3. For C_2, the algorithm returns that C_2 is not a frequent itemset because $\{1, 4, 5\}$ does not belongs to the L_3 set.

This process is repeated until no more candidates could be generated. The search scheme used by Apriori is a breadth-fist one. All the candidates are generated by

Algorithm 2 Candidate Generation (apriori-gen)

Require: L_{k-1}
1: $C_k = \emptyset$; // Initializes the set of candidates
2: **for all** $f_1, f_2\ t \in L_{k-1}$ **do**
3: **if** $f_1 = \{i_1, ..., i_{k-1}, i_k\} \wedge f_2 = \{i_1, ..., i_{k-1}, i\prime_k\} \wedge i_k \leq i\prime_k$ **then**
4: $f = f_1 \cup f_2 = \{i_1, ..., i_{k-1}, i_k, i\prime_k\}$; // It explores all the pairs of frequent itemsets that differ in only one item and they are in lexicographic order
5: **if** $\forall i \in f : f - \{i\} \in L_{k-1}$ **then**
6: $C_k = C_k \cup f$;
7: **end if**
8: **end if**
9: **end for**
10: $C_k = prune(C_k)$;
11: **return** c_k;

Algorithm 3 Candidate Pruning (prune)

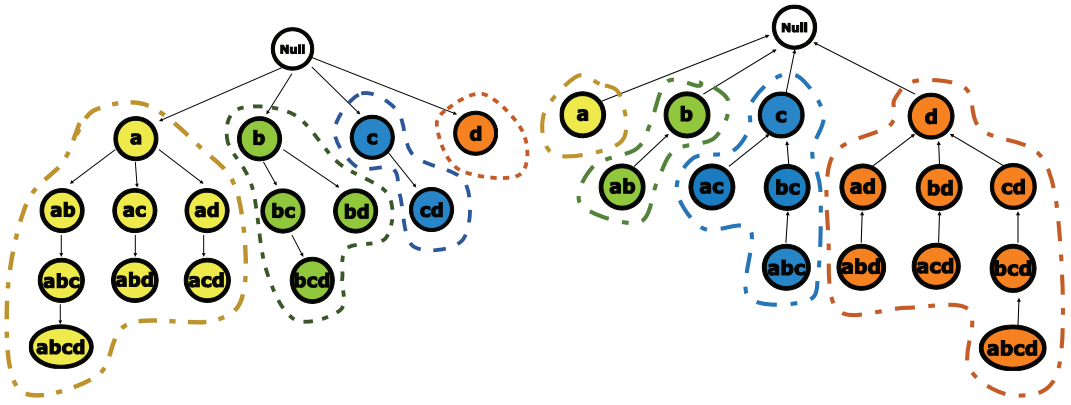
Require: C_k
1: **for all** $itemsetsc \in C_k$ **do**
2: **for all** $\{k-1\}$ subset s of c **do**
3: **if** $s \ni L_{k-1}$ **then**
4: delete c from C_k;
5: **end if**
6: **end for**
7: **end for**

merging itemsets that differ in only one item. The main advantage of Apriori is the pruning of infrequent candidates, compared against a brute force scheme, it reduces the workload, and it performs the search process avoiding segments in the Hasse diagram. The most remarkable disadvantages of this approach are that it can require a lot of memory resources to store all the set of candidates, and the support counting consumes a considerable execution time because the entire dataset must be read to verify the frequency of an itemset [11].

Since Apriori was proposed, several extensions have been proposed. Hashing Technique [17, 43], partitioning technique [49], parallel and distributed mining [37, 53], among others. All of them aimed at reducing the counting support time through reducing the number of dataset scans.

2.4.2 Pattern Growth

Apriori-based algorithms process candidates in a breath-first search manner, decomposing the itemset lattice(Hasse diagram) into level-wise itemset-size based in the idea that k-itemsets must be processed before (k + 1) itemsets. Assuming a lexicographic ordering of itemset, the search space can also be decomposed into prefix-based and suffix-based equivalence classes. Figures 2.5 shows equivalence classes for 1-length itemset prefixes and 1-length itemset suffixes, respectively. Once frequent 1-itemsets are discovered, their equivalence classes can be mined independently.

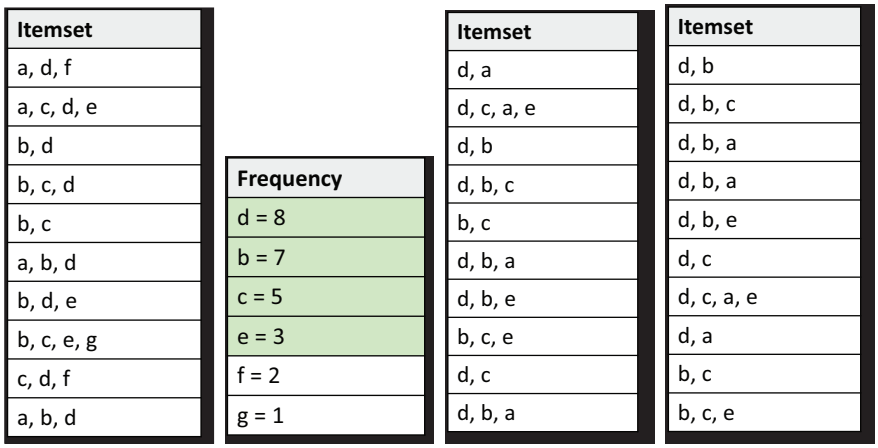


(a) Prefix Tree (b) Suffix Tree

Figure 2.6: In left, prefix tree showing prefix-based equivalence classes in the itemset lattice. In right, suffix tree showing suffix based equivalence classes in the itemset lattice.

Patterns are grown by appending appropriate items that follow the parent's last (first) item in a lexicographic order. Zaki [68] was the first to suggest prefix-based equivalence classes as a means of independent sub-lattice mining in his algorithm, Equivalence Class Transformation (Eclat). In order to improve candidate support counting, Zaki transforms the transactions into a vertical database format. Frequent 1-itemsets are then those with at least $\lceil s_{min}(T) \rceil$ listed transaction id lists. He uses lattice theory to prove that if two itemsets C_1 and C_2 are frequent, so their intersection set $C_1 \cap C_2$ will be frequent. After creating the vertical database, each equivalence class can be processed independently, in either breath-first or depth-first order, by recursive intersections of candidate itemset, while it still takes advantage of the downward closure property. For

example, assuming b is infrequent, we can find all frequent Items having prefix a by intersecting $tid-lists$ of a and c to find support for ac, then $tid-lists$ of ac and d to find support for acd, and finally $tid-lists$ of a and d to find support for ad. Note that the $ab-rooted$ sub-tree is not considered, as b is infrequent and will thus not be joined with a.



(a) Original Dataset (b) Frequency of Items (c) Items in transactions sorted descendent (d) Transactions sorted lexicographically

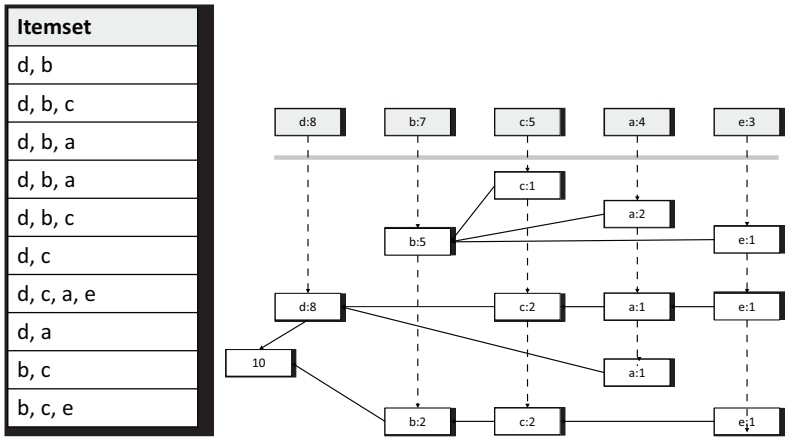
Figure 2.7: Steps involved in the FP-Tree generation with a $s_{min} = 3$.

A similar divide-and-conquer approach is employed by Han et al [29] in the FP-growth algorithm that decomposes the search space based on $length-1$ suffixes. Additionally, they reduce database scans during the search using a compressed representation of the transaction database, via a data structure called an FP-tree. The FP-tree is a specialization of a prefix-tree, storing an item at each node, along with the support count of the itemset denoted by the path from the root to that node, fp-tree combines horizontal and vertical database representation. All transactions containing a given item can easily be found by the links between the nodes corresponding to this item.

The generation of the FP-Tree is explained in figure 2.6. The first step consists in calculating the Frequency of each item, and then according to $s_{min} = 3$, all the infrequent items are removed from the transactions. The second step consists in sorting the items in transactions in descendent order with respect to their frequency. Then,

the transactions are sorted lexicographically in ascending order.

Each database transaction is mapped onto a path in the tree. The FP-Tree is a prefix tree with links between the branches that link nodes with the same item and a header table for the resulting item lists. All the single items could be read directly from the FP-Tree. Frequent single item sets can be read directly from the FP-tree.



(a) Transactions sorted lexicographically

(b) FP-Tree

Figure 2.8: FP-Tree Structure.

Figure 2.8 shows an FP-tree constructed for our example database. Dashed lines show item-specific inter-node pointers in the tree. Since the ordering of items within a transaction will affect the size of the FP-tree, a heuristic attempt to control the tree size is to insert items into the tree in non-increasing frequency order, ignoring infrequent items. Once the FP-tree has been generated, no further passes over the transaction set are necessary. The frequent itemsets can be mined directly from the FP-tree by exploring the tree from the bottom-up, in a depth-first manner.

The most remarkable advantage of FP-Growth is that it is often the fastest algorithm or among the fastest algorithms, compared against candidate generation algorithms. Due to the use of complex data structure it is more difficult to implement than other approaches, and an FP-tree could need more memory than a list or array of transactions [12]. In recent years, several applications and extensions using FP-Growth algorithm

have been proposed [15, 34, 52].

2.5 Other Algorithms

There are several algorithms proposed as extensions of the algorithms explained in previous sections. In the Apriori-based algorithms, there are those based on hashing, distributed strategies, and those that deal with dynamic datasets [32, 33, 41].

In [33, 41] a distributed strategy to approach Frequent Itemset mining using Apriori Algorithm is proposed. The Count Distribution Algorithm [33] has a set of nodes (computer used as a server), where each node holds a subset of items and performs the candidate generation in a local manner, which are related with the original dataset. These algorithms use Map Reduce, using the Hadoop tool. The main idea is to make partitions of the original database among a set of nodes of a cluster. Each cluster processes independently using an Apriori algorithm. Then the results are collected to obtain a global result. In [41] is proposed an algorithm for Fast Frequent Itemsets Mining using Nodesets that is similar to [33]. This algorithm divides the dataset among a set of nodes using a Map-Reduce Strategy, the main contribution is the fact that the algorithm is aimed at aproaching dynamic datasets.

There are several Hash implementations of Apriori; for example, in [32] it is proposed an algorithm where the dataset is stored in a hash table to accelerate the candidate generation task. Also, there are implementations based on the FP-Growth algorithm; for example, those based on a fully parallel implementation. For example in [58], a hierarchical partition is shown, based on a structure called Frequent Pattern List. The main objective is to avoid re-scanning the entire dataset to check the s_{min}. The Frequent Pattern List has many desirable characteristics, the most important is the capacity of creating independent partitions of the dataset, offering the possibility of determining the size of those partitions according to memory requirements. Each partition then is processed to obtain Frequent Itemsets using the FP-Growth algorithm.

Incrementally Building Frequent Closed Itemset Lattice algorithm [35] proposes a rep-

resentation of the search space in the form of a lattice as it was done in the Eclat algorithm. The main difference between ECLAT and this algorithm is the fact that each node in the lattice contains a Closed Frequent Itemset instead of one Frequent Itemset. When a new transaction is added to the dataset, the lattice is modified avoiding to reload the entire dataset.

Also, concurrent and parallels Eclat implementations were proposed, under the idea that each sublattice could be explored in an independent way. For example, in [71], the MREclat algorithm is shown as an implementation of the Eclat algorithm using a Map-Reduce architecture.

2.6 Summary

In this chapter, a frequent itemset introduction has been presented. The theory about Frequent Itemset Mining has been exposed with the intention of creating a context for this thesis research. The most important algorithms for Frequent Itemset Mining like Apriori, Eclat and FP-Growth, were presented. There is not a better algorithm for Frequent Itemset Mining, but a prefix based strategy like FP-Growth is considered one of the fastest algorithms. FP-Growth has better performance than Apriori and Eclat, but its high memory requirement prevents it from being used on large datasets. Single-threaded performance comparisons show that the FPGrowth is faster than Apriori and Eclat. However, in certain situations, such as when the frequency threshold is high, Apriori will outperform FP-Growth [50].

3

State of the Art

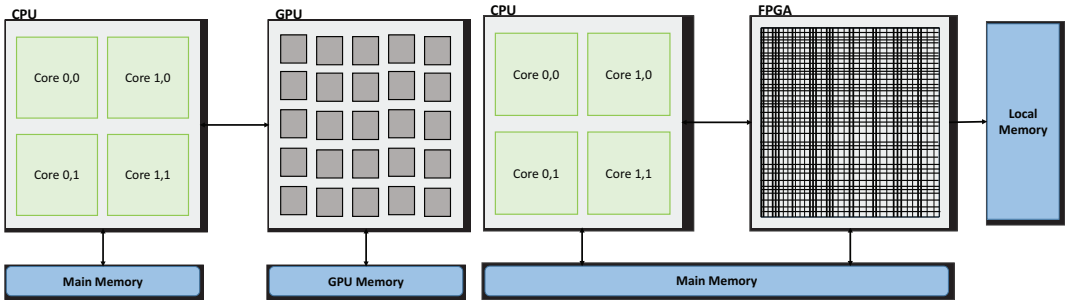
In this chapter, a review of the state of the art of Hardware Architectures for Frequent Itemset Mining is presented. Nowadays, there is a growing interest in designing Hardware Architectures to implement demanding and computationally expensive algorithms to gain speed up compared to their software counterpart. For this reason, a section of this chapter is dedicated to explaining the advantages of hardware acceleration and the challenges involved in this task. Therefore, the next section presents a classification of Hardware Architectures for Frequent Itemset according to the device employed (FPGA or GPU) and also on the used search space exploration approach (Depth First Search or Width First Search). In the next sections, a review Hardware Architectures of Apriori, FP-Growth and Eclat is presented. This classification also includes a revision of advanced hardware techniques like Addressable Memories and Systolic Arrays used for Frequent Itemset Mining.

3.1 Advantages of Hardware Acceleration for Frequent Itemset Mining

In recent years, there is a growing interest in accelerating software algorithms using hardware architectures. The use of existing accelerators, such FPGAs and GPUs, has demonstrated the ability to speed up a wide range of applications. Examples include image processing [16, 28, 62], data mining [6, 51, 54] and bioinformatics[5, 31] for FPGAs, and linear algebra [57], database operations [14], clustering [36], and simulations [9, 44] on GPUs. NVIDIA's Compute Unified Device Architecture, or CUDA, and AMD's Compute Abstraction Layer, or CAL, are new development environments for programming GPUs without the need to map traditional OpenGL and DirectX APIs to general purpose operations. On the other hand, FPGA applications are mostly programmed using hardware description languages such as VHDL and Verilog. Recently there has been a growing trend to use high-level languages such as Vivado HLS, SystemC and Handel-C, which aim to raise FPGA programming from gate-level to a high-level, using a modified C syntax with the inherent limitations of the Hardware Description Languages [18].

According to the application, heterogeneous architectures have been proposed, using an accelerator (FPGA or GPU) and a CPU. Figure 3.1 shows a GPU with 30 highly multi-threaded SIMD accelerator cores in combination with a standard multicore CPU. The GPU has a vastly superior bandwidth and computational performance and is optimized for running Single Instruction Multiple Data programs with little or no synchronization. GPU is designed for high-performance graphics, where the throughput of data is necessary [18]. Finally, figure 3.1 right shows an FPGA consisting of an array of logic blocks in combination with a standard multi-core CPU. FPGAs can also incorporate traditional CPU cores on-chip, making it a heterogeneous chip by itself. FPGAs can be viewed as user defined application specific integrated circuits (ASICs) that are reconfigurable. They offer fully deterministic performance and are designed for high throughput.

FPGAs have a set of hardware resources that can be programmed during the development stage, but they allow the hardware functionality to be configured and reconfig-



(a) CPU and GPU combination (b) CPU and FPGA combination

Figure 3.1: Schematic of heterogeneous architectures. In left, CPU in combination with a GPU is a heterogeneous system. In right, a CPU in combination with an FPGA is also a heterogeneous system.

ured during the execution stage. Such advantage gives a chance to implement different applications on the same hardware [18].

On the other hand, the hardware acceleration of algorithms has been recently explored. In a traditional algorithm implementation, the computer software is written for serial computation. In consequence, all the algorithms are implemented as a set of serial instructions. There have been several attempts to implement Data Mining algorithms in hardware to accelerate them. In [19], one of the first attempts to accelerate Data Mining is proposed using a Hardware Architecture. The chosen algorithms were: K-Means, Fuzzy K-Means, and Decision Tree Classification. A study of the most demanding tasks in the chosen algorithms is performed and, the proposed architecture implements kernels (one for each algorithm) that will be used as co-processors to accelerate the computation.

In [21], another hardware implementation of the K-means algorithm is proposed. The acceleration is used for clustering multi-spectrum images. The main task is to obtain the most important features from the image without dealing with the whole image data.

In [46], a parallel implementation of Space Saving Algorithm for Frequent Items Mining is developed. The proposed implementation uses a Single Instruction Multiple Data architecture. The used methodology is divided into two steps. The first step consists

in a pre-processing stage to verify that there are no dependencies in the data, the second step consists in executing Space Saving algorithm. In the experimental report, they verify that a parallel implementation of the proposed strategy gets a better result than its serial counterpart.

In all the previous works [19, 21, 46], they have achieved a remarkable acceleration compared to their software counterpart. Although, certain limitations could be observed like the knowledge of specialized development tools like OpenGL (for GPU) or Hardware Description Languages (for FPGA). Recently high-level development tools have been released to the market, for example, CUDA (for GPU) and High-Level Synthesis (for FPGA), that allow acceleration in the development of applications and they do not require a high level of specialization on GPUs or FPGAs.

3.2 Experimental Platform

The first alternative to implementing an algorithm is using a General Purpose Processor, most of the implementations of Frequent Itemset Algorithms have been done using a General Purpose Processor [2, 3, 11, 12, 29, 68]. All these previous implementations have been reporting acceptable results, but some applications require the result as soon as possible.

Another alternative to implementing an algorithm is using an Application Specific Integrated Circuit. An ASIC is an integrated circuit customized for a particular use. The implementation of an algorithm in an ASIC implies that the design will be optimal in time constraints and area. That is a real advantage, but once that the ASIC has been implemented, the design can not be modified. If there is a modification in the ASIC, all the entire design must be modified, and all the previous ASIC implementation become useless.

Another platform is the Graphic Processor Units [42], and the modern GPU is not only a powerful graphics engine but also a highly parallel programmable processor featuring peak arithmetic and memory bandwidth that substantially outpaces its CPU

counterpart. The GPU's rapid increase in both programmability and capability has spawned a research community that has successfully mapped a broad range of computationally demanding, complex problems to the GPU. This effort in general-purpose computing on the GPU, also known as GPU computing, has positioned the GPU as a compelling alternative to traditional microprocessors in high-performance computer systems [18]. If the memory access increase and parallelism are limited, the use of GPUs are not recommended, in Frequent Itemsets Mining according to the algorithm (Apriori or Eclat) are necessary much memory accesses.

An intermediate solution between the ASICs and the General Purpose Processors is the Reconfigurable Hardware. Field Programmable Gate Arrays (FPGAs) are the most used platform of Reconfigurable Computing; these devices are composed of hundreds or thousands of configurable logic devices modules. The advantages of using FPGAs are that they can get a better performance in time compared against a General Purpose Processor, and they offer more flexibility than ASICs [18].

In this dissertation, FPGAs are used as an experimental platform because they are excellent prototyping platforms, and they offer the flexibility to probe different architectural designs with the same hardware.

The development board used to implement the previous architectures is the Zedboard. The Zedboard contains a Zynq 7000 System on Chip. The Zynq 7000 is divided into the Processing System and Programmable Logic Area. All Zynq devices have the same basic architecture, and all of them contain, as the basis of the processing system, a dual-core ARM Cortex-A9 processor that is a "hard" processor. The Zynq processing system contains not just the ARM processor, but a set of associated processing resources forming an Application Processing Unit (APU), and further peripheral interfaces, cache memory, memory interfaces and clock generation circuitry. The Processing System contains a set of external interfaces, SPI, I2C, CAN, UART, GPIO, and USB. The second principal part of the Zynq 7000 architecture is the Programmable Logic PL that is based on the Artix®-7 FPGA fabric. The PL is predominantly composed of general purpose FPGA logic fabric, which consists of slices and Configurable Logic Blocks (CLBs), and there are also Input/Output Blocks (IOBs) for interfacing.

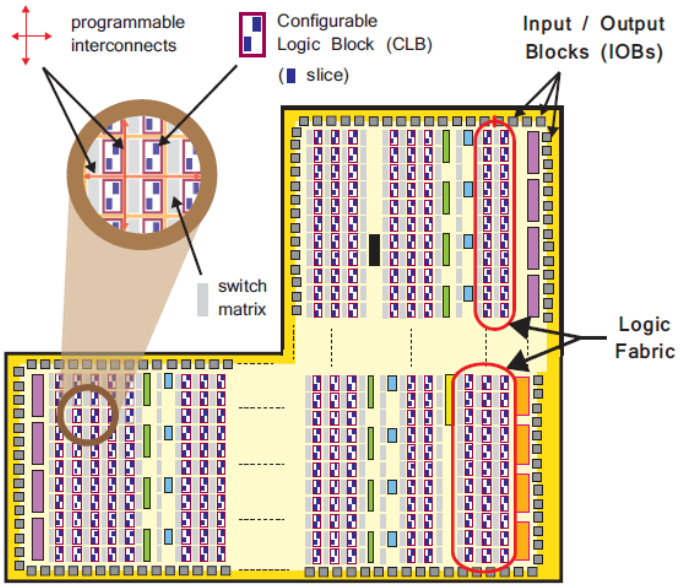


Figure 3.2: FPGA logic and its components.

– **Configurable Logic Block (CLB)**: CLBs are small, regular groupings of logic elements that are laid out in a two-dimensional array on the PL, and connected to other similar resources via programmable interconnects. Each CLB is positioned next to a switch matrix and contains two logic slices.

– **Slice**: A sub-unit within the CLB, which contains resources for implementing combinatorial and sequential logic circuits. Zynq slices are composed of 4 Lookup Tables, 8 Flip-Flops, and other logic.

– **Lookup Table (LUT)**: A flexible resource capable of implementing a logic function of up to six inputs; a small Read Only Memory (ROM); a small Random Access Memory (RAM); or a shift register. LUTs can be combined to form larger logic functions, memories, or shift registers, as required.

– **Flip-flop (FF)**: A sequential circuit element implementing a 1-bit register, with reset functionality. One of the FFs can optionally be used to implement a latch.

– **Switch Matrix**: A switch matrix sits next to each CLB, and provides a flexible routing facility for making connections between elements within a CLB, and from

one CLB to other resources on the PL.

– **Carry logic**: Arithmetic circuits require intermediate signals to be propagated between adjacent slices, and this is achieved via carry logic. The carry logic comprises a chain of routes and multiplexers to link slices in a vertical column.

– **Input / Output Blocks (IOBs)**: IOBs are resources that provide interfacing between the PL logic resources, and the physical device 'pads' used to connect to external circuitry. Each IOB can handle a 1-bit input or output signal. IOBs are usually located around the perimeter of the device.

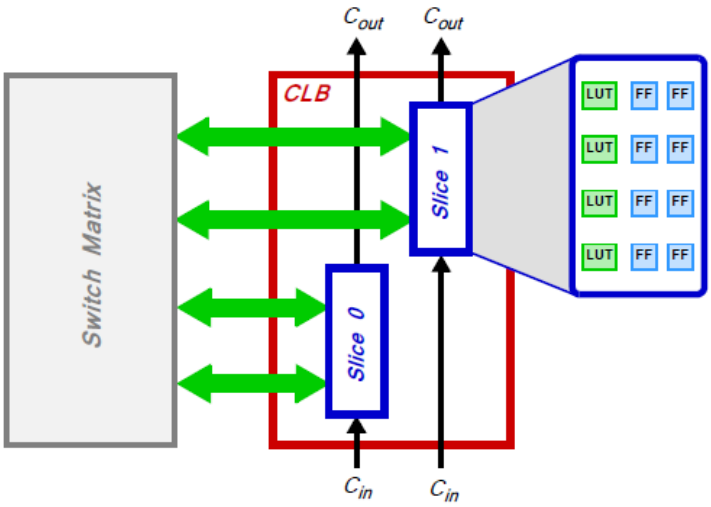


Figure 3.3: Elements of a Configurable Logic Block.

Table 3.1: Frequency of Items in set I.

Items	Frequency
Device Name	Z-7020
Part Number	XC7Z020
Programmable Logic Device	Artix-7 FPGA
Programmable Logic Cells	85K Logic Cells
Look-Up Tables	53,200
Flip Flops	106,400
Extensible Block RAM	560 KB (140)

In table 3.1, a summary of the Zynq 7020 device is shown.

3.3 Hardware Definition Languages and High Level Synthesis Languages

HDL (Hardware Description Languages) are used to describe the behavior of one hardware architecture, the design, and the electronic devices employed. These languages are useful to describe an electronic appliance, and they make easy to analyze and simulate them. The HDL languages might be used in different abstraction levels. The highest abstraction level is algorithmic also know like behavioral level, in this level the architecture is described using the algorithmic behavior instead describing the physical components and the interconnections between them. In the structural level, the architecture is described as a collection of logic gates and hardware components connected to perform the desired task. HDLs allow simulation using tools like ModelSim, Active-HDL or Xilinx ISim. These tools are used to test and debug the architecture without implement it in a physical device. The most common HDLs are VHDL, ABEL, Verilog, AHDL, Handel-C, and System-C. The most significant advantage of HDLs is that they use a synthesizer. A synthesizer is a software tool that transforms the HDL script into a hardware circuit that performs the desired function [18].

On the other hand, High-Level Synthesis Languages is the current trend in hardware description languages, this is a consequence because there is a research interest in accelerating software algorithms using hardware devices. All the software algorithms use a high-level abstraction, and there was not a direct form to translate this high-level code into hardware structures, and this guard against the development of theses architectures because it is needed an expert in High-Level Software Languages and an expert in Hardware Development. The main purpose of HLS are as follows:

- Algorithmic-based approaches are getting popular due to accelerated design time and time to market.
- The Industry trend is moving towards hardware acceleration to enhance performance and productivity.
    - CPU-intensive tasks can be offloaded to a hardware accelerator in FPGA.

    * Hardware accelerators require a lot of time to understand and design.

– Vivado HLS tool converts algorithmic description written in C-based design flow into hardware description (RTL).

    * Elevates the abstraction level from RTL to algorithms.

– High-level synthesis is essential for maintaining design productivity for large designs.

For these reasons, HLS has been developed with the intention of accelerating the production stage.

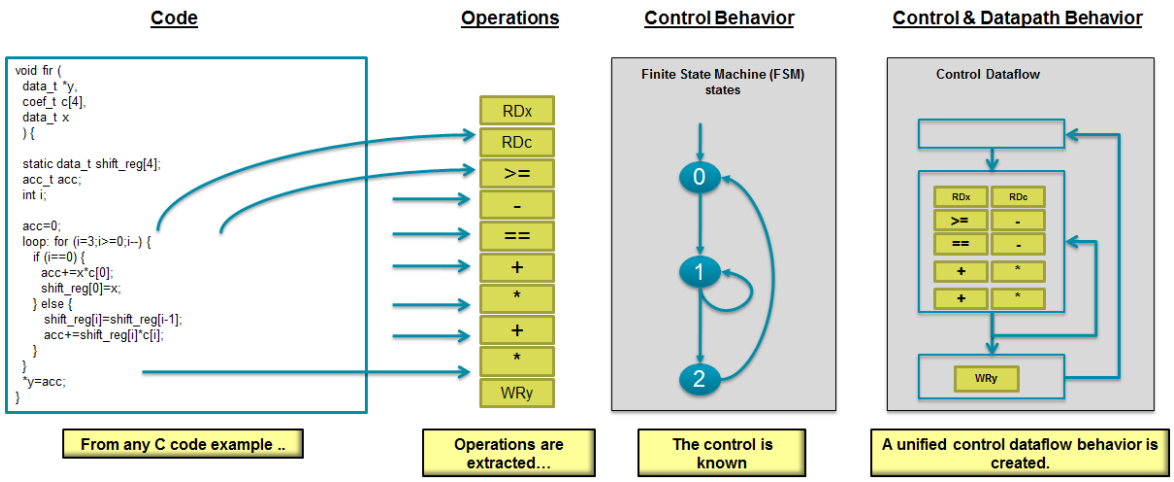


Figure 3.4: High Level Synthesis transformations from C Language to RTL level.

In figure 3.4, an example of how HLS perform the transformation from a C program into an RTL design is shown. The first step consists of identifying the operations. Once that the operations has been identified, the control state machine is generated. In these tasks, all the conditional and loop statements are mapped into a Finite State Machine. The last step consists of creating a unified control-data-flow behavior using the operations and the Finite State Machine generated in the previous steps.

3.4 Classification of Hardware Architectures

In literature, there have been proposed several software implementations of Frequent Itemset Algorithms and extensions [1, 3, 29, 33, 35, 68] but in recent years hardware architectures have been explored to offer a solution to the acceleration problem. The principal technologies employed in Hardware Architectures for Frequent Itemset Mining are GPU and FPGA. In figure 3.5 a classification of the literature respect to the hardware platform and the used algorithm is shown.

In the related work, there are implementations of the Apriori algorithm using GPUs like a viable alternative to accelerate Frequent Itemset Mining due to the parallel characteristics of GPUs. In [23], two implementations of Apriori have been developed. The first one performs all the processing in the GPU avoiding losing time in the communication between the GPU and the CPU. In the second, the GPU is used as a co-processor to perform the support counting. In both designs, the transactions are represented as binary vectors.

In [6, 7, 56, 60] presented FPGA implementations of Apriori. In the FPGA architectures, authors have explored diverse approaches to implementing custom architectures, using systolic arrays and content addressable memories. In [39, 54, 55], FPGA architectures based on the FP-Growth algorithm are presented. The most significant challenge is the data representation in Fp-Tree; for this reason, they propose the implementation of this structure in hardware. Finally, in [51, 70] hardware architectures of Eclat algorithm are shown. All the hardware implementations achieve a speed up compared with their software counterpart, but all of them share the same disadvantage, the resources of the used platform limits the number of different items that could be processed.

In figure 3.6, a classification on the used partition scheme is shown. According to this review, three partition schemes are used in the literature: Data Segmentation, Map Reduce, and Equivalence Classes.

In [33, 41, 59, 71], a partition of the search space using MapReduce is proposed, these methods allow parallelization of the algorithms using distributed environments and

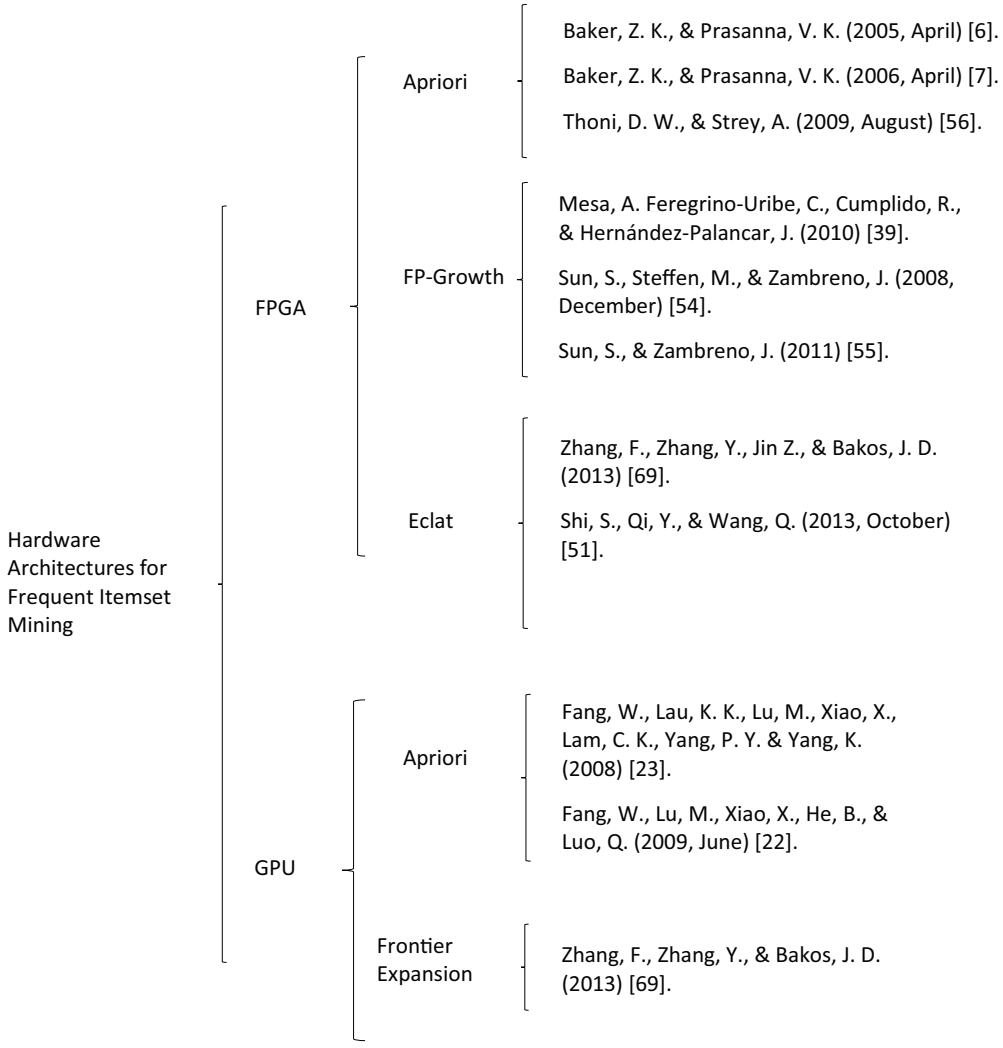


Figure 3.5: Classification of hardware architectures based on the device and algorithm employed.

distributed data store. Finally, in [30, 50, 51, 70, 70], a segmentation of the search space is done using Equivalence Classes. The idea is to divide the search space under the premise that equivalence classes could be defined as all itemsets that share the same prefix, so, the search space is divided into sub-lattices that could be processed

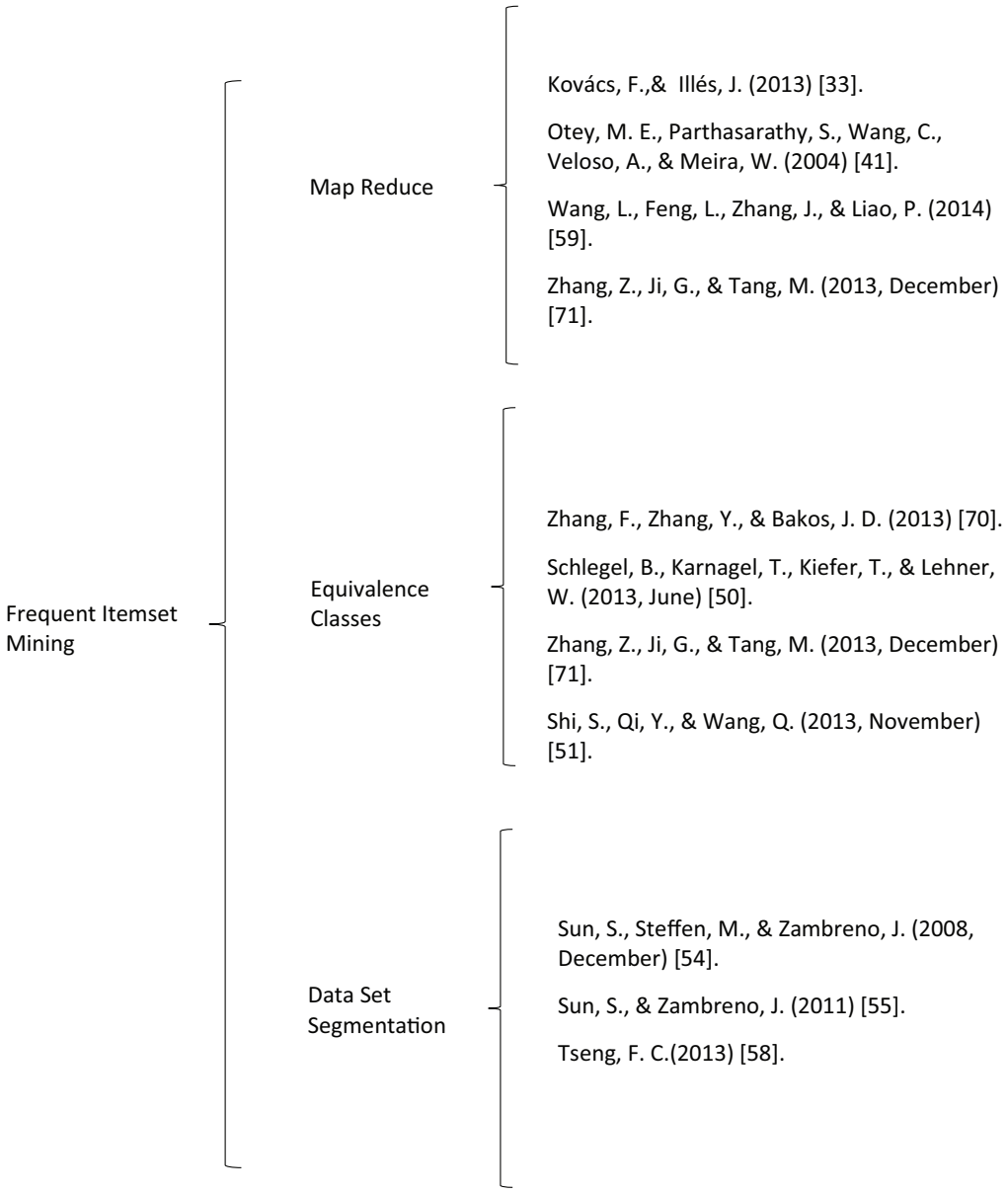


Figure 3.6: Classification of Algorithms that use a Segmentation Strategy.

independently.

3.5 Apriori Based Implementations of Frequent Itemset Mining

There have been several attempts to improve the Frequent Itemset Mining algorithms, being Apriori the most popular and widely spread algorithm. In the next lines, some of the most important Apriori implementations are presented. The main task is to implement the Apriori algorithm in an efficient manner using the minimum hardware resources and the minimum execution time.

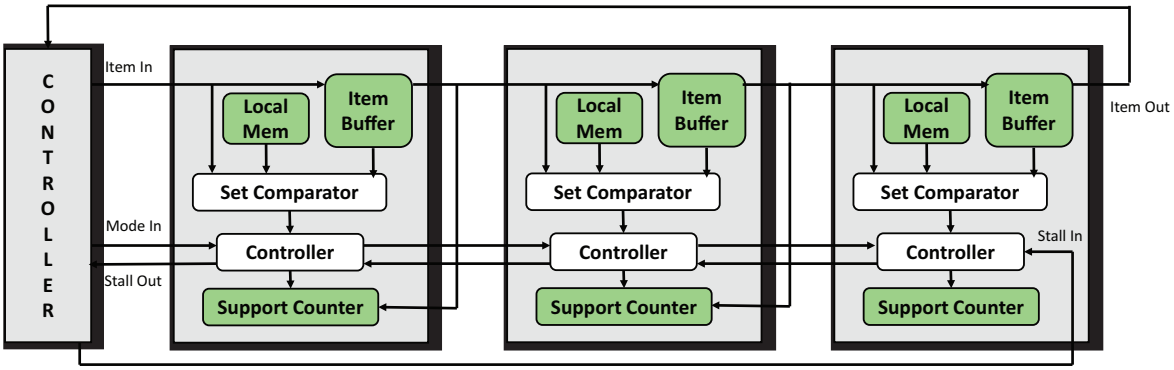
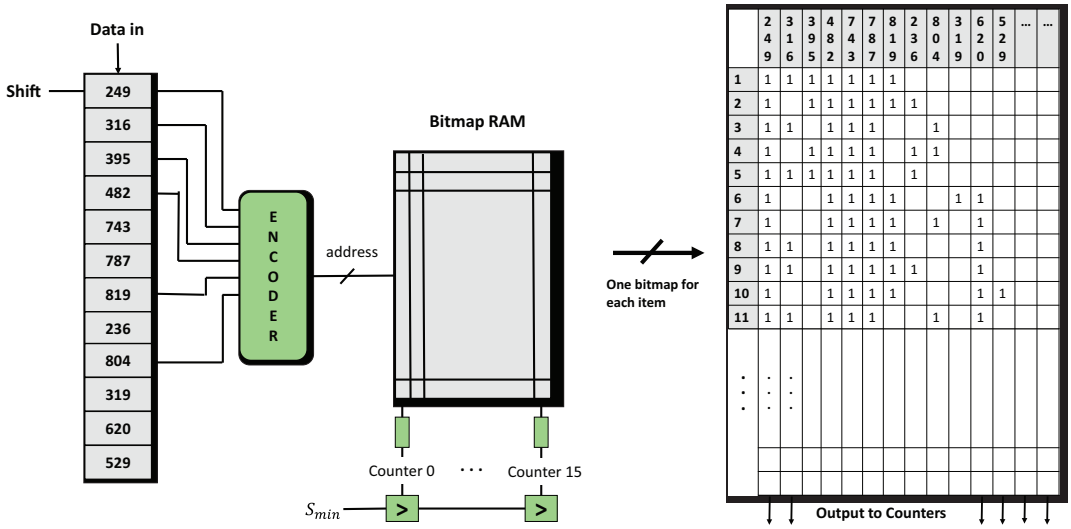


Figure 3.7: Systolic array employed in Apriori hardware implementation.

In [6], the proposed architecture is a systolic array implementation (Figure 3.7). The entire architecture is formed of 560 processor elements. Each processor element uses 70 slices so; the complete architecture requires 44000 slices. All the processor elements are connected in a linear array. A memory to store the candidates, an index counter, and a comparator (which allows the output of the candidate memory to be compared with the next item) compound a processor element. The data flow is one direction, and the stall data flow is the opposite direction. The support calculation is performed using two loops with no dependencies; this characteristic allows high parallelization. The first step consists of loading the processor units with candidates. Candidates enter at one end of the linear array. After the first candidate is stored in the first processor element, the i_{th} candidate is sent along the i_{th} processor element until all the processor elements in the array are full. All the transactions are sent through the array; one element is sent one per clock cycle. As each item arrives at a processor element, it is compared with the current item in the processor element. If the items match, the candidate pointer is



(a) Bitmapped CAM architecture

(b) Architectural desing for the bitmap operation

Figure 3.8: Apriori Bitmapped CAM architecture.

incremented, if the item in the processor element is lexicographically greater than the incoming item, the counter is not incremented. In the candidate generation stage, the candidates are injected into the systolic array, and this allows to each candidate set to be compared against all the candidate sets. The first step of this stage is to inject the new candidates into the array; each new candidate is written into the memory of each processor element. Next, all the candidates are injected through the linear array, to be compared to the candidates stored in the processor elements. The last stage consists of pruning candidates. Once again, all the candidates are injected into the linear array; the task is to determine the a priori existence of subsets of the new candidates within the current generation. If the candidate is not a superset of the itemsets of the current generation, the candidate is not a Frequent Itemset. Otherwise, the candidate is a Frequent Itemset. The reason to implement a systolic array is to decrease the number of the connections among processors elements and to ease the control.

Baker and Prasanna in [7], propose an improvement of the work presented in [6]. They pay special attention to improving the support counting due to this task consumes two orders of magnitude of time respect to the other tasks involved (candidate generation and candidate prune). The basic idea is to develop an architecture based on Bitmapped

Content Addressable Memories to store a set of candidates and then, perform the counting support operation. The Content Addressable Memories or CAM are a special kind of memories used in applications that require high-performance searches. The difference between a CAM and an RAM is that RAMs require an address. Thus, the memory returns the value stored in that address location; on the other hand, the CAMs require data and then, they search into the memory and verify if the data is located. This characteristic eases the counting support, because every time that a counting support operation is needed, it is only necessary to count the number of times that the CAM finds the required data into the memory. The architecture is a heterogeneous one integrated by a CPU and an FPGA. The CPU runs a software program that can perform candidate generation and candidate prune. The FPGA performs the counting support task. All transactions are streamed through the architecture shown in Figure 3.8. Each transaction is input to the CAM. If the incoming item matches any of the items in the CAM, the CAM produces an address that corresponds to the item that is stored in the bitmap. If a bit of the RAM output is set, the corresponding candidate counter increments. This operation is performed until all the transactions have been processed. Finally, if itemset counter is greater than s_{min}, the Itemset is a frequent one. The results reported shown a speedup of 24x compared to the Apriori software implementations of Borgelt [11] and Goethals [27]. In conclusion, multiple iterations of the architecture are needed to verify all the complete dataset. Then, it is important to maximize the number of candidates that could be processed in parallel but, the resources of FPGA are limited, and the performance of the architecture is constrained by the available resources.

In [60], a hardware architecture called HAsh-based, and PiPelIned HAPPI based on the Direct Hashing and Pruning DHP algorithm is presented. DHP algorithm is a variation of Apriori algorithm. Both algorithms generate candidate **k+1-itemsets** from large **k-itemsets**, and large **k+1-itemsets** are found by counting the occurrences of candidate **k+1-itemsets** in the database. The main advantage of DHP algorithm is that it uses a hashing technique to filter out unnecessary itemsets for the generation of the next set of candidate itemsets. The architecture is based on the previous works of Baker and Prasanna[6, 7], because a systolic array is used, and also a filter to performs the pruning caller Trimming Filter and a Hash-table.

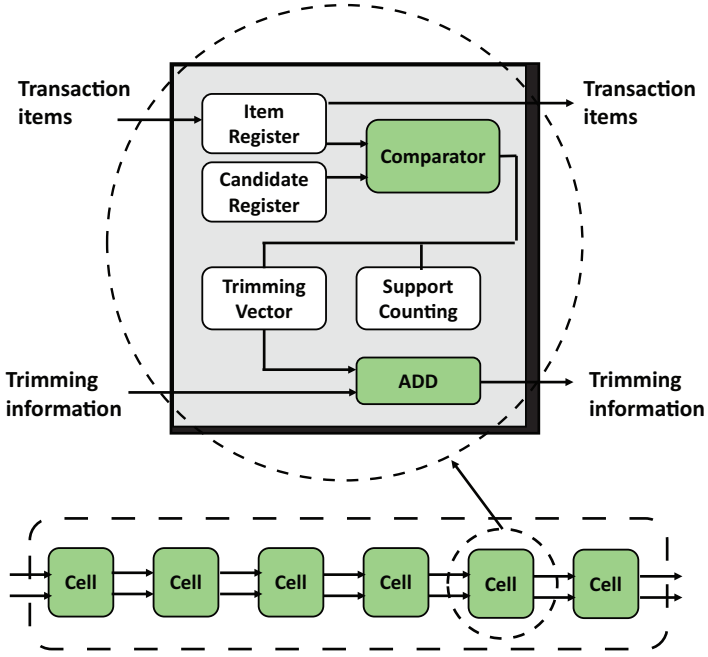


Figure 3.9: Systolic array of HAPPI architecture.

The architecture is composed of three modules: Systolic Array (shown in Figure 3.9), Trimming Filter, and Hash Table Filter. In the first step, all the dataset is injected in the architecture, and all the transactions are compared against the transactions in the systolic array to obtain the **k-itemsets**. In the pruning filter, it is determined when an itemset is a frequent one. Then, the Hash-table is constructed with all the possible itemsets of each transaction. The Hash-table is used to prune candidates because previously all the subsets of each transaction have been stored. The trimming filter is fed with the output of the systolic array; then the trimming filter returns the pruned transactions. Meanwhile, the Hash Table Filter receives as inputs all the **k-itemsets**, then the Hash Table for the next level is created. In the experimental part, the author reports an acceleration around 47-122x compared against [6].

In [22, 23], an architecture based on GPU is reported. In [22] an architecture called GPU miner is reported to perform K-means and Apriori algorithms. Specifically for the Apriori algorithm a hybrid architecture is proposed using the GPU as a co-processor to perform the counting support operation because, it is the most time demanding operation, and it can generate a bottleneck. Two series of experiments were performed; the first one consisted of comparing software implementations using a frequency of 1%.

The results show that the heterogeneous architecture gets better performance with speed up of 7.5x and 10.4x; the second one consisted of performing experiments over the T40.I10.D100K dataset varying the frequency value from 0.5% to 0.2%, this case resulted in speed up of 6.2x to 12.1x.

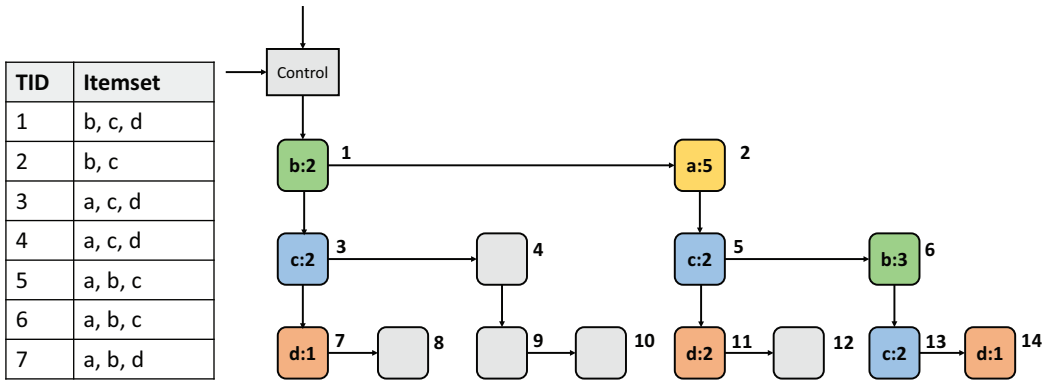
In [23] two Apriori-based architectures are reported. The first one is a full GPU implementation called Pure Bitmap Based Implementation (PBI). PBI avoids the data transfer between GPU memory and CPU; it also takes advantage of a binary representation to perform the intersection of Itemsets; furthermore together with a lookup table, the bitmap representation also accelerates the support counting, which is a time demanding component in Apriori. The second architecture is a heterogeneous one; it is called Trie-Based implementation (TBI), and it uses a tree structure to store itemsets, this architectures respects the flow of Apriori algorithm. The trie is incrementally constructed level by level, growing the trie of depth k, it generates all k+1 itemsets. The performance evaluation consists in comparing the GPU architectures against their GPU counterpart. In the TBI case, the results show that the GPU implementation gets better results for large support values, and if the value is near to zero the GPU implementations gets better results than CPU. The only algorithm reported that gets better results than the GPU implementations is FP-Growth.

In [56] an architecture based on systolic arrays to execute Apriori algorithm is proposed. It is the first one to use Block RAMs to store the results instead of external memories as the previous works reported because capacity in FPGA has grown in recent years. Authors proposed to implement a systolic array architecture where the data path is connected in a sequential way to data and a control bus. The main disadvantage is that this design was only synthesized to approach a problem that fits the FPGA. They implement 70 processors elements that can process 1120 candidates in parallel.

3.6 FP-Growth Based Implementations of Frequent Itemset Mining

In the literature, implementations of hardware architectures using the FP-Growth algorithm could be found in [39, 54, 55]. These hardware architectures emulate the FP-Tree structure used in FP-Growth.

In [54] a systolic tree structure is proposed to implement a hardware version of FP-Growth algorithm. A systolic tree is a set of processor elements with the same characteristics and connected to form a tree structure as shown in figure 3.10. A systolic tree is an array of pipelined processing elements in a multi-dimensional tree pattern. The purpose of this structure is to mimic the memory structure used by the FP-Growth algorithm. The structure of the systolic tree has a depth of W and K children for each node. Each node in the original FP-tree corresponds to a processor element, resulting in the total number of processing elements in a tree as $K^W + K^{W-1} + ... + K + 1 = \frac{K^{W+1}-1}{K-1}$. The architecture has three operation modes: write, scan, and count. The systolic tree is built in write mode; the items are streamed from the root node in the direction set by the write mode algorithm. The root node is the processor control element, and it acts as the controller to the rest of the tree. The support count of a candidate itemset is extracted in both scan and count mode.



(a) Example Dataset

(b) Systolic Tree Architecture

Figure 3.10: FP-Growth Architecture using a Systolic Tree.

In consequence, two methods of projection of the dataset are proposed: parallel projection and partition projection. The partition projection uses less memory and I/O operations but requires that each projection will be mined in a sequential manner. Meanwhile, in the parallel projection the execution time is reduced by the parallel nature of this projection but the memory consumption increases alarmingly.

In [39] another hardware architecture to execute FP-Growth algorithm is proposed. Therefore, in this research, a binary representation is proposed, using a vertical bits vector. Authors continue with the idea of implementing Frequent Itemset Mining algorithms using a binary representation and binary logical operations to perform support counting tasks. They proposed an Equivalence Class segmentation based on Eclat algorithm but once the segmentation is done, FP-growth algorithm takes the control to generate frequent itemsets. In this architectural design there is no candidate generation, instead the full search space is explored until reaching a node for which the support is zero.

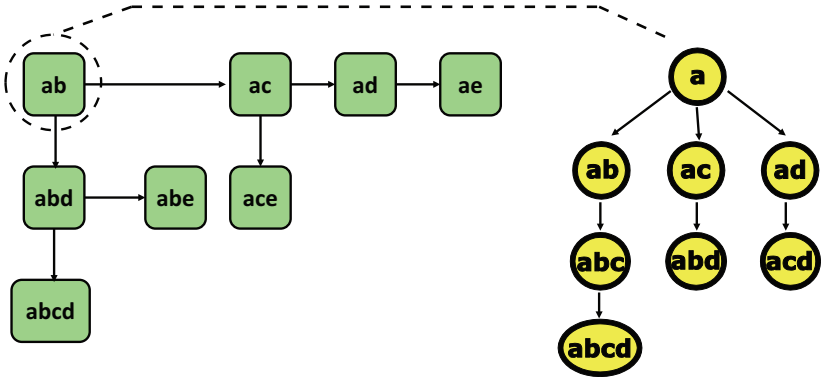


Figure 3.11: Tree Structure used to store an Equivalence Class.

The architecture has three operation modes: Data In, Support Threshold, and Data Flush. In the Data In mode, the search space is loaded by levels and all sections of the database are injected. When the entire dataset is injected, the architecture enters in the Support Threshold mode and the s_{min} value is provided and propagated through the systolic tree. Once all nodes receive the s_{min} value, all the processor elements change to Data Flush mode and the results are extracted from the architecture. The most demanding tasks in the architecture are the propagation of the values in the systolic array and the extraction of the results from each processor element in the

Data Flush Mode. The results reported show that the proposed architecture gets better performance than the hardware implementation reported in [54], specifically in the Chess dataset, the architecture gets a speedup of one order of magnitude. The results also give evidence that the architecture fits better for big, and dense datasets.

In [55], an extension of work [54], the authors still use the systolic tree structure. The main contribution of this research is an algorithm to be used in the datasets projection generation when the dataset does not fit well in the hardware architecture due to memory limitations (mainly memory resources). The main advantage of the proposed dataset projections is that they segment the search space with the intention of reusing the hardware resources. Two methods for dataset projection are introduced: parallel projection and partition projection. The partition projection requires less memory, and I/O operations but, it requires that the dataset must be mined in a sequential manner, this limits the parallelization of the algorithm. The parallel projection gets better performance in execution time but, the disadvantage is the excessive memory consumption.

The performance evaluation was performed against the FP-Growth software implementation. For the Chess, Accident and Retail datasets the architecture achieves a better performance using both projections methods. In consequence, the reported results indicate that the execution time of FP-Growth is directly related to the FP-Tree structure, and it is not related to the partition method employed.

3.7 Eclat Based Implementations of Frequent Itemset Mining

The Eclat algorithm is based on depth-first search and the exploration of Equivalence Classes. The first step consists of calculating all the Frequent Items, and in a recursive way the search space is traversed. Recently there has been an interest in developing hardware architectures for Frequent Itemset Mining using the Eclat algorithm [51, 70]. In Bakos et al. [70], a hardware architecture using Eclat algorithm is shown in Figure 3.12.

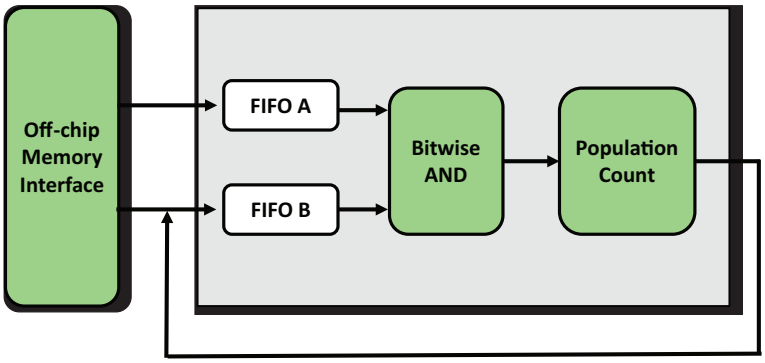
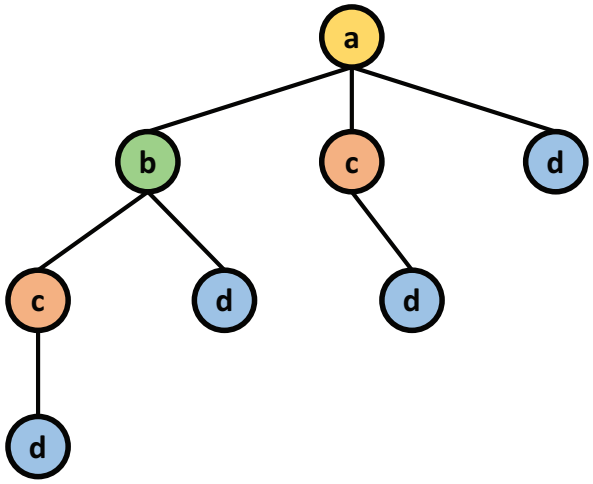


Figure 3.12: Hardware architecture of Eclat algorithm.

They also use a binary representation to save space in memory and perform only logical operations. They propose a design using two stacks called FIFOA and FIFOB that take charge of storing Frequent Itemsets. The first itemset is read from the FIFOA, and another from FIFOB and an AND operation is performed to generate the intersection of the two sets. Then the count support operation is implemented using the population count module. If the support is higher than s_min the frequent itemset is stored in an external memory or else the data is erased. A control state machine handles the FIFO and the memory during the recursive search in the search space, and it is implemented using a stack and a Finite State Machine.

The behavior of the architecture is depicted in figure 3.13. In the example shown in figure 3.13, a is loaded into FIFOB and b is loaded into FIFOA, the result of the intersection of a and b is stored in FIFOB (itemset ab is stored in FIFOB). Then if the itemset ab is frequent and valid itemset, ab is kept in FIFOB. In step 2, ab is stored into FIFOA and FIFOA must store an item lexicographically greater than item b, for this reason, FIFOA holds item the c. Then, the intersection between itemset ab and item c is performed; the vector binary resultant abc is stored in FIFOB. The itemset abc is frequent, and it is a valid itemset, and it is kept in FIFOB. A special case is when a result itemset is not a frequent one. In step 3 this scenario is depicted, the itemset acb is stored in FIFOB, FIFOA holds the item d, the result of the binary intersection is $abcd$, but $abcd$ is not a frequent itemset. For this reason, the $abcd$ itemset is flushed from the FIFOB, and the algorithm continues the execution until no more frequent itemsets could be generated.



(a) Traversing of search space

Step	FIFO B	FIFO A	Result (in FIFO B)	Valid	Output	Flush
1	a	b	a & b	Yes	Yes	No
2	a & b	c	a & b & c	Yes	Yes	No
3	a & b & c	d	a & b & c & d	No	No	Yes
4	a & b	d	a & b & d	No	No	Yes
5	a	c	a & c	Yes	Yes	No
6	a & c	d	a & c & d	Yes	Yes	Yes
7	a	d	a & d	Yes	Yes	Yes

(b) FIFO control sequence

Figure 3.13: Behaviour of Eclat hardware architecture.

Memory constraints are the natural limitations of Frequent Itemset Mining Architectures. Memory off chip is used to approach this limitation because the FPGA employed device has 2MB of internal memory. For this reason, a coding strategy for sparse datasets is used, the coding method chosen is the differential coding (it only consists of XOR operations). Besides, a cache memory system (scratchpad memory) is used to store the coded elements. A rule is proposed, for those transactions that have at least one zero are deleted. The next step consists in applying the differential code, and if the compressed ratio is higher than a threshold, this transaction is stored in the cache, in another case the transaction is stored in the off-chip memory without codifying. The performance is compared against a software implementation of Eclat algorithm.

In [51], authors propose the acceleration of the intersection operation in the Eclat algorithm. The objective is to compute $L_1 \cap L_2 \cap, ..., \cap L_n$ as soon as possible. This hardware architecture has the next characteristics: the parallel load of items into the memory modules, binary vectors are sent to a comparator module with the intention of finding the common values, and return the common data to the memory modules. In the same way, a full comparator matrix structure is provided to perform the parallel intersection computation. The experimental results show that the proposed architecture achieved a speedup of 26.7x on the intersection operation compared against software implementations of Eclat.

3.8 Comparison of Related Work

In the previous sections, the related work in literature has been reviewed. In table 3.2 and table 3.3, a summarized revision is presented. The main characteristics reported in both tables are device employed (GPU or FPGA), technology, the datasets employed in the evaluation of results, the area used in each design, operation frequency, speed up and the algorithms used for evaluation purposes. The information about operation frequency and area are available only for FPGA architectures. Most of the reviewed works are constrained by the resources of the device employed; only [39, 54] have proposed a segmentation or projection of datasets. For example in [6] the reported slices are 44000 that represents a 99.7 % of usage of the employed device [64]. In [7] the resource consumption of the employed device was of 100 %. In [54] the resource consumption is about 99 % of the employed device [66]. In most of the reported architectures, the resources consumption is around 90 % to 100 % because they try to process the maximum possible number of transactions and itemsets. In consequence, expensive architectures in terms of area are obtained, and in some cases they do not guarantee that all the frequent itemsets will be mined [6, 7].

The second aspect to consider is the speed up achieved. In the Apriori-based architectures [6, 7, 22, 23, 56] the maximum speed up is 30x. Eclat based architectures [51, 70] got a maximum speed up of 68x. Apparently, the algorithm that gets better results

is FP-Growth, but the datasets employed contain few transactions and items. For example, chess dataset contains 75 items and 3196 transactions. FP-Growth based architectures implement a tree structure in FPGA and this become unpractical for big datasets. So FP-Growth based architectures are good enough for small and dense datasets. On the other hand, Apriori-based Architectures also are limited by the number of transactions and itemsets that could be processed by the respective architecture. Eclat based architectures do not get a performance like FP-Growth architectures, but Eclat architectures can deal with big datasets.

One inconvenient when the previous works are compared is the non-existence of a framework to evaluate Frequent Itemset Mining architectures, because each work use different datasets, and each work is compared with different related works. Also, comparisons against software implementations are sometimes unfair, and it makes difficult to do a consensus of the related work.

Table 3.2: Comparison of related work, Part1.

Work	Algorithm	Device	Technology	Datasets	Area	Operation Frequency	Speed up	Comparison Against
[6]	Apriori	FPGA	Virtex II Pro XC2VP100 with -6 speed grade [64]	T40I10D100K, T10I4D100K	44000 slices	112 Mhz	4x to 30x	[11, 27]
[7]	Apriori	FPGA	Virtex II XC2V6000 with -4 speed grade [65]	T40I10D100K, T10I4D100K	33792 slices	120 MHz	24x	[6],[11, 27]
[60]	DHP	FPGA	Altera Stratix 1S40 [4]	T5I12D100K, T10I4D100K, T15I6D100K, T20I8D100K	-	58.6 MHz	47x to 122x	[6]
[22]	Apriori	GPU	GPU NVIDIA GTX280	T10.I4D10K, T10I4D100K, T40.I10D100K	-	-	7.5x to 10.4 x	[11], [27]
[54]	FP-Growth	FPGA	Virtex-4 XC4VFX140 with -10 speed grade [63]	Chess, Accidents and Retail	22272 slices	360 Mhz	12x to 2230×	[29]
[23]	Apriori	GPU	GPU NVIDIA GTX280	T40I10D100K, Chess and Retail	-	-	4x to 16x	[11], [27]
[56]	Apriori	FPGA	Virtex 5 XC5VLX110T with -3 speed grade [66]	T10.I4.D100K	Slices: 17226 LUTs: 67680/69221	155 MHz	4x	[6, 7]

Table 3.3: Comparison of related work, Part2.

Work	Algorithm	Device	Technology	Datasets	Area	Operation Frequency	Speed up	Comparison Against
[39]	FP-Growth	FPGA	Virtex-4 XC4VFX140 with -10 speed grade [63]	Chess, Accidents and Retail	PE: 349 Flip Flops: 31061 LUTs:124593	137 MHz	100x	[54]
[69]	Frontier Expansion	GPU	GPU Tesla S1070 GPU server with four Tesla T10 Processors	T40I10D100K, T40I10D500K, T40I10D1000K, T40I10D1500K, T10I5D3000K, T40I10D3000K	-	-	6x to 30x	[10, 11]
[70]	Eclat	FPGA	GiDEL PROC-Star III PCIe add-in card with four FPGA Stratix III 260 connected to three independent memory block [26]	T40I10D03N500K, T40I10D03N1000K, T60I20D05N500K, T90I20D05N500K	Registers: 44145/203520 LUT:38552/203520 Block Memory Bits: 5939472/15040512 DSP Block: 24/768	200 MHz	4x to 68x	[11]
[51]	Eclat	FPGA	Virtex LX240T with -1 speed grade, DDR3 SDRAM memory with a bandwidth of 6.4GB/s [67]	T40I10D100K	Registers: 3102 LUT:12602 BRAM:86	274.217MHz	26.7x	[11]

3.9 Summary

In this section, a revision of the state of the art was presented. First, all the advantages of using Hardware acceleration were exposed with the intention of remarking the importance of this dissertation. Then, a detailed taxonomy of the research trends in the last years has been presented, including those implementations based on GPUs and FPGAs. Independently of the used hardware platform, all the implementations of Frequent Itemset mining have achieved acceptable results. In the literature, the most recurrent implemented algorithms are Apriori, FP-Growth, and Eclat. All these implementations have their limitations, but all of them have in common that they are constrained by the available resources of the employed devices. The intention of this dissertation is to approach this common drawback of the previous works.

4

Architectural Design and Hardware Implementation

In this chapter, a detailed explanation of the proposed architectures is presented. In the first section, the proposed strategy to go over the search space is exposed, this strategy is mainly based on equivalence classes. The most remarkable advantage of the proposed strategy is that the equivalence classes could be processed independently, in consequence the Frequent Itemset Mining can be parallelized. It is necessary to perform a full hardware implementation of the proposed strategy with the intention of taking advantage of the inherent parallelism of FPGA. Accordingly, two hardware architectures are presented. The first one consists in a full hardware implementation of the proposed search strategy where each equivalence class is processed in a sequential manner using 32 bits words. The second one is a parallel version of the proposed strategy that has the advantage of processing 100 words or 32 bits in parallel. Finally with the intention of speeding up the proposed architectures, a parallel model using two processor elements to divide the workload is described.

4.1 Proposed Search Strategy

The Equivalence Class Transformation algorithm Eclat was proposed by Zaki[68]. The main idea is to represent the search space in a lattice as the one shown in Figure 2.2. With this representation, the search space could be divided into equivalence classes where the equivalence relationship is the prefix. Frequent itemsets with the same prefix share the same sublattice. In consequence, this representation offers the possibility of exploring the search space in depth-first search and breath-first search. Eclat traverses the itemsets in lexicographic order and uses a representation called transactions id list or tid-list. Each tid-list is a vertical projection of each item, and every row contains the id of one transaction.

	a	b	c	d	e
1	1	1	0	1	1
2	0	1	1	1	0
3	1	0	1	0	1
4	1	0	1	1	1
5	1	0	0	0	1
6	1	0	1	1	0
7	1	1	1	0	0
8	1	0	1	1	1
9	0	1	1	0	1
10	1	1	0	1	1

(a) Dataset using vertical binary vectors.

	a
1	1
2	0
3	1
4	1
5	1
6	1
7	1
8	1
9	0
10	1

$S_a = 8$

	b
1	1
2	1
3	0
4	0
5	0
6	0
7	1
8	0
9	1
10	1

$S_b = 5$

	ab
1	1
2	0
3	0
4	0
5	0
6	0
7	1
8	0
9	1
10	0

$S_{ab} = 3$

(b) Intersection and support counting operations in binary vectors.

Figure 4.1: Data Representation and operations used by our proposal.

Our proposal consists in a variant of the search strategies proposed in Eclat. The representation employed in our proposal is the vertical binary vector because the intersection and support counting operation can be implemented as a combinatorial system. The transactions are coded in 32 bits integers using the compressed array representation used in [30]. The word size is 32 bits because the memory employed to store the transactions uses 32 bits words. For example in figure 4.1, a binary vector dataset of five items is shown. For items a and b, their support values are calculated

counting the set bits in the correspondent vectors being $S_a = 8$ and $S_b = 5$. The intersection operation is performed using Boolean AND operations. For example to get the itemset ab, an AND operation between the binary vector of item a and b is performed. The result is the binary vector ab shown in figure 4.1, and $S_{ab} = 3$.

Our search strategy is a combination of breadth and depth first search. This strategy has the advantage that the search space can be partitioned, in consequence each partition of the search space can be processed in parallel. For example, figure 4.2 describes the behaviour of the proposed strategy. The first step consists in taking item a and generate all the 2-itemsets being ab, ac and ad frequent itemsets. The next step is to generate all the 3-itemsets. abc is generated intersecting ab and ac. abd is generated intersecting ab and ad. acd is generated intersecting ac and ad. The final step consists in generate all the 4 itemsets, $abcd$ is generated intersecting abc and abd. In this first stage, all the itemsets with prefix a or that belong to the equivalence class a are generated. This process is repeated for all the remaining items.

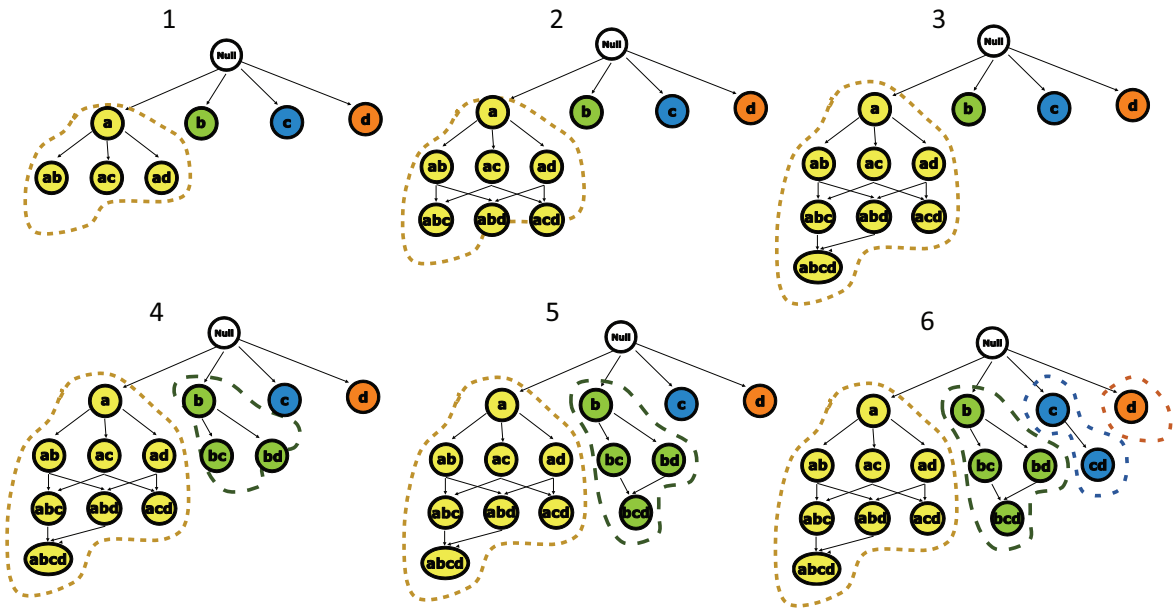


Figure 4.2: Search strategy proposed for four items.

Algorithms 4 and 5 describe the behaviour of the proposed search strategy. In algorithm 5 the partition of the search space into equivalences classes is performed. In

line four, the initial items are sent as a parameter to the function *search_strategy*. In line five, the processed item is removed from the list of initial items. For example, to process the equivalence class a the initial items are a, b, c, d and e. Once the equivalence class a is processed, a it is not necessary anymore and it is removed from the initial items. For equivalence class b, the initial items are b, c, d and e. Once that all the equivalence class a is processed, all the itemsets that belong to this class can be removed from memory because they are not necessary anymore. In consequence, this search strategy is useful in scenarios where there are memory constraints like in FPGA design because in memory only will be stored the itemsets that belong to the current class.

Algorithm 4 Proposal of search strategy for each equivalence class

Require: C_1 Initial Items
1: $C_i = C_1$;
2: $search\_strategy(C_i)$:
3: $C_{i+1} = \emptyset$;
4: **for all** $c_j \in C_i$ **do**
5: **for all** $c_k \in C_i$ with $k > j$ **do**
6: $F = c_j \cup c_k$; $T(F) = T(c_j) \cap T(c_k)$
7: **if** $support(F) \leq s_{min}$ **then**
8: $C_{i+1} = C_{i+1} \cup F$;
9: **end if**
10: **end for**
11: **end for**
12: **if** $C_{i+1}! = \emptyset$ **then**
13: $search\_strategy(C_{i+1})$;
14: **end if**
15: **return** $\bigcup C_i$; // All the frequent itemsets of this equivalence class

The search strategy described in algorithm 4 performs the itemset mining of each equivalence class. For example, let $C_1 = a, b, c, d, e$ be the initial items. The $2-itemsets$ for class a are, generated being $c_3 = \{ab, ac, ad, ae\}$. Once that all the 2-itemset have generated, they are sent as a parameter to the function *search_strategy*. For each itemset, the intersection with its next itemset in the list is performed if both itemsets share the same prefix. The result of the intersection of ab with ac, ad, ae can be performed because all of them share the prefix a and their last item is greater than b. For this example, all the itemsets are frequent, so $C_3 = \{abc, abd, abe\}$. The result

Algorithm 5 Search Strategy

Require: C_1 Frequent Items
Require: $C_i = C_1$;
1: $D = C_1$; // Set of initial items.
2: $result = \emptyset$;
3: **for all** $c_j \in C_i$ **do**
4: $result = result \cup search\_strategy(D)$;
5: $D = D - c_j$;
6: **end for**
7: **return** $result$; // All the frequent itemsets in the dataset

of the intersection of ac with ad, ae is stored in $C_3 = \{abc, abd, abe, acd, ace\}$. The result of the intersection of ad with ae is stored in $C_3 = \{abc, abd, abe, acd, ace, ade\}$. Once that all the 3-itemsets have been generated they are used as a parameter of $search\_strategy$ function to generate the $4-itemsets$. This process is repeated until no more frequent itemsets can be generated.

4.2 Architecture based on the proposed search strategy

Our first proposal is the implementation of a full hardware implementation of the proposed search strategy, The behaviour of this architecture is divided into two parts; the first one consists in the generation of the frequent items and the second one consists in the Frequent Itemset Mining using the proposed search strategy.

Figure 4.3 shows a high-level diagram of the proposed architecture. This architecture is composed of a general purpose processor, an UART module, an off-chip memory, a memory subsystem and the hardware accelerator. The general purpose processor and the UART module are necessary to create an interface to receive the datasets and send out the frequent itemsets. The off-chip memory is the DDR2 of the ZedBoard. The memory subsystem creates an interface between the off-chip memory and the hardware accelerator that contains a memory management unit and a memory arbiter.

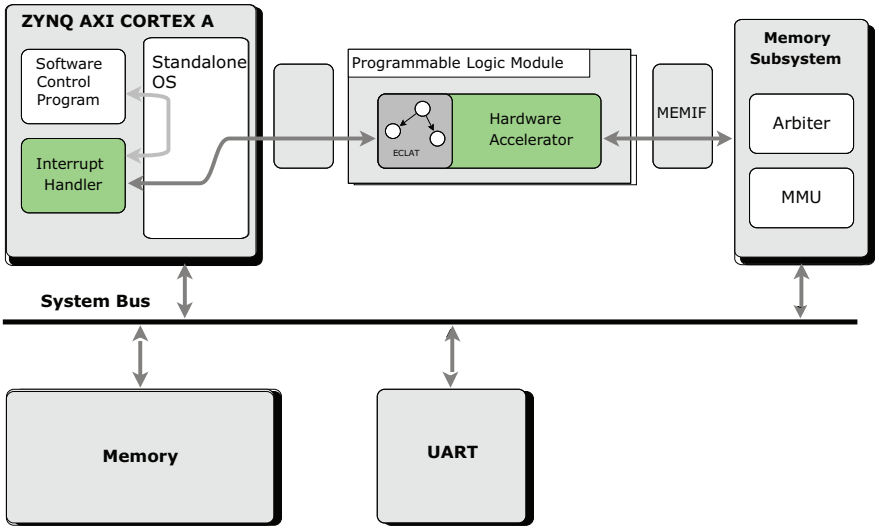


Figure 4.3: Hardware architecture that performs the proposed search strategy.

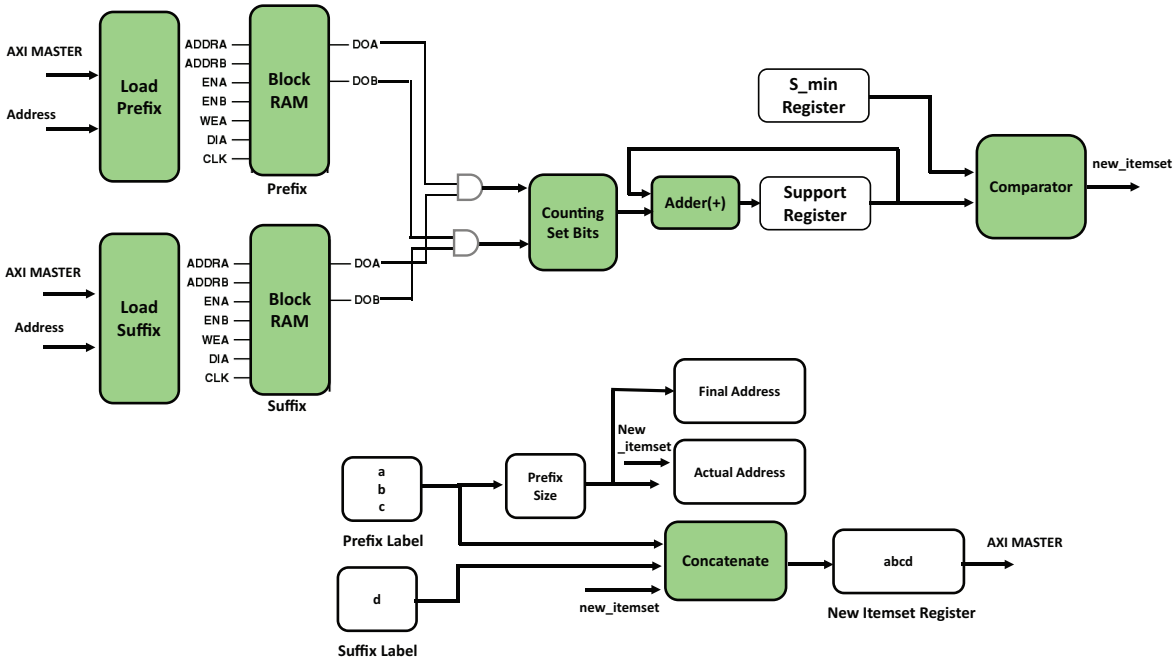
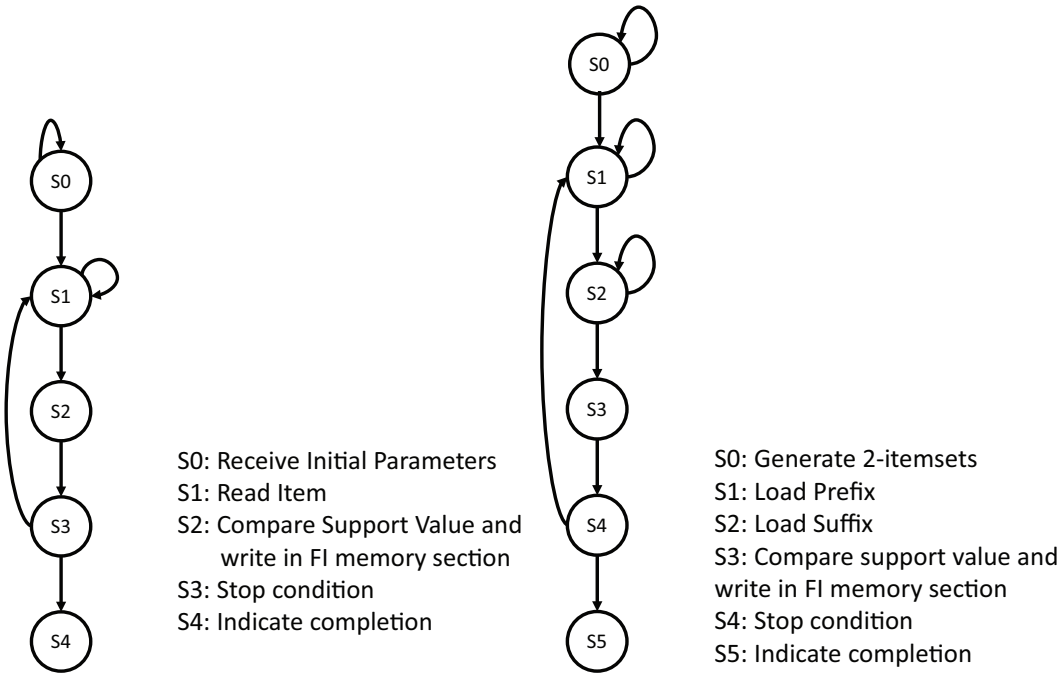


Figure 4.4: Low level design of the proposed architectural design.

Figure 4.4 describes a block diagram of the hardware accelerator. It consists of two dual block RAM memories called $prefix$ and $suffix$. The BRAMs have a storage capacity of 122 Kb, in consequence they can store one million of transactions but it is not limited only to one million of transactions because the **Load Sufix** and

Load Prefix modules can iterate to cover more than one million of transactions. The outputs of each memory are connected to AND gates that perform the intersection using 32-bits words. The counting support module receives as inputs two 32-bit words that are the result of the AND gate. The output of the counting support module is accumulated in the support register until all the transactions have been covered. And finally, a comparator compares the support register value with the S_{min} register value. If the current itemset is a frequent itemset, the prefix label is concatenated with the suffix label and then the concatenated label is stored in the off-chip memory with its corresponding binary vector.



(a) Items mining state machine. (b) Itemset mining state machine.

Figure 4.5: Finite state machines of the proposed search strategy.

Figure 4.5 shows two finite states machines that describe the behavior of the proposed architecture. The first state machine corresponds to the frequent items generation task. In state $S0$, the architecture receives the initial direction where the binary vectors are stored, the number of transactions, the number of items, the label of the actual item, the direction where the frequent item labels will be stored and the S_{min} value. In state $S1$, the architecture reads the binary vector of the current item and stores it in the load prefix BRAM, and then the counting set bits module computes the support

value. In state $S2$, if the item is frequent, its label is stored in the off-chip memory as a frequent itemset. State $S3$ verifies that all the items have been processed.

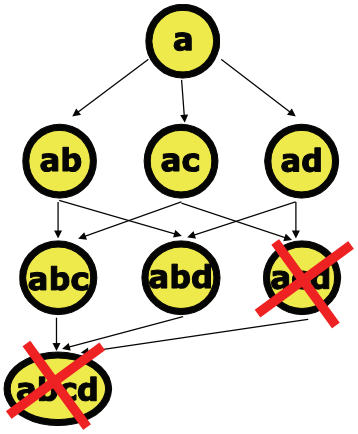


Figure 4.6: Search space for item a.

The second state machine describes the behaviour of the Frequent Itemset Mining stage. In state $S0$, the 2-itemsets are mined. Table 4.1 describes the operations involved in state $S0$. The first step consists in receiving a set of initial items, for this example the initial items are $D = a, b, c, d$ being all the $a - prefixed$ itemsets the equivalence class to process. The first item in the initial items list determines the equivalence class to process. The second step consists in performing the intersection and support counting of the 2-itemsets; Prefix and Suffix BRAMs are used in this task. The item a is stored in $prefix$ BRAM, and the next items will be stored in $suffix$ BRAM to perform the intersection and support counting operation. All the frequent 2-itemsets will be stored in the off-chip memory.

Table 4.1: 2-itemsets generation.

Frequent itemsets in memory	Prefix BRAM	Suffix BRAM	Frequent
{ }	a	b	Yes
{ab}	a	c	Yes
{ab, ac}	a	d	Yes
{ab, ac, ad}	a	e	Yes

Once that the 2-itemsets have been calculated, the next step corresponds to state $S1$ and consists in the k-itemsets mining. Table 4.2 describes the operations employed

in this stage using the search space of figure 4.6. The $2 - itemsets = \{ab, ac, ad\}$, so in state $S1$ the binary vector of ab is stored in Prefix BRAM, and in state $S2$, then itemset is stored in Suffix BRAM. In state $S3$, the intersection operation and the support counting operation indicate that abc is a frequent itemset and in consequence, abc is written in the off-chip memory. The itemset ab is not flushed from the Prefix BRAM because there is an itemset that shares the same prefix. So, ab and ad are intersected to generate a new itemset. In this case, ab is flushed from memory because the next itemset in memory is abc and it is a 3-itemset and they do not share the same prefix. The intersection of two itemsets can only be performed, if both of them have the same cardinality and share the same prefix. For example, the prefix of itemset abc is ab and the prefix of itemset acd is ac, although they have the same cardinality they do not share the same prefix, and they cannot be intercepted to generate a new itemset. In contrast for itemsets abc and abd, they share the prefix ab and the same cardinality, in consequence they can generate the itemset $abcd$.

Table 4.2: Operations performed by the architecture

Frequent itemsets in memory	Prefix BRAM	Suffix BRAM	Result in Suffix BRAM	Frequent	Output	Flush prefix BRAM
{ab, ac, ad}	ab	ac	abc	Yes	Yes	No
{ab, ac, ad, abc}	ab	ad	abd	Yes	Yes	Yes
{ab, ac, ad, abc, abd}	ac	ad	acd	No	Yes	Yes
{ab, ac, ad, abc, abd}	ad	-	-	No	No	Yes
{ab, ac, ad, abc, abd}	abc	abd	abcd	No	No	Yes

The previous steps are executed until no more itemsets can be generated, for the example in table 4.2 the $a - prefixed$ frequent itemsets are $\{ab, ac, ad, abc, abd\}$. The second finite state machine is executed for all the frequent items in a serial form processing independently each item.

4.3 Unrolled Architecture based on the proposed search strategy

In this section, an improvement over the previous architecture is shown. The previous architecture performs the intersection and support counting operations iteratively using 32-bits words because the BRAM memories perform I/O operations using two words of 32-bits. The I/O operations over the BRAMs represents the critical path in the previous architecture. For example, if the dataset has 3200 transactions,it is necessary 100 words of 32-bits, in consequence the previous architecture needs 50 clock cycles to store the prefix in the prefix BRAM and 50 clock cycles to store the suffix in the suffix BRAM and also 50 clock cycles are required for the intersection operation. For each intersection and support counting operation, the architecture requires 150 clock cycles. Accordingly, an unrolled architecture is proposed to improve the execution time. Figure 4.7 describes the unrolled hardware architecture proposed.

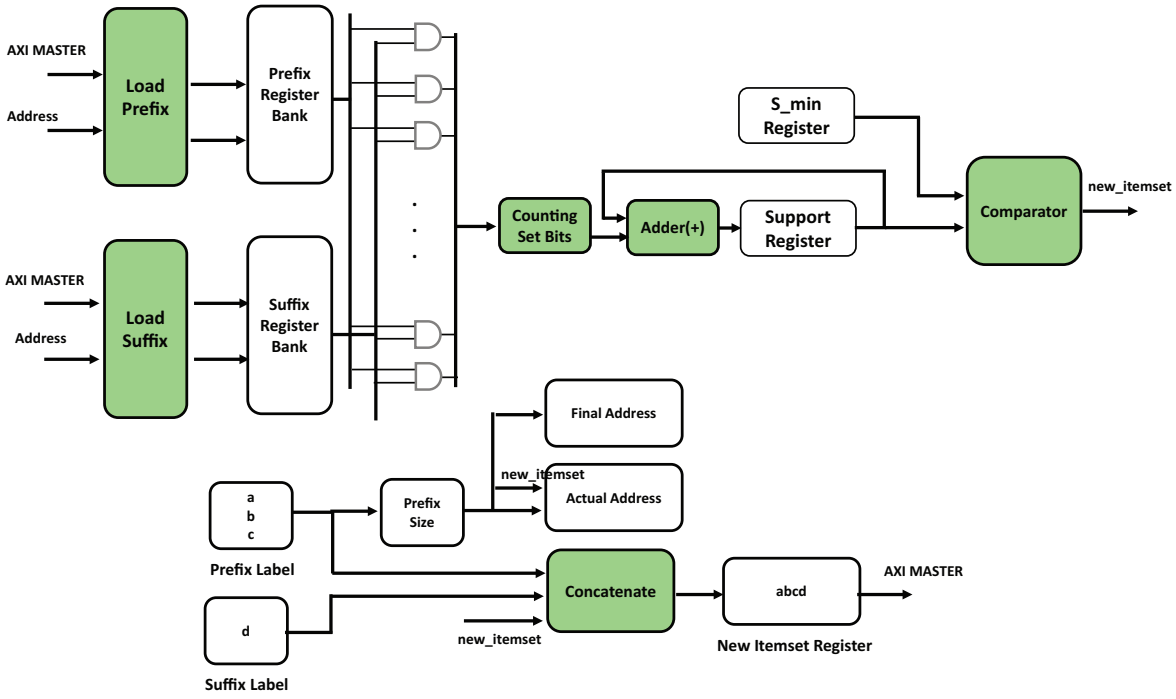


Figure 4.7: Hardware architecture that performs an unrolled implementation of the proposed search strategy.

The first change to improve the performance of the architecture is to replace the BRAMs memories with register banks. Although the register banks must be implemented using resources of the programmable logical area of the FPGA (slices and LUTs), they can perform parallel reads and writes on all their locations, in consequence, reads and writes of n words in parallel can be performed. For this architectural design register banks of 100 words of 32 bits have been implemented. The second change is to unroll the cycle that performs the intersection and support counting of each word of 32-bits stored in the BRAMs in the previous architecture. It is necesary 100 AND gates connected to the output of each register in the register banks like in figure 4.7, and the output of each AND gate is connected to the support counting module that is implemented using look-up tables to obtain the number of set bits in each word of 32-bits. Finally the sum of each word that contains the number of set bits is done, and the value obtained is compared with the value stored in the S_{min} register. The advantage of unrolling the architecture is that the number of clock cycles employed is reduced, The intersection and support counting operation is performed using three clock cycles. In consequence, the number of clock cycles employed in the intersection of two itemsets is reduced from 150 clock cycles (employed in the previous architecture) to 103 clock cycles. The gain reported is a consequence of the usage of more hardware resources flip-flops and multiplexers used in the register banks and their interconnections with the AND gates. In comparison, the unrolled architecture has better speed up but it consumes more hardware resources.

4.4 Dual Core Design and Partition Strategy

With the intention to get a speed up, a dual-core architecture is proposed for each of our proposals. Figure 4.8 shows a high-level representation. In the logical programmable area, two hardware accelerators are implemented with the intention of distributing the workload between the two of them.

Previously it has mentioned that the proposed search strategy has the advantage of splitting the search space into disjoint sets or classes. This can be used in a high level of parallelism because each core can process an equivalence class independently.

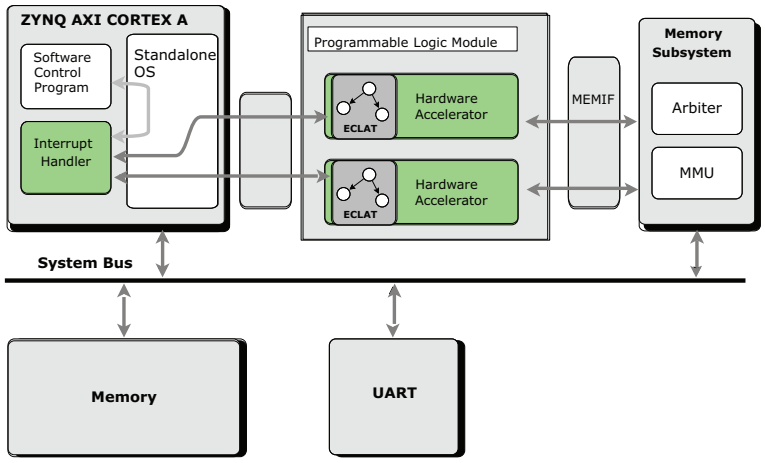


Figure 4.8: Dual core hardware architecture proposed

Figure 4.9 describes the partition of the search space for four items. The first processor element receives the set of items $D = \{a, b, c, d\}$ and it processes the equivalence class a. The result of the Frequent Itemset Mining is stored in its memory section, and the frequent itemsets are $a, ab, ac, ad, abc, abd, acd, abcd$. Meanwhile, the second processor element receives the sets of items $D = \{b, c, d\}$ and it process the equivalence classes b, c and, d. The resultant frequent itemsets are $\{b, bc, bd, bcd, c, cd, e\}$. The dual core architecture obtains a parallelism to process independent equivalence classes, and this impacts directly on the performance of the proposed search strategy. The number of cores can be incremented as much as the FPGA employed permits it.

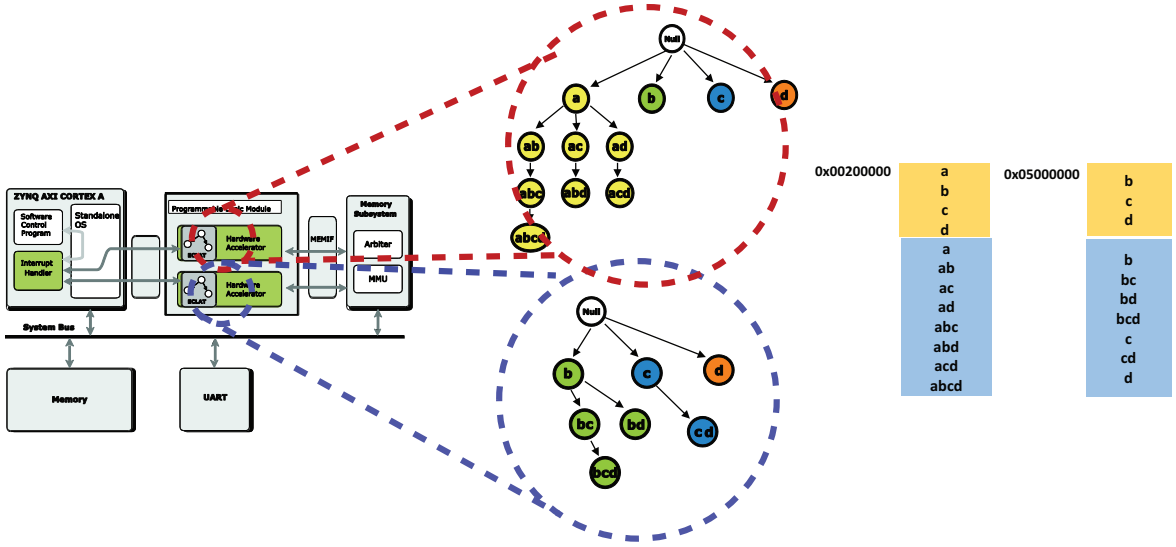


Figure 4.9: Partition of the search space using 2 processor elements.

4.5 Summary

In this chapter, the search strategy used for the implementation of two architectures was presented. The advantage of the search strategy is the ability to divide the searching space into sub-lattices that can be processed independently regardless the dataset size. The division into equivalence classes also has the save memory advantage compared with other strategies like FP-growth. The hardware architectures proposed perform a full hardware implementation of the proposed strategy, and the main issue is to accelerate the intersection and support counting operations because these tasks are the most used in the proposed search strategy.

5

Experimental Results and Performance Evaluation

In this chapter, all the details involved in the experimental test and the evaluation of the results are exposed. The first section includes a revision of the metrics used for the evaluation: area and runtime, and how these metrics can be used as indicators of the performance of one hardware design. Then, a review of the data sets and their characteristics and how to generate them is shown and how certain characteristics impact directly in the performance of the algorithms. Finally, an analysis of the results achieved by the two developed hardware architectures is performed making comparisons among these architectures and the most representative software implementations reported in literature.

5.1 Evaluation Metrics

The most common metrics used to evaluate Hardware Architectures are the Throughput and the Area. In combinatorial designs, the complexity in time is determined by the maximum frequency operation that is determined by the maximum delay in the combinatorial route. In sequential designs, the complexity in time is calculated by clock cycles required by the entire architecture.

The area indicates the hardware resources that a design uses. Unfortunately, there are not a standard metric to measure how many resources an FPGA design uses. After implementing an architectural design in an FPGA device, the synthesis, place, and route tools report about the hardware resources used from the FPGA. Some of the most important resources are number of slices, number of flip flops, number of 4 or 6 inputs look up tables LUTs, number of clocks, block RAM, multipliers, among others.

A Hardware Architecture might be reported regarding LUTs or number of slices. A fair comparison could be to compare all the resources available in the FPGA. A hardware design that uses the dedicated integrated blocks of an FPGA uses fewer resources from the programmable logic section, and this implies a minor use of hardware area compared against design that does not use the dedicated blocks, and these blocks are implemented in the programmable logic section. It is observed that the same HDL code synthesized for different FPGAs of the same family get different measurements of area. This gap increases when the same hypothetical design is implemented in FPGA of different vendors. In some cases, when there is a need to classify FPGA designs some characteristics might be ignored. According to the application, some designs report only timing constraints because they report the execution speed of the architecture. A fair area comparison could be performed only if FPGA of the same characteristics are employed.

5.2 Validation Datasets

In the literature diverse datasets have been used to test the functionality of the software algorithms and hardware architectures. In [3], an algorithm is proposed to generate synthetic datasets over a large range of characteristics. These synthetic transactions imitate the characteristics of transactions in the retailing environment. This model follows the premise that people in the real world have the intention to buy sets of items together. For example, people can buy beer, chips, soda, and pencils, but another person can buy chips and soda, another person could buy pencils and chips, and all these sub-items of the same transaction still could be frequent.

Table 5.1: Data sets used to validate the Hardware Architecture.

Dataset	Size(MB)	Binary Dataset Size (MB)	Average Length Transaction	Number of Transactions	Number of Items
Chess	0.330	0.013	37	3196	75
T40I10D100K	14.6	4.32	10	100k	1000
T40I3N500k	68	11.9	40	500k	299
T40I3N1000k	136	24.1	40	1000k	300
T60I5N500k	106	18.9	60	500k	500
T90I5N499k	160	22.901	90	499k	500

The characteristics employed to generate the datasets are: number of transactions $|D|$, average size of transactions $|T|$, average size of the maximal potentially large itemsets $|I|$, number of potentially large itemsets $|L|$ and, number of items $|N|$. All these characteristics are used to generate synthetic datasets. For this dissertation, three values for $|T|$: 40, 60 and 90 have been chosen. The values for $|I|$ are 3,5 and 10. Table 5.1 summarizes the dataset parameter settings, and also an estimated of the size in MB of the datasets. The proposed strategy is evaluated using a dense dataset (chess) and sparse datasets because it is necessary to evaluate the performance of the architecture with dataset of different characteristics.

5.3 Performance evaluation of the first proposed hardware architecture

The hardware architecture is evaluated using area and execution time metrics. The area evaluation is performed using the hardware report usage that provides Vivado HLS synthesizer. The execution of the architecture has been compared to the execution time of Apriori, Eclat and FP-Growth software implementations [11, 12] because these are optimized implementations of such algorithms. The previous implementations can be found on the personal website of Christian Borgelt [13]. The software implementations have been tested on a PC with an Intel i3-3217U processor at 1.8 GHz and 8 GB DDR2 RAM memory with Windows 7 ultimate. The execution time considers the input and output operations and the CPU time for all the algorithms and the hardware architectures. In the performance test, all the datasets described in previous sections are used, a dense dataset (chess) and sparse datasets with a different number of items and transactions have been employed to evaluate the hardware architecture and compare it with the software algorithms.

From figure 5.1 to 5.6, the hardware architecture and the dual core hardware architecture are faster than most of the software algorithms, although FP-growth gets a better execution time for the T40I10D100K dataset. Figure 5.1 shows the performance of the three software implementations and the hardware architectures for the chess dataset. The maximum speedup achieved by the architecture proposed is 112x and 210x for the dual core architecture, this comparison is unfair because Apriori gets bad results when it deals with dense datasets. For chess dataset, Fp-growth was the algorithm that obtained better results compared with the proposed architecture the maximum speedup obtained is 2.9x and 5.8x for the dual core architecture.

The better performance reported for the hardware architectures is when they have to deal with sparse datasets (Figures 5.2, 5.3, 5.5, 5.6). For algorithms like Fp-Growth, the tree structures employed consume much memory, in some cases the algorithm consumes all the available memory (Figures 5.3, 5.5, 5.6). The proposed architecture only needs to store the set of current itemsets $k-itemsets$ and the generated $k+1-itemsets$, and the number of itemsets stored in the memory is reduced because the search strategy

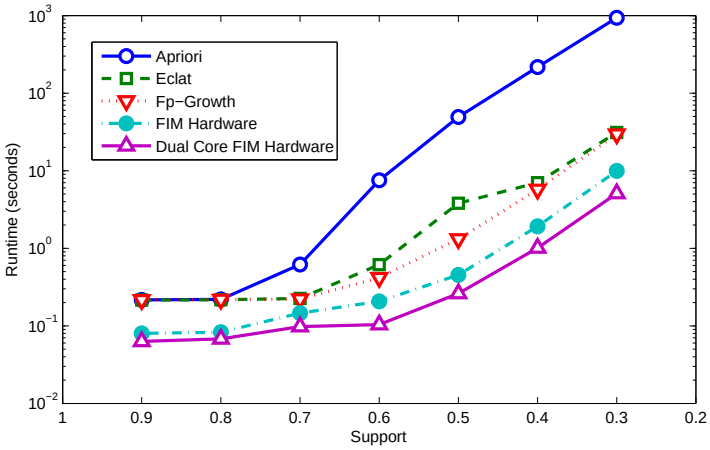


Figure 5.1: Execution time comparison (Chess).

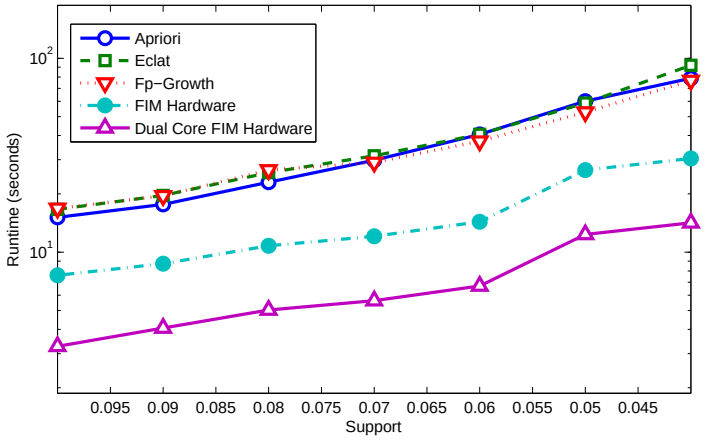


Figure 5.2: Execution time comparison (T40I3N500k).

does not generate all the $k-itemsets$, it only generates the $k-itemsets$ that belong to the current equivalence class. Figure 5.4 shows that Fp-growth has better performance than the proposed architecture because all the transactions in the dataset share the same items, this favours to Fp-Growth in the construction of the tree structures.

The experiments show that the proposed architecture has good performance for sparse and dense datasets obtaining a speedup of 2.5x for the proposed search architecture and 6x for the dual core architecture compare with the best software implementations (it depends on the dataset).

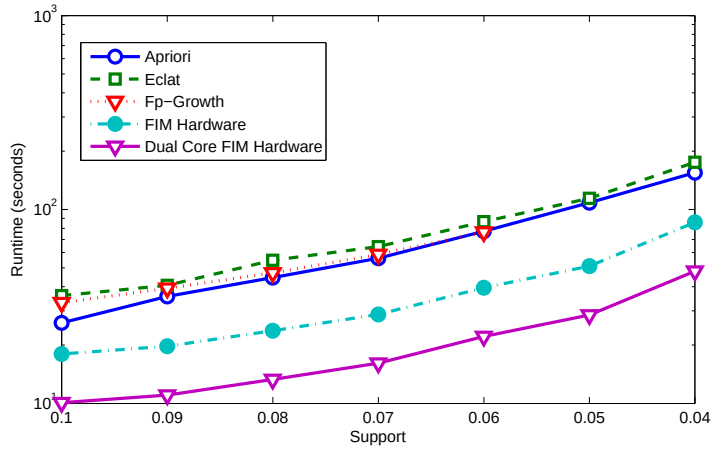


Figure 5.3: Execution time comparison (T40I3N1000k).

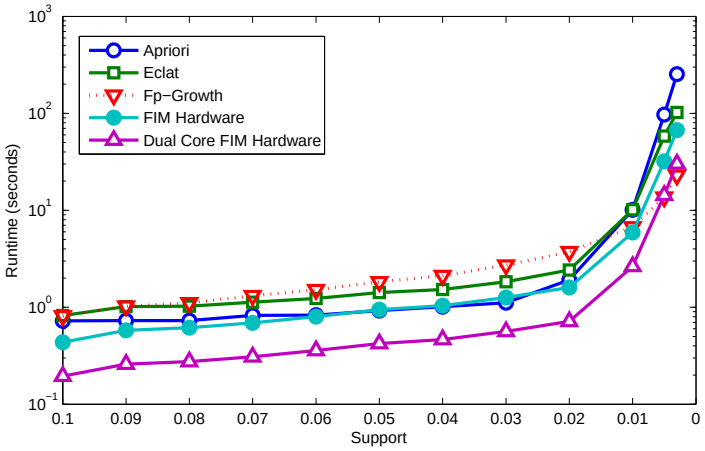


Figure 5.4: Execution time comparison (T40I10D100K).

Table 5.2 and 5.3 show the area reports for both architectures. The operation frequency reported for both is 114 Mhz. The elements reported are BRAMs, DSP48E, Flip Flops and LUTs. For the first architecture (Table 5.2), 46 % of the BRAMs are used because they are necessary to store the prefix and suffix, and they have to store one million of bits. The usage of Flip-Flops (3 %) and LUTs (9 %) is minimum because the architecture only needs a few registers to store S_{min} value and the control signals, this is the most compact design obtained.

For the second architecture (table 5.3), although there has been an increment in the

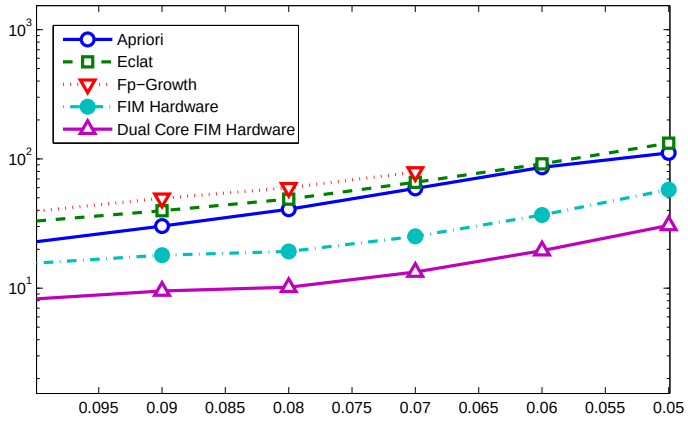


Figure 5.5: Execution time comparison (T60I5N500k).

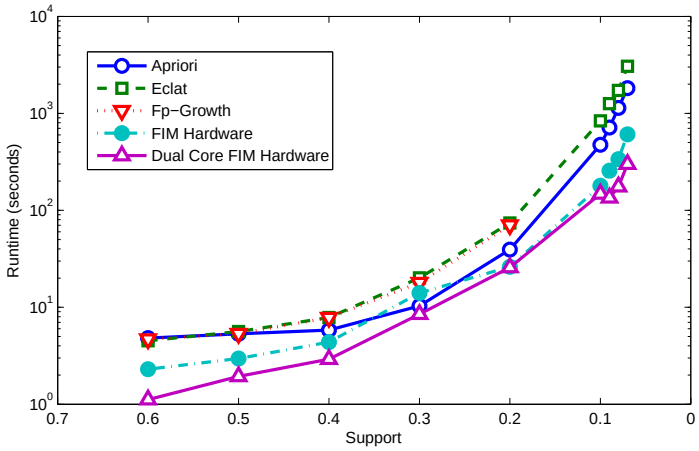


Figure 5.6: Execution time comparison (T90I5N499k).

resources employed, the architecture is still a compact one. The only resource that has increased considerately is the BRAMs because the number of stored bits is two million. Intuitively, an advantage of the compact design is that the number of cores that can be attached to the architecture can grow, and the workload can be divided among other processor elements to speed up the execution time.

Compactness is the main advantage of the proposed hardware architecture. Although, it is a compact design, both versions accelerate the Frequent Itemset Mining problem. The main reason is that the architectures sacrifice speed to obtain a compact design.

Table 5.2: Hardware resources used by proposed hardware architecture.

Name	BRAM	DSP48E	FF	LUT
Expression	-	-	0	2618
Memory	129	-	0	0
Multiplexer	-	-	-	1943
Registers	-	-	3475	-
Shift Memory	-	-	0	164
Total	129	20	3475	4725
Available	280	220	106400	53200
Utilization (%)	**46**	**9**	**3**	**9**

Table 5.3: Hardware Resources used by the dual core architecture.

Name	BRAM	DSP48E	FF	LUT
Expression	-	-	0	3806
Memory	258	-	0	0
Multiplexer	-	-	-	2891
Registers	-	-	5234	-
Shift Memory	-	-	0	279
Total	258	32	5234	6976
Available	280	220	106400	53200
Utilization (%)	**92**	**14**	**4**	**13**

One alternative is to create an array of compact processor elements to get a better speedup.

5.4 Performance evaluation of the unrolled hardware architecture

The evaluation of this architecture is performed using the same datasets described previously. The speed up obtained in this architecture is considerable because the intersection and support counting has been unrolled. In the previous architecture, these operations were implemented in an iterative way. The previous architecture is a compact one, but it requires more clock cycles to perform the operations and a low hardware usage. The unrolled architecture has a greater hardware usage, but

the execution time for each dataset is reduced considerably. Due to this increase in the hardware usage, the architecture does not fit in the FPGA device available but it was simulated using the operation frequency and the clock cycles reported by the synthesizer. From figure 5.7 to figure 5.12 the unrolled hardware architecture gets the best performance compared with Apriori, Eclat and Fp-growth because each intersection and support counting operation is performed using 100 words of 32-bits in parallel.

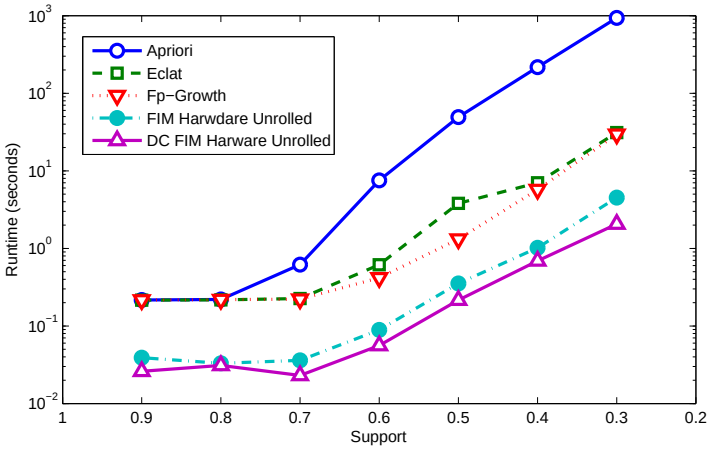


Figure 5.7: Execution time comparison (Chess).

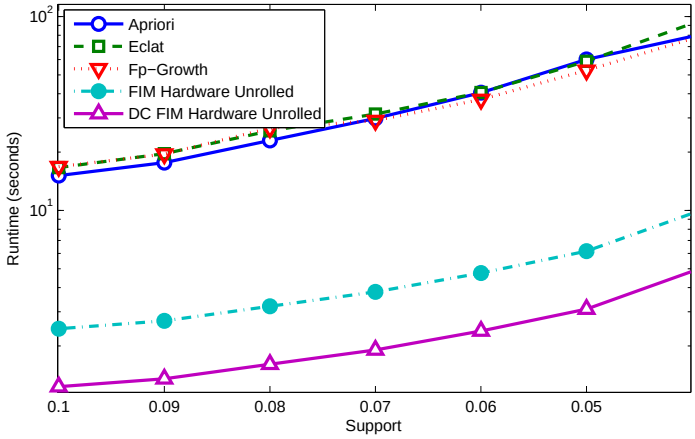


Figure 5.8: Execution time comparison (T40I3N500k).

For example using the $T90L5N499k$ dataset, the speedup achieved by the unrolled hardware architecture is 16x and 36x for the dual core unrolled architecture. The best

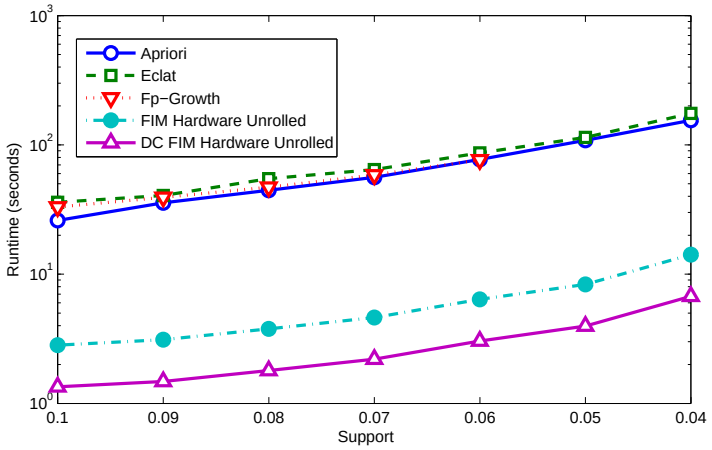


Figure 5.9: Execution time comparison (T40I3N1000k).

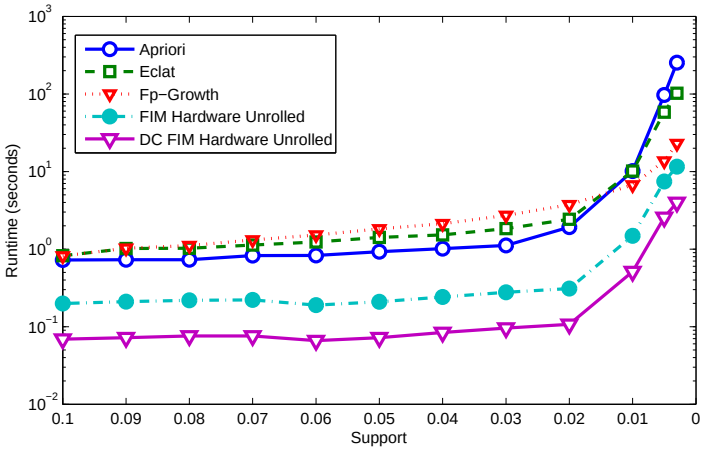


Figure 5.10: Execution time comparison (T40I10D100K).

speed up reported for the dual core unrolled architecture is 48x compared with the best software implementation (without taking in account the Apriori algorithm for chess dataset).

The experiments show that these architectures have good performance for sparse and dense datasets, but for dataset $T40I10D100K02$ the speedup obtained is not considerable because the transactions of the dataset share the same items. The architectures proposed are good for sparse and datasets with a big number of items because the partition proposed by the search strategy keeps in memory compact binary vectors.

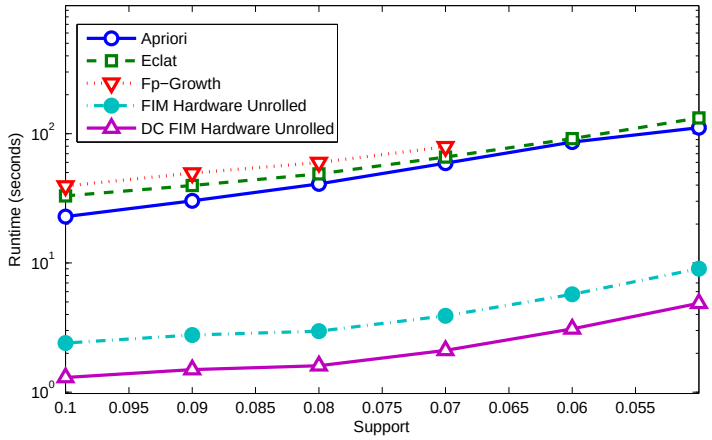


Figure 5.11: Execution time comparison (T60I5N500k).

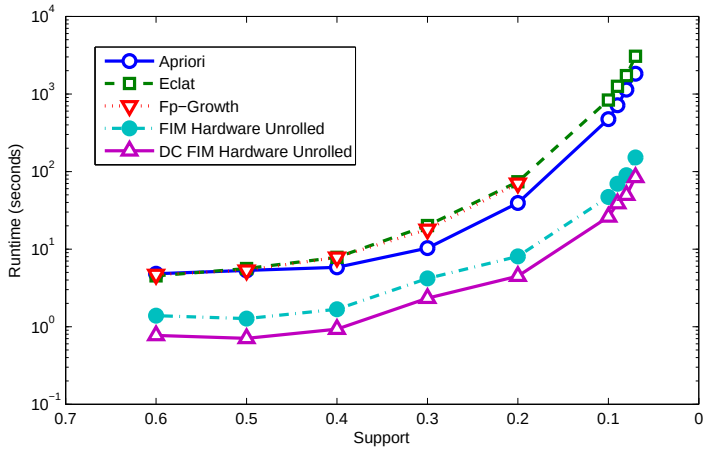


Figure 5.12: Execution time comparison (T90I5N499k).

These binary vectors are used to generate the $k-itemsets$ in consequence the complete dataset only is read twice (in the 1-itemset and 2-itemset generation). Also, the partition in equivalence classes reduces the number of itemsets generated in each iteration. For example, Apriori in one iteration generates all the $k+1-itemsets$ from $k-itemsets$. Instead the proposed architectures only generate the $k+1-itemsets$ of the current equivalence class, once that they have been processed they are not necessary anymore, reducing the memory usage.

The dual core strategy gets the best results in each experiment reducing the execution

Table 5.4: Hardware Resources used by the one core of the unrolled architecture.

Name	BRAM	DSP48E	FF	LUT
Expression	-	-	0	5847
Instance	200	16	248	6648
Memory	2	-	-	-
Multiplexer	-	-	-	51244
Registers	-	-	70224	-
Shift Memory	-	-	0	91
Total	202	16	70472	63830
Available	280	220	106400	53200
Utilization (%)	**72**	**7**	**66**	**120**

time considerably for each dataset. Also, an array of processor could be implemented but the principal inconvenient is the area resources employed and each processor element needs an independent off-chip memory to avoid memory conflicts.

The operation frequency reported is 114 MHz, and the area resources are described in Table 5.4 for the one core unrolled architecture. The unrolled hardware architecture has a considerable consumption of LUTs and Flip Flops, in consequence this architecture does not fit in the Zynq 7020 device that is a low-end device. The increment is because two register banks of 100 words of 32 bits are implemented and multiplexers are required to interconnect the registers to select the desired operation with the data of those registers. For the unrolled intersection module, 4400 LUTs were employed, and it requires one clock cycle to perform the AND operation of 100 words of 32 bits. The support counting module requires 200 BRAMs modules, 248 flip-flops, and 2258 LUTs; the BRAMs modules are necessary to store the look-up tables that determines how many set bits contains the 100 32-bit words. This operation only uses three clock cycles to count the set bits in the register bank. The experiments show that an unrolled hardware architecture gets better speed up compared with the previous architecture reported, and the memory constraints are approached using a partition strategy of the search space.

5.5 Summary

In this chapter, the results and performance evaluation were presented. The evaluation was performed using synthetics datasets used in the literature. All the developed architectures have been tested in the same circumstances, and the reported data is in area and operation frequency. The compact hardware implementation gets a poor speed up compared to the unrolled architecture, but its speedup can be increased attaching more processor elements. The dual core unrolled architecture has obtained the best performance of all our implementations, and it has proved that frequent itemset mining can be accelerated using specialized hardware architectures.

6

Conclusions and Future Work

In this dissertation, a search strategy for Frequent Itemset Mining, and it's hardware implementation have been presented. The proposed search strategy fits well for hardware implementations; the strategy splits the search space into separate equivalence classes making disjoint sets of the original dataset. In consequence, the amount of itemsets stored in the memory is reduced, this is a real advantage for memory constrained scenarios like in the hardware architecture development. Another advantage of the partition into separate equivalence classes is that the equivalence classes can be distributed among a set of processor elements to parallelize and distribute the workload.

Two hardware architectures and their dual-core implementations for Frequent Itemset Mining has been proposed, the first one called hardware architecture based on the proposed search strategy and the second one called unrolled hardware architecture. The performance of both hardware architectures has been compared with software implementations of Apriori, Eclat and FP-Growth [11, 12].

For the first architecture, a one core implementation and a dual core implementation have been developed. Both implementations have obtained better execution times for almost all the datasets, for only one dataset FP-Growth obtained better execution because the characteristics of such dataset. The most remarkable characteristic of this architecture is that gets a 2.5x to 6x speedup despite its compactness.

For the second architecture, also a one core and a dual core implementation have developed. Both implementations have also been compared with Apriori, Eclat, and FP-Growth. For all the datasets, the unrolled architecture reports the best execution times. With the experiments performed over this architecture, it has been shown that an acceleration of the Frequent Itemset Mining problem can be achieved for dense and sparse datasets. The hardware usage of this architecture can be justified because a remarkable acceleration is achieved. There is a cost-benefit tradeoff between hardware usage and execution time.

The objectives presented in the first chapter of this dissertation have been fulfilled, because an efficient search strategy for hardware architectures has been proposed and a remarkable speed up for Frequent Itemset Mining problem has been achieved. Also, this hardware architecture can process any dataset regardless of its size.

6.1 Future Work

Based on the results obtained in this dissertation, there are some variations of the proposed hardware architecture that can be explored. First, it is possible to implement an array of processor elements, in other words, scale up the proposed dual-core architecture from 2 to n processor elements, in consequence a better execution time can be achieved. Therefore, it is necessary to introduce a fair division of the equivalence classes among an array of processor elements that guarantees that all processor elements have a balanced partition of the search space. For this purpose, it is necessary to perform an analysis of the datasets to determine what is the best partition of the search space.

Second, it is possible to scale up the register banks that store the prefix and the suffix in the proposed hardware architecture, in consequence more than 100 32-bit words can be processed in parallel but it is necessary to find the most efficient way to achieve this goal.

Finally, as an immediate goal, it is necessary to perform and study of memory consumption of the proposed search strategy to validate the fact that the proposed architecture saves memory compared with other algorithms proposed in the literature.

References

[1] János Abonyi. A novel bitmap-based algorithm for frequent itemsets mining. In *Computational Intelligence in Engineering*, pages 171–180. Springer, 2010.

[2] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD Record*, volume 22, pages 207–216. ACM, 1993.

[3] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499, 1994.

[4] Altera. *Nios Development Board*, 7 2003. V. 1.1.

[5] AG Anan'ko, KF Lysakov, M Yu Shadrin, and MM Lavrentiev. Development and application of an fpga-based special processor for solving bioinformatics problems. *Pattern Recognition and Image Analysis*, 21(3):370–372, 2011.

[6] Zachary K Baker and Viktor K Prasanna. Efficient hardware data mining with the apriori algorithm on fpgas. In *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, pages 3–12. IEEE, 2005.

[7] Zachary K Baker and Viktor K Prasanna. An architecture for efficient hardware data mining using reconfigurable computing systems. In *Field-Programmable Custom Computing Machines, 2006. FCCM'06. 14th Annual IEEE Symposium on*, pages 67–75. IEEE, 2006.

[8] Michael JA Berry and Gordon S Linoff. *Data mining techniques: for marketing, sales, and customer relationship management*. John Wiley & Sons, 2004.

[9] Benjamin Block, Peter Virnau, and Tobias Preis. Multi-gpu accelerated multi-spin monte carlo simulations of the 2d ising model. *Computer Physics Communications*, 181(9):1549–1556, 2010.

[10] Francesco Bonchi and Bart Goethals. Fp-bonsai: the art of growing and pruning small fp-trees. In *Advances in Knowledge Discovery and Data Mining*, pages 155–160. Springer, 2004.

[11] Christian Borgelt. Efficient implementations of apriori and eclat. In *FIMI'03: Proceedings of the IEEE ICDM workshop on frequent itemset mining implementations*, 2003.

[12] Christian Borgelt. An implementation of the fp-growth algorithm. In *Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations*, pages 1–5. ACM, 2005.

[13] Christian Borgelt. Christian borgelt's web pages. url-http://www.borgelt.net/fimgui.html, 2015.

[14] Sebastian Breß, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. Gpu-accelerated database systems: Survey and open challenges. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XV*, pages 1–35. Springer, 2014.

[15] Toon Calders, Calin Garboni, and Bart Goethals. Efficient pattern mining of uncertain data with sampling. In *Advances in Knowledge Discovery and Data Mining*, pages 480–487. Springer, 2010.

[16] Leonardo Chang, José Hernández-Palancar, L Enrique Sucar, and Miguel Arias-Estrada. Fpga-based detection of sift interest keypoints. *Machine vision and applications*, 24(2):371–392, 2013.

[17] Rui Chang and Zhiyi Liu. An improved apriori algorithm. In *Electronics and Optoelectronics (ICEOE), 2011 International Conference on*, volume 1, pages V1–476. IEEE, 2011.

[18] Shuai Che, Jie Li, Jeremy W Sheaffer, Kevin Skadron, and John Lach. Accelerating compute-intensive applications with gpus and fpgas. In *Application Specific Processors, 2008. SASP 2008. Symposium on*, pages 101–107. IEEE, 2008.

[19] Alok Choudhary, Ramanathan Narayanan, Berkin Özıs Ikyılmaz, Gokhan Memik, Joseph Zambreno, and Jayaprakash Pisharath. Optimizing data mining workloads using hardware accelerators. In *In Proc. of the Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW*. Citeseer, 2007.

[20] Thomas H Davenport, Paul Barth, and Randy Bean. How 'big data'is different. *MIT Sloan Management Review*, 54(1), 2013.

[21] Mike Estlick, Miriam Leeser, James Theiler, and John J Szymanski. Algorithmic transformations in the implementation of k-means clustering on reconfigurable hardware. In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pages 103–110. ACM, 2001.

[22] Wenbin Fang, Ka Keung Lau, Mian Lu, Xiangye Xiao, Chi Kit Lam, Philip Yang Yang, Bingsheng He, Qiong Luo, Pedro V Sander, and Ke Yang. Parallel data mining on graphics processors. *Hong Kong University of Science and Technology, Tech. Rep. HKUST-CS08-07*, 2, 2008.

[23] Wenbin Fang, Mian Lu, Xiangye Xiao, Bingsheng He, and Qiong Luo. Frequent itemset mining on graphics processors. In *Proceedings of the fifth international workshop on data management on new hardware*, pages 34–42. ACM, 2009.

[24] Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From data mining to knowledge discovery in databases. *AI magazine*, 17(3):37, 1996.

[25] R Ferguson. It's all about the platform: What walmart and google have in common. *Sloan Management Review*, 2013.

[26] GiDEL. *PROCStar III PCIe x8 Computation Accelerator*, 2 2009. V. 1.

[27] Bart Goethals and Mohammed J Zaki. Fimi'03: Workshop on frequent itemset mining implementations. In *Third IEEE International Conference on Data Mining Workshop on Frequent Itemset Mining Implementations*, pages 1–13, 2003.

[28] Zhengyang Guo, Wenbo Xu, and Zhilei Chai. Image edge detection based on fpga. In *Distributed Computing and Applications to Business Engineering and Science (DCABES), 2010 Ninth International Symposium on*, pages 169–171. IEEE, 2010.

[29] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *ACM SIGMOD Record*, volume 29, pages 1–12. ACM, 2000.

[30] Raudel Hernández-León, J Hernández-Palancar, Jesús A Carrasco-Ochoa, and José Fco Martínez-Trinidad. Algorithms for mining frequent itemsets in static and dynamic datasets. *Intelligent Data Analysis*, 14(3):419–435, 2010.

[31] Hanaa M Hussain, Khaled Benkrid, Huseyin Seker, and Ahmet T Erdogan. Fpga implementation of k-means algorithm for bioinformatics application: An accelerated approach to clustering microarray data. In *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, pages 248–255. IEEE, 2011.

[32] N Jayalakshmi, V Vidhya, M Krishnamurthy, and A Kannan. Frequent itemset generation using double hashing technique. *Procedia Engineering*, 38:1467–1478, 2012.

[33] Ferenc Kovacs and János Illés. Frequent itemset mining on hadoop. In *Computational Cybernetics (ICCC), 2013 IEEE 9th International Conference on*, pages 241–245. IEEE, 2013.

[34] B Santhosh Kumar and KV Rukmani. Implementation of web usage mining using apriori and fp growth algorithms. *Int. J. of Advanced Networking and Applications*, 1(06):400–404, 2010.

[35] Phuong-Thanh La, Bac Le, and Bay Vo. Incrementally building frequent closed itemset lattice. *Expert Systems with Applications*, 41(6):2703–2712, 2014.

[36] You Li, Kaiyong Zhao, Xiaowen Chu, and Jiming Liu. Speeding up k-means algorithm by gpus. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 115–122. IEEE, 2010.

[37] Ming-Yen Lin, Pei-Yu Lee, and Sue-Chen Hsueh. Apriori-based frequent itemset mining algorithms on mapreduce. In *Proceedings of the 6th international conference on ubiquitous information management and communication*, page 76. ACM, 2012.

[38] Andrew McAfee, Erik Brynjolfsson, Thomas H Davenport, DJ Patil, and Dominic Barton. Big data. *The management revolution. Harvard Bus Rev*, 90(10):61–67, 2012.

[39] Alejandro Mesa, Claudia Feregrino-Uribe, René Cumplido, and José Hernández-Palancar. A highly parallel algorithm for frequent itemset mining. In *Advances in Pattern Recognition*, pages 291–300. Springer, 2010.

[40] David L Olson and Dursun Delen. *Advanced data mining techniques*. Springer Science & Business Media, 2008.

[41] Matthew Eric Otey, Srinivasan Parthasarathy, Chao Wang, Adriano Veloso, and Wagner Meira. Parallel and distributed methods for incremental frequent itemset mining. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 34(6):2439–2450, 2004.

[42] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J Purcell. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007.

[43] Jong Soo Park, Ming-Syan Chen, and Philip S Yu. *An effective hash-based algorithm for mining association rules*, volume 24. ACM, 1995.

[44] Nunu Ren, Jimin Liang, Xiaochao Qu, Jianfeng Li, Bingjia Lu, and Jie Tian. Gpu-based monte carlo simulation for light propagation in complex heterogeneous tissues. *Optics express*, 18(7):6811–6823, 2010.

[45] Philip E Ross. Top 11 technologies of the decade. *IEEE Spectrum*, 48(1):27–63, 2011.

[46] Pratanu Roy, Jens Teubner, and Gustavo Alonso. Efficient frequent item counting in multi-core hardware. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1451–1459. ACM, 2012.

[47] Philip Russom et al. Big data analytics. *TDWI Best Practices Report, Fourth Quarter*, 2011.

[48] Neil Savage. Twitter as medium and message. *Communications of the ACM*, 54(3):18–20, 2011.

[49] Ashok Savasere, Edward Robert Omiecinski, and Shamkant B Navathe. An efficient algorithm for mining association rules in large databases. 1995.

[50] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. Memory-efficient frequent-itemset mining. In *Proceedings of the 14th International Conference on Extending Database Technology*, pages 461–472. ACM, 2011.

[51] Shaobo Shi, Yue Qi, and Qin Wang. Accelerating intersection computation in frequent itemset mining with fpga. In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*, pages 659–665. IEEE, 2013.

[52] Andreia Silva and Cláudia Antunes. Pattern mining on stars with fp-growth. In *Modeling Decisions for Artificial Intelligence*, pages 175–186. Springer, 2010.

[53] R Sumithra and Sujni Paul. Using distributed apriori association rule and classical apriori mining algorithms for grid based knowledge discovery. In *Computing Communication and Networking Technologies (ICCCNT), 2010 International Conference on*, pages 1–5. IEEE, 2010.

[54] Song Sun, Michael Steffen, and Joseph Zambreno. A reconfigurable platform for frequent pattern mining. In *Reconfigurable Computing and FPGAs, 2008. ReConFig'08. International Conference on*, pages 55–60. IEEE, 2008.

[55] Song Sun and Joseph Zambreno. Design and analysis of a reconfigurable platform for frequent pattern mining. *Parallel and Distributed Systems, IEEE Transactions on*, 22(9):1497–1505, 2011.

[56] DW Thoni and Alfred Strey. Novel strategies for hardware acceleration of frequent itemset mining with the apriori algorithm. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 489–492. IEEE, 2009.

[57] Stanimire Tomov, Rajib Nath, Hatem Ltaief, and Jack Dongarra. Dense linear algebra solvers for multicore with gpu accelerators. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010.

[58] Fan-Chen Tseng. Mining frequent itemsets in large databases: The hierarchical partitioning approach. *Expert Systems with Applications*, 40(5):1654–1661, 2013.

[59] Le Wang, Lin Feng, PL Jing Zhang, and Pengyu Liao. An efficient algorithm of frequent itemsets mining based on mapreduce. *Journal of Information and Computational Science*, 11(8):2809–2816.

[60] Ying-Hsiang Wen, Jen-Wei Huang, and Ming-Syan Chen. Hardware-enhanced association rule mining with hashing and pipelining. *Knowledge and Data Engineering, IEEE Transactions on*, 20(6):784–795, 2008.

[61] Ian H Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[62] Di Wu and Andreas Moshovos. Image signal processors on fpgas. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 176–176. IEEE, 2014.

[63] Xilinx. *Virtex-4 Family Overview*, 8 2010. V. 3.1.

[64] Xilinx. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, 6 2011. V. 5.0.

[65] Xilinx. *Virtex-II Platform FPGAs: Complete Data Sheet*, 5 2014. V. 4.0.

[66] Xilinx. *Virtex-5 Family Overview*, 8 2015. V. 5.1.

[67] Xilinx. *Virtex-6 Family Overview*, 8 2015. V. 2.5.

[68] Mohammed Javeed Zaki. Scalable algorithms for association mining. *Knowledge and Data Engineering, IEEE Transactions on*, 12(3):372–390, 2000.

[69] Fan Zhang, Yan Zhang, and Jason D Bakos. Accelerating frequent itemset mining on graphics processing units. *The Journal of Supercomputing*, 66(1):94–117, 2013.

[70] Yan Zhang, Fan Zhang, Zheming Jin, and Jason D Bakos. An fpga-based accelerator for frequent itemset mining. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 6(1):2, 2013.

[71] Zhigang Zhang, Genlin Ji, and Mengmeng Tang. Mreclat: An algorithm for parallel mining frequent itemsets. In *Advanced Cloud and Big Data (CBD), 2013 International Conference on*, pages 177–180. IEEE, 2013.