UNIVERSITY CA' FOSCARI OF VENICE

TURKU UNIVERSITY OF APPLIED SCIENCES

B.Sc. in Computer Science

Empirical Analysis of a Parallel Data Mining Algorithm on a Graphic Processor

Candidate
Davide Berdin

Supervisor
Prof. Claudio Silvestri

Referee
Prof. Jari-Pekka Paalassalo

November 2nd, 2012

Viagiar descanta,
ma chi parte mona, torna mona.
Antico detto veneziano

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

In this thesis, we analyze in an empirical way a different approach of the algorithm SPAM (Sequential PAttern Mining using A Bitmap Representation) made by J. Ayres, J. Gehrke, T. Yiu and J. Flannick from Cornell University, exploiting GPUs.

SPAM is a novel approach for FSM (Frequent Sequence Mining) where the algorithm is not looking for a single item during the mining process but for a set of items that customers have taken in their transactions. Basically we can think of Spam as a supermarket where the customers have the "Fidelity card" and that the supermarket can track the transactions. What Spam does is to calculate which are the most bought items per customer.
But the question is: Why graphic cards ? The answer is pretty easy: graphic cards can benefit of a rich amount of data parallelism allowing many arithmetic operations to be safely performed on program data structures in a simultaneous manner. In our tests we noticed that performing a massive amount of multiplications (or any other kind of operation) with the CPU took at least the 60% more rather than the GPU. Just with this simple case, we could understand the potential of GPUs.

ACM Classification: B.2.1, B.2.2, B.2.4, B.3.2, B.3.3, H.2.8
Keywords: GPGPU, CUDA, Frequent itemsets, Frequent sequences

Acknowledgements

Many people have been important to this work, with their guidance, suggestions and corrections.
First of all, I would like to thank my teachers: Prof. Claudio Silvestri introduced me into the Data Mining and Parallel Programming world that I found absolutely fascinating. He supported me during the whole period while I was writing the thesis giving good explanations for a good understanding of GPU programming and data structure algorithms.

In the same way, I'd like to thank my referee Jari-Pekka Paalassalo who always supported my work even when it apparently seemed to be unfruitful. During these months he has made available his support and many times he helped me to figure out when I was stuck or without any ideas for continuing the project.

This year abroad gave me the opportunity to meet a lot of people from many parts of Europe. The Embedded Software lab at ICT-Talo has been a very nice environment to work at, and I would like to thank everybody there: I would like to show my gratitude to Till Riemer who has been a perfect colleague to work with and, more important, a good friend.

This thesis is dedicated to my family. So many thanks to my parents and my brother Elia Berdin for their unconditional support: I'd have never reached this goal without their help. It has always been important to me to know that there is a safe place where I can always rest. Their trust in me has been really motivating especially in those moments that you would give up. I will never forget it!

Chapter 1

Introduction

"I find that a great part of the information I have
was acquired by looking up something and
finding something else on the way"

Franklin Pierce Adams
Reader's Digest (1960)

Progress in digital data acquisition and storage technology has resulted in the growth of huge databases. This has occurred in all areas of human endeavor, from the mundane (such as supermarket transaction data, credit card usage records and telephone call details) to the more exotic (such as images of astronomical bodies, molecular databases, and medical records). Little wonder, then, that interest has grown in the possibility of mining these data, of extracting from them information that might be of value to the owner of the database. The discipline concerned with this task has become known as *Data Mining*.

Data mining is the analysis of (often large) observational data sets to find unsuspected relationships and to summarize the data in novel ways that are both understandable and useful to the data owner. The relationships and summaries derived through a data mining exercise are often referred to as *Models* or *Patterns*. [1] From now, I will reserve the term *Data Mining* to the first meaning, using the more general KDD - *Knowledge Discover in Databases* - for the whole workflow that is needed in order to apply Data Mining algorithms to real world issues.

The kind of knowledge we are interested in, together with the organization of input data and the criteria used to discriminate among useful and useless information, contributes to characterize a specific data mining problem and its possible algorithmic solutions. Common data mining tasks are the classification of new objects according to a scheme learned from examples, the partitioning of a set of objects into homogeneous subsets, the extraction of association rules and numerical rules from a database. [2]

Chapter 2

CUDA Framework

"The real technology behind all our other
technologies is language.
It actually creates the world our consciousness lives in."

Andrei Codrescu

2.1 What is CUDA?

CUDA™ is a parallel computing platform and programming model invented by NVIDIA. It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU). With millions of CUDA-enabled GPUs sold to date, software developers, scientists and researchers are finding broad-ranging uses for GPU computing with CUDA.

To a CUDA™ programmer, the computing system consists of a *host*, which is a traditional central processing unit (CPU), and one or more *devices*, which are massively parallel processors equipped with a large number of arithmetic execution units. In modern software applications, program sections often exhibit a rich amount of data parallelism, a property allowing many arithmetic operations to be safely performed on program data structures in a simultaneous manner. The CUDA devices accelerate the execution of these applications by harvesting a large amount of data parallelism.

2.2 Program Structure

A CUDA program consists of many phases that are executed on either the CPU (Host) or a device such as a GPU. All the phases are implemented in the part called *host code*. The other phases, which exhibit a rich amount of data parallelism, are implemented in the part called *device code*. Of course the NVDIA C compiler (nvcc) splits the parts related to the host from these related to the device. The host code is written using ANSI C code and it is compiled with the standard C compilers and runs as an normal CPU process. The device's code is also written using ANSI C extended with keywords for labeling data parallelism functions, called *kernels*, and their data structures that belong to them.

"The kernel functions generate a large number of threads to exploit data parallelism. It is worth nothing that CUDA threads are of much lighter weight than the CPU threads. CUDA programmers can assume that these threads take very few cycles to generate and schedule due to efficient hardware support. This is in contrast with the CPU threads that typically requires thousands of clock cycles to generate and schedule".[3]

As you can evaluate from Figure 2.1, the execution starts with a CPU execution and then, when a kernel function is *launched*, the execution is moved to the GPU, where a large number of threads are generated. *All the threads that are generated by a kernel during an invocation are collectively called a grid.* When all threads have completed their execution, the corresponding grid terminates and the execution continues on the CPU until another kernel function is launched.

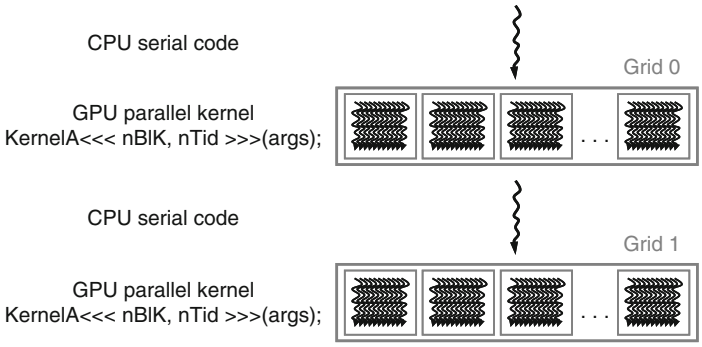

Figure 2.1: Execution of a CUDA program.

As the reader can evaluate from the picture above, every time that a kernel is launched, the grid is composed with either one or several subsets of blocks. Indeed, a grid is a two dimensional level where blocks are distributed. This means that a grid can have just precise amount of blocks along the x *axes* and a precise amount of blocks along the y *axes* as well. But, a single block is just a set of threads which is composed in a three dimensional way. Based on these instructions, the programmers have every time to define the amount of blocks and threads per blocks for taking advantage of data parallelism.

Figure 2.2 depicts how the structure is organized.

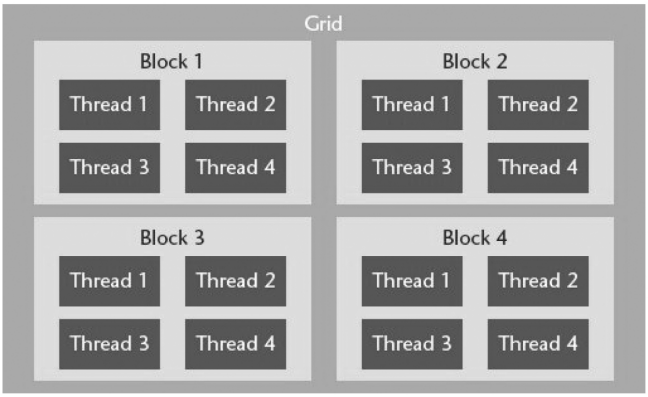

Figure 2.2: Thread distribution into the two level hierarchy of CUDA.

2.3 Kernel Functions and Threading Operations

In CUDA, a kernel function specifies the code to be executed by all threads during a parallel process. *Because all of these threads execute the same code, CUDA programming is an instance of the well-known single-program, multiple-data (SPMD) parallel programming style, a popular programming style for massively parallel computing system.*[4] As said above, CUDA provides some new keywords beyond ANSI C standard code. In general, CUDA extends C function declarations with three qualifier keywords (see Table 2.1).
Some definitions:

- The `__global__` keyword indicates that it is a kernel function. It will be executed on the device and can only be called from the CPU to generate a grid of threads on the GPU.
- The `__device__` keyword indicates that it is a device function. It will be executed on a CUDA device and can only be called from a kernel function or another device function. Device functions can have neither recursive nor another device function.
- The `__host__` keyword indicates that is a host function. It is simply a traditional C function that executes on the host and can only be called from another host function. By default, all functions in a CUDA program are host functions if they don't have any of the CUDA keywords in their declaration.

	Executed on the:	Only callable from the:
__device__ float DeviceFunc()	device	device
__global__ void KernelFunc()	device	host
__host__ float HostFunc()	host	host

Table 2.1: CUDA extensions to C functional declaration.

CUDA offers other notable extensions of ANSI C like *threadIdx.x* and *blockIdx.x*, which refer to the thread indices of a thread and an indices of a block respectively. It has to be noticed that all the threads are executing the same kernel code. The keywords *threadIdx.x* and *threadIdx.y* identify predefined variables that allow the thread to access the hardware register at runtime that furnish the identifying coordinates to the thread.

As said above, when a kernel is invoked, it's executed as a *grid* of parallel threads. Those two keywords appear as well-defined variables that can be accessed just with kernel functions and assigned at CUDA runtime:
- *blockIdx.x* for block index;
- *threadIdx.x* for thread index.

Additional *reserved* variables, *gridDim* and *blockDim*, provide the dimension of the whole grid and the dimension of each block as well.
Figure 2.3 shows the whole organization:

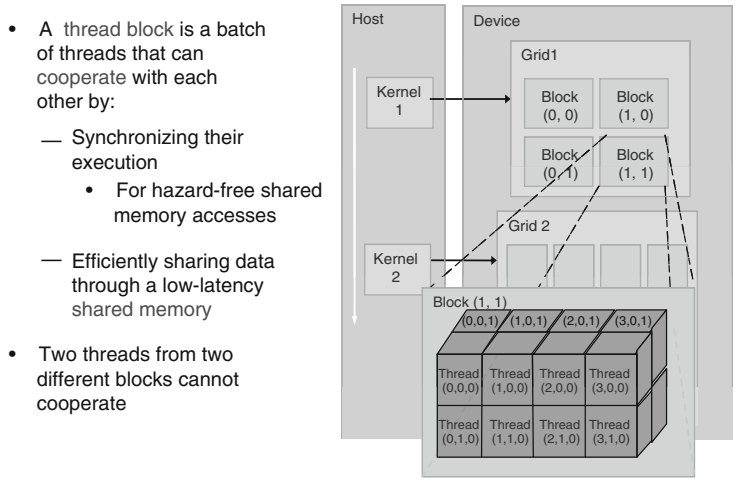

Figure 2.3: CUDA Thread Organization.

The total size of a block is limited to 512 threads, with flexibility in distributing these elements into the three dimensions as long as the total number of threads does not exceed 512. But in our case, the GPU that we used could have as maximum value 1024 threads per block.

2.4 Cuda Device Memory Types

In this section we describe different memory types [5][6]. There are several different types that CUDA has access to and for all of them there different features to keep in mind during the development. For example *Global Memory* has a very large address space but the time required to access this memory is very high, rather then *Shared Memory* where the address space is very small but the memory latency is very low. Those are just some examples; Figure 2.4 shows the whole organization:

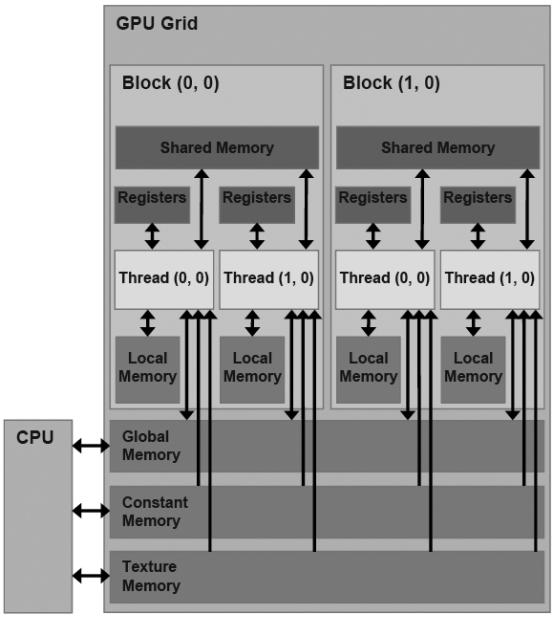

Figure 2.4: Example of CUDA memory organization for each block.

2.4.1 Register

Register memory access is very fast but the number of available registers for each block is very limited. Each register's variable is private for the thread, so each thread will have a private version of every register variable and, of course, the life-time is as long as the thread exists. Not less important, register variables can be both read and written inside the kernel and do not be synchronized.

2.4.2 Local

Local memory is used from the compiler when any variable that cannot fit into the register space. Thus, it will put automatically in another space address. Local memory has the same memory latency as global memory, so it takes a lot before to transfer all the data. This type is allowed just on devices with CUDA capability 2.x. As for register variable, local variables has life-time as long as the thread who it belongs to.

2.4.3 Shared

Variables that are declared with *__shared__* attribute are stored in shared memory. Accessing the shared memory is very fast although each streaming multiprocessor has a limited amount of shared memory address space. Shared variables have to be declared within the scope of the kernel but the life time is as long as the block is used. Hence, when a block has finished the execution, the shared memory that was defined in the kernel cannot be accesed again.
Since shared memory is very fast, it is more efficient to copy part of the data in that space address because the memory latency is quite low compared with global memory. In this way it is reduced the number of access to the global memory. Modification of shared memory must be synchronized unless it is sured that each thread will only access the memory that will not be read-from or written-to by other threads in the block. In the next section we will talk about the whole thread synchronization. A good point to use shared memory it is that all threads can share this kind of memory and being sure that all of them wil stay in the same SM (Streaming Multiprocessor - explained later) disputing the same resources.

2.4.4 Global

Global variables are those which are declared with *__device__* attribute and since they are declared outsite of the kernels functions (global scope) they are stored in global memory. The time requested for accesing to those variables is very high but the advantage is that the amount of space address is huge compared with shared memory. Global memory has a life-time of the application and is accessible to all threads of all kernels. The programmer has to pay attention when global memory is going to be either read or written because thread execution cannot be synchronized across different blocks.

2.4.5 Constant

Variables that are decorated with the *__constant__* attribute are declared in Constant memory. Like global variables, constant variables must be declared in global scope, so outside of kernel functions. Constant variable shares the same memory banks as global memory but unlike global memory, there is only a limited amount of constant memory that can be declared. As global memory, also constant memory has life-time of the whole application. It can be accessed by all threads of all kernels and the value will not change across kernel invocations unless explicitly modified by the host process.

Here, the all kind of memories resumed in one table with their own Location and kind of Access as well.

Memory	Location	Access	Scope
Register	On-chip	Read/Write	One thread
Local	Off-chip	Read/Write	One thread
Shared	On-chip	Read/Write	All threads in a block
Global	Off-chip	Read/Write	All threads + host
Constant	Off-chip	Read	All threads + host

Table 2.2: Memory types resume.

2.5 Thread Synchronization

CUDA allows threads in the same block to coordinate their activities using a synchronization function, *__syncthreads()*. When a kernel function is called and the keyword *__syncthreads()* has been invoked, the thread that executes the function call will be held at the calling location until each thread in the block reaches the proper location. [3] reports that, when the synchronization function is placed in an if statement, either all threads in a block execute the path that includes the *__syncthreads()* or none of them do. For an if-*then*-*else* statement, if each path has a *__syncthreads()* statement, then either all threads in a block execute the *__syncthreads()* on the *then* path or all of them execute the *else* path.

An important step for threads is how they are assigned to compute their job. Once a kernel is invoked, the CUDA runtime system generates the corresponding grid of threads. The execution resources are organized into *streaming multiprocessors* - SMs. *In certain situations it could happen that the amount of any one or more types of resources needed for simultaneous execution is not sufficient; in this case the CUDA runtime automatically reduces the number of blocks assigned to each SM until the resources usage is under limit* [3]. Figure 2.5 shows how threads are assignment to SMs.

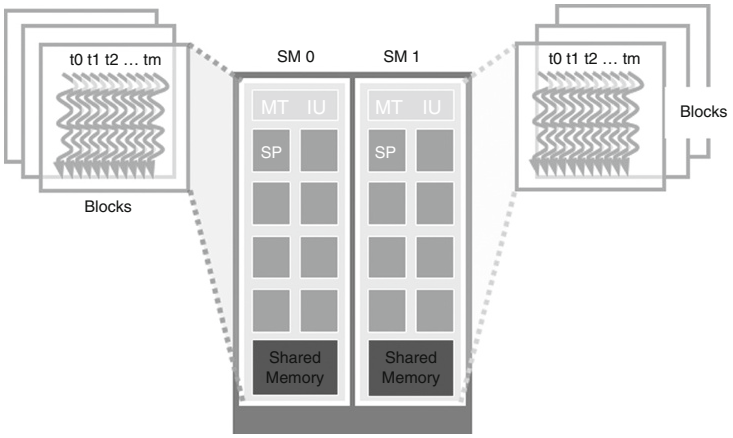

Figure 2.5: Thread block assignment to streaming multiprocessors (SMs).

Chapter 3

Association Mining

"Computers are famous for being able
to do complicated things starting
from simple programs"

Seth Lloyd
Interview at EdgeFoundation (1999)

3.1 Association Mining Rules

The core of associative rules is to describe and analyze strong rules using different measures of interestingness. If, for example, the data represents the objects bought in the same shopping chart by the customers of a supermarket, then the main task will be finding rules relating the fact that a market basket contains an item with the fact that another item has been bought at the same time.

Referring to the supermarket, one of these rules could be *"people who buy Bread and also buy Peanut butter at the same time will also buy Milk later in conf% cases"*, but also the more complex *"people who buy Bread followed by Peanut butter, after some time will also buy Milk later in conf% cases"* where $conf\%$ is the confidence of the rule. Another important measure, frequently used combined with confidence, is the support of a rule, which is defined as the number of records in the database that confirm the rule. Generally, the user specifies minimum thresholds for both, so an interesting rule should have both a high support and a high confidence, i.e. it should be based on a significant number of cases to be useful, and at the same time, there should be few cases in which it is not valid.
As next step, it is necessary to find the, as it called, $frequent\ patterns$, such as patterns that occur in a significant number of records. When those pattern are determined, the actual association rule can be defined as follow: $X \Rightarrow Y$, that it is read as *every time that X occurs in a transaction (sequence), most likely also Y will also occur (later).*

Table 3.1 depicts the previous example:

Transaction	Items
t_1	**Bread, Jelly, Peanut butter**
t_2	**Bread, Peanut butter, Diaper**
t_3	**Bread, Milk, Peanut butter**
t_4	**Beer, Bread**
t_5	**Beer, Milk**

Table 3.1: Market Basket Analysis

In this case $I = \{$Beer, Bread, Jelly, Milk, Peanut butter$\}$ and, for instance the minimum Support is 60% the result will be the set composed by the Itemset {Bread, Peanut butter}.

Going deeply with math definitions, the *Association Rule* is defined as $X \Rightarrow Y$ where $X, Y \subseteq I$ and $X \cap Y = \emptyset$. In the end we have the *Confidence of Association Rule* (α) $X \Rightarrow Y$ which is the ratio of the number of transactions that contain $X \cup Y$ to the number that contain X.

Sequential rules are an extension of association rules, which also considers sequential relationships, but this time the input data are sequences of sets. Continuing with the example of the supermarket (and the table above), each transaction is related to a customer. Hence, the supermarket could track the item bought from each customer and make statistics (or suppositions) that when the customer will come back there are a $conf\%$ cases that he/she buys Peanut butter if he/she buys Bread.

3.2 Goals of Association Mining

There are three goals [7] that Association Mining has to reach:

Association

Association rules are statements of the form $\{X_1, X_2, \ldots, X_n\} \Rightarrow Y$, meaning that if we find all the elements (or items) in the market basket, then we have high probability of finding Y.

Causality

Ideally, we would like to know that in an association rule the presence of $\{X_1, X_2, \ldots, X_n\}$ actually "causes" Y to be bought. Still, "causality" is an exclusive concept, nevertheless, for market basket data, the following test suggests what causality means. But as [8] reports in the Example 1, the item Y can be bought if the association $\{X_1, X_2, \ldots, X_n\} \Rightarrow Y$ have both high $confidence$ and high *support*. A practical example: if we lower the price of Diapers and raise the price of Beer, we can lure diaper buyers, who are more likely to pick up Beer while in the store, thus covering our losses on the Diapers. That strategy works because "Diapers causes Beer". However, working it the other way round, running a sale on Beer and raising the price of Diapers, will not result in Beer buyers buying Diapers in any great numbers, and we lose money.

Frequent Itemset

In several situations, we only care about the first two goals involving sets of items that appear frequently in baskets. For instance, we can't run a good marketing strategy involving items that anyone buys anyway. Hence, much data mining starts with the assumption that we only care about sets of items with high support. We can then find association rules or causalities involving high-support set of items (i.e. $\{X_1, X_2, \ldots, X_n, Y\}$ which must appear in at least a certain percent of the baskets, as we defined above, supporting threshold.

3.3 Frequent Itemset Mining

We formalize the problem of mining frequent itemsets as follows: Let $I = \{x_1, \ldots, x_n\}$ be a set of items and an itemset X is a subset of I. A transactional database is a collection of itemsets such as $D = \{t_1, t_2, \ldots, t_n\}$ with $t_i \geq I$ called transaction. The support of an itemset X is the database D, denoted as $\sigma(X)$ when D is clear from the context, is the ratio of transactions that includes X. Given a minimum support $\overline{\sigma}$, an itemset X such that $\sigma(X) \geq \overline{\sigma}$ is called frequent or large, since they have a large support. The Frequent Itemset Mining problem requires to discover all the frequent itemsets in D given $\overline{\sigma}$. [9]
Moreover, a subset of a frequent itemset is frequent itself. This reason is given by the Anti-Monotonic property, that we explain with the following example:
If $\{AB\}$ is a frequent itemset, both $\{A\}$ and $\{B\}$ are frequent because $\sigma(\{A\}) \geq \sigma(\{AB\})$ and $\sigma(\{B\}) \geq \sigma(\{AB\})$.
Of course, if the $\{A\}$ and {B} is not-frequent as well as {AB} is not frequent becuase the $minimum\_supp > \sigma(\{A\}) \geq \sigma(\{AB\})$. Indeed, those rule are being used for generating the Association Rules as the reader can see above.
The main difficulty of FIM algorithms comes from the large size of the search space. In principle, every itemset in powerset $P(I)$ is potentially frequent, and a whole scan of the dataset is required to calculate its exact support. Indeed, given the Anti-monotonic property, Data mining algorithm can drastically reduce the huge amount of computation that we have without exloiting it. Just imagine with 1 billion of transaction, if we should mine through whole this massive dataset without using this property: the number of candidates producted from the itemsets generation are prohibitive.
As usual we will use a lattice of itemsets, where node are in lexicographical order, to visualize such search space as in Figure 3.2:

TID	Items
1	b, d
2	a, b, c, d
3	a, c, d
4	c

Table 3.2: Dataset D

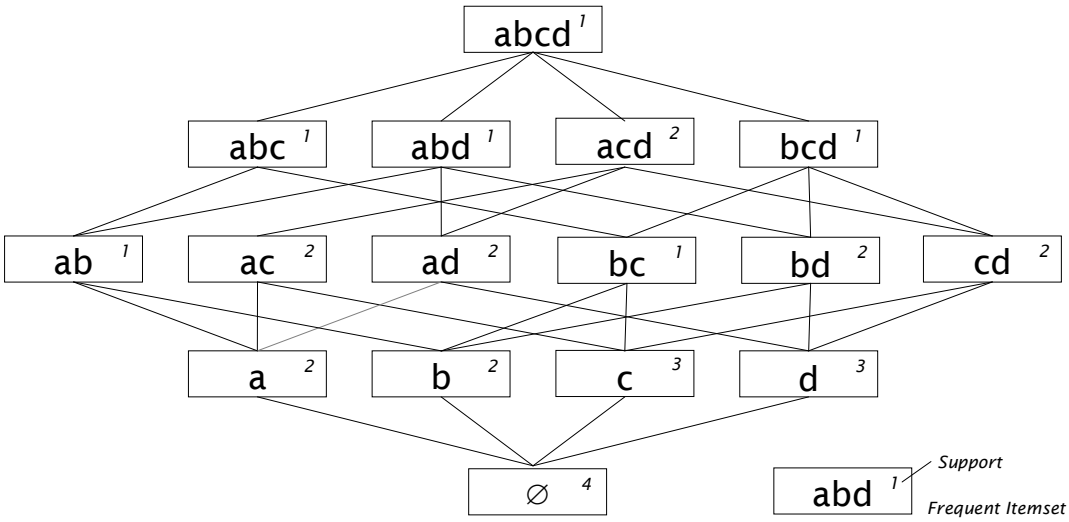

Figure 3.1: (top) The input transactional dataset D, represented in its horizontal form. (bottom) Lattice of all the frequent itemsets with $\sigma = 1$

Two Approaches can be used:

1 Naive or Brute-Force Approach

2 Apriori (see next Section)

The Naive or Brute-Force one is based on the principle that "each itemset in the lattice is a frequent itemset" where every transaction is matched against each candidate. In this way, the Computational Complexity is prohibitive becuase, according with [10], we have $O(MNw)$, where M is the List of Candidates, N is the number of transactions and w is the itemset for each transaction. And this is very expensive becase it produces $M = 2^d$. Continuing with more math, this following formula calculates the number of rules:

$$R = \sum_{k=1}^{d-1} \left[\binom{d}{k} \sum_{j=1}^{d-k} \binom{d-k}{j} \right] = 3^d - 2^{d+1} + 1$$

having d as a unique items.
With different strategies we can reduce the total complexity of the itemset generation, such as:

Candidates (M)
Complete search $M = 2^d$, using pruning techniques for reducing M

Transactions (N)
Reduce size of N as the size of itemset increases

Comparisons (NM)
Use efficient data structures to store the candidates or transactions and there is no need to match every candidate against every transaction

with Apriori we are going to reduce the number of candidates.

3.4 Apriori Algorithm

As [11] says, Frequent itemset mining sets of items that appear in a percentage of transactions with the percentage, called *support*, larger than a given threshold. The *Apriori* algorithm finds all frequent itemsets in multiple passes, given a support threshold. At the first pass, it finds the frequent items. Generally at the l-th pass, the algorithm finds the frequent items each consisting of l items (named l-itemsets). In each pass, it first generates *candidate* (l+1)-itemsets from the frequent l-itemsets, then counts the supports of these candidates and prunes those candidates whose supports are less than a given support threshold. The algorithms ends when no candidate is generated in a pass.

3.4.1 Formal Statement

[12] Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of *items*. Now, let D be a set of transactions, where each transaction T is a set of items such that $T \subseteq I$. Associated with each transaction is a unique identifier, called its TID. The authors say that a transaction T *contains* X, a set of some items in I, if $X \subseteq T$. An *association rule* is an implication of the form $X \Rightarrow Y$, where $X \subset I, Y \subset I$, and $X \cap Y = \emptyset$. The rule $X \Rightarrow Y$ holds in the transaction set D with *confidence* $c\%$ of transaction in D that contain X also contain Y. The rule $X \Rightarrow Y$ has *support* s in the transaction set D if $s\%$ of transaction in D contain $X \cup Y$.

3.4.2 Pseudocode

The following code explains how the Apriori algorithm works:

Algorithm 3.1 Apriori Pseudocode

```
1:  L_1 = {large 1 − itemsets};
2:  for (k := 1; L_{k−1}; k++) do
3:      C_k = apriori − gen(L_{k−1});                ▷ New candidates
4:      for all transaction t ∈ D do
5:          C_t = subset(C_k, t);                    ▷ Candidates contained in t
6:          for all candidates c ∈ C_t do
7:              c.count++;
8:          end for
9:          L_k = {c ∈ C_k | c.count ≥ minsup}
10:     end for
11: end for
12: return Answer = ⋃_k L_k;
```

3.5 Computational Complexity

The computational complexity of the Apriori algorithm can be influenced by the following factors:

Support Threshold
Lowering the support threshold often results in more itemsets being declared as frequent. This has an negative effect on the computational complexity of the algorithm because more candidate itemsets have to be generated and counted.

Number of Items (Dimension)
If the number of items increases, more space is needed for storing the support counts of items. But also, if the number of frequent items grows, even the computation will increase because of the larger number of candidate itemsets generated by the algorithm.

Number of Transactions
Since the Apriori algorithm scans several times the data set, its run time increases with a larger number of transactions.

Average Transaction Width
As [1] reports: *"For dense data sets, the average transaction width can be very large. This affects the complexity of the Apriori algorithm in two ways. First, the maximum size of frequent itemsets tends to increase as the average transaction width increases. As a result, more candidate itemsets must be examined during candidate generation and support counting. Second, as the transaction width increases, more itemsets are contained in the transaction. This will increase the number of hash tree traversals performed during support counting."*

Generation of frequent 1-itemsets
For each transaction, we need to keep updating the support count for every item present in the transaction itself. If w is the average transaction width, this operation requires $O(Nw)$ time, where N is the total number of transactions.

Candidate generation
The candidate generation depends in which data structure is apriori algorithm performing its steps. For a complete example based on Hash Tree data structure, see [1].

Support counting
Each transaction of length $\mid t \mid$ produces $\binom{|t|}{k}$ itemsets of size k. The cost for support counting is $O(N \sum_k \binom{w}{k} \alpha_k)$, where w is the maximum transaction width and α_k is the cost for updating the support count of candidate.

Actaully we cannot define the exact computation complexity of Apriori since those different factors, but still, we have a radical improvment of performance when solutions exploit this one rather than Brute-Force method.

3.6 Frequent Sequences Mining

Sequential pattern mining[1] (FSM) represents an evolution of *Frequent Itemsets Mining* (FIM), allowing also for the discovery of before-after relationships between subsets of input data. The patterns we are looking for are sequences of sets, indicating that the elements of a set occurred at the same time and before the items contained in the following sets. The *"occurs after"* relationship is indicated with an arrow, i.e.[2] $\{Bread, Peanutbutter\} \rightarrow \{Peanutbutter\}$ indicates an occurrence of both item **Bread** and **Peanut butter** followed by an occurrence of item **Peanut butter**. Clearly, the inclusion affinity is more complex than in case of subsets, so it needs to be defined. For now, we consider that a sequence pattern Z is supported by an input sequence IS, if Z can be obtained by removing items and sets from IS. As an example the input sequence $\{Bread, Peanutbutter\} \rightarrow \{Milk\} \rightarrow \{Bread\}$ supports the sequential patterns $\{Bread, Peanutbutter\}, \{Bread\} \rightarrow \{Milk\}, \{Bread\} \rightarrow \{Bread\}$ but not the pattern $\{Bread, Milk\}$, because the occurrence of **Bread** and **Milk** in the input sequence are not simultaneous. We highlight that the "occurs after" relationship is satisfied by $\{Bread\} \rightarrow \{Bread\}$, since anything between the two items can be removed. [13]

[1] In those lines we simply gave a very brief introduction. In the next chapter we will describe deeper with also math definitions of the problem

[2] The following examples are referred to the supermarket which we discussed in the previous paragraph

Chapter 4

SPAM

"There are two ways of constructing a software design.
One way is to make it so simple that there are obviously no deficiencies.
And the other way is to make it so complicated that
there are no obvious deficiencies."

C.A.R. Hoare
Turing Award Lecture (1980)

4.1 Overview

J. Ayeres, J. Gehrke, T. Yiu and J. Flannick [14] introduced an algorithm for mining sequential patterns. Their algorithm is especially efficient when the sequential patterns in the database are very long. A novel *depth-first* search strategy that integrates a depth-first traversal of the search space with effective pruning mechanisms is introduced.

Their algorithm of the search strategy combines a vertical bitmap representation - discussed later - of the database with efficient support counting. An important feature is that it incrementally outputs new frequent itemsets in an online fashion.

I will report the essential parts of their work because some definitions and some names will be useful in the next chapters for a good understanding of our work.

4.2 Problem

The problem of mining sequential patterns and the support-confidence framework were originally proposed by Agrawal and Srikant [15][16]. Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of items. A subset $X \subseteq I$ is an *itemset* and the $\mid X \mid$ is the *size* of X. A *sequence* $s = (s_1, s_2, \ldots, s_m)$ is an ordered list of itemsets, where $s_i \subseteq I, i \in \{1, \ldots, m\}$. The size, m, of a sequence is the number of itemsets in the sequence, i.e. X. The length l of a sequence $s = (s_1, s_2, \ldots, s_m)$ is defined as

$$l \stackrel{\text{def}}{=} \sum_{i=1}^{m} \mid s_i \mid$$

A sequence with length l is called an l-sequence. A sequence $s_a = (a_1, a_2, \ldots, a_n)$ is contained in another sequece $s_b = (b_1, b_2, \ldots, b_m)$ if there exist integers $1 \leqslant i_1 < i_2 < \ldots < i_n \leqslant m$ such that $a_1 \subseteq b_{i1}, a_2 \subseteq b_{i2}, \ldots, a_n \subseteq b_{in}$. If sequence s_a is contained in sequence s_b, then s_a is a *subsequence* of s_b and s_b is a *supersequence* of s_a.

As said above, the algorithm has a database for keeping the sequence. A database D is a set of tuples(cid, tid, X), where cid is the costumer-id, tid is the transaction-id and X is an itemset such that $X \subseteq I$. Each tuple in D is referred to as a *transaction*. Very important thing is that for a given costumer-id, there are no transaction with the same transaction ID.

Definition:
The absolute support of a sequence s_a in the sequence representation of a database D is defined as the number of sequence $s \in D$ that contain s_a.

Definition:
The relative support is defined as the percentage of sequences $s \in D$ that contain s_a.

Given a support threshold $minSup$, a sequence s_a is called a $frequent\ sequential\ pattern$ on D if $sup_D(s_a) \geq minSup$, where $sup_D(s_a)$ is the support of s_a in D.

4.3 The SPAM Algorithm

The originator of [14] based this algorithm on the *Lexicographic Tree of Sequence*. Also in other work such as *MaxMiner*[17] and *MAFIA*[18], it has been used a sort of this technique.

Assume that there is a lexicographic ordering $\leqslant$ of the items I in the database. If item i occurs before item j in the ordering, then we denote this by $i \leqslant_I j$. This ordering can be extended to sequences by defining $s_a \leqslant s_b$ if s_a is a subsequence of s_b. Of course, if s_a is not a subsequence of s_b, then there is no relationship in this ordering.

Now, consider all sequences arranged in a *sequence tree*. The structure is described as follow: the root of the tree is labeled with $\emptyset$. Recursively, if n is a node in the tree, then n's children are all nodes n' such that $n \leqslant n'$ and $\forall m \in T\colon n \leqslant n' \Longrightarrow n \leqslant m$. A particular attention is that given definition the tree is infinite. Due to a finite database, all trees in practice are finite.

Each sequence in the sequence tree can be considered as either a sequence-extended sequence or an itemset-extended sequence, where they are defined as follows:

- *A sequence-extended sequence is a sequence generated by adding a new transaction consisting of a single item to the end of its parent's sequence in the tree.*
- *An itemset-extended sequence is a sequence generated by adding an item to the last itemset in the parent's sequence, such that the item is greater than any item in that last itemset.*

4.4 S-step and I-step

The originator refer to the process of generating sequence-extended sequence as the *sequence-extended step* (S-step), then the process of generating itemset-extended sequences as the *itemset-extended step* (I-step). In this way, the algorithm can associate with each node n in the tree two sets: S_n, the set of candidate items that are considered for a a possible S-step extensions of node n (*s-extensions*), and I_n, which identifies the set of candidate items that are considered for a possible I-step extensions (*i-extensions*).

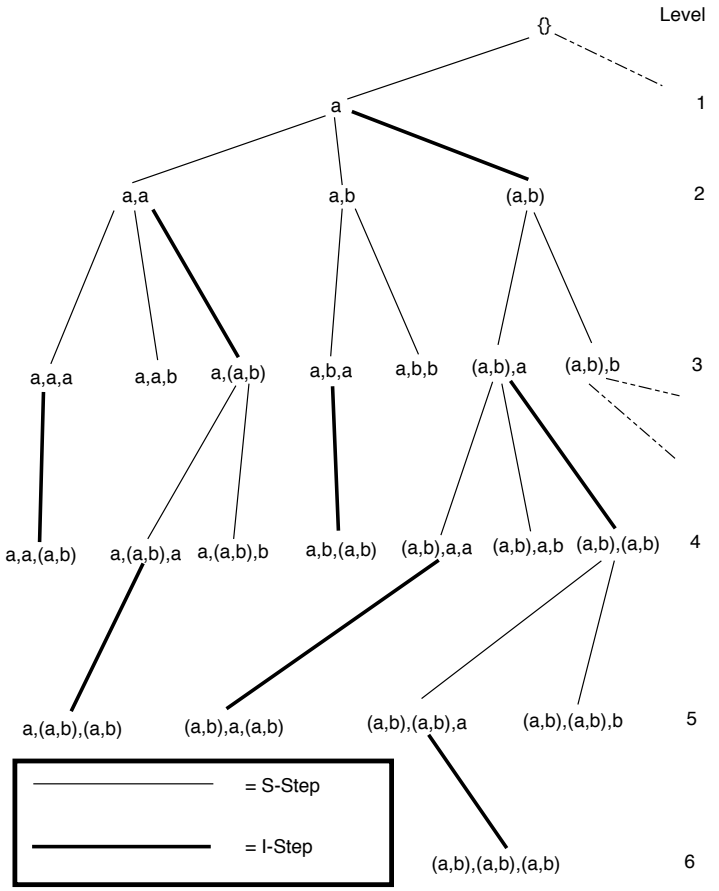

Figure 4.1: Lexicographic Sequence Tree

Figure 3.1 shows an example of a complete sequence tree for *two* items, a and b, given that the maximum size of a sequence is *three*. The top element in the tree is the null sequence and each lower level k contains all of the k-sequences, which are ordered lexicographically with sequence-extended sequences ordered before itemset-extended sequences. Each element in the tree is generated only by either an S-step or an I-step, e.g. sequence $(\{a, b\}, \{b\})$ is generated from sequence $(\{a, b\})$ and not from sequence $(\{a\}, \{b\})$ or $(\{b\}, \{b\})$.

4.5 Pruning

The previous algorithm has a huge search space. In [14] to improve the performance, it is possible to prune candidate s-extensions and i-extensions of a node n in the tree. In the algorithm, the pruning technique are Apriori-based and are aimed at minimizing the size of S_n and I_n at each node n. At the same moment, the pruning guarantee that all nodes corresponding to frequent sequences are visited. The algorithm provides two pruning techniques both for S-step and I-step, but I will not describe in this report. Of course the reader can easily check for pruning technique in [14].

4.6 Data Structure

SPAM uses the *vertical bitmap* representation of the data for an efficient counting. A vertical bitmap is created for each item in the dataset, and each bitmap has a bit corresponding to each transaction in the dataset. If item i appears in transaction j, then the bit corresponding to transaction j of the bitmap for item i is set to *one*; otherwise, the bit is set to *zero*.

This bitmap idea extends naturally to itemsets. Suppose we have a bitmap for item i and a bitmap for item j. The bitmap for the itemset $\{i, j\}$ is simply the bitwise AND of these two bitmaps. Sequences can also be represented using bitmaps. If the last itemset of the sequence is in transaction j and all the other itemsets of the sequence appear in transactions before j, then the bit corresponds to j will be set to *one*; otherwise, it will be set to *zero*. In the research, the authors define $B(s)$ as the bitmap for sequence s.

4.7 Candidate Generation

A very important aspect of the algorithm is how it performs the candidate generations using the bitmap representation described above. There are two subsections related for both S-step processing and I-step processing.

4.7.1 S-step Processing

Suppose to have bitmaps $B(s_a)$ and $B(i)$ for sequence s_a and item i respectively, and that we have to perform a *S-step* on s_a using i. This *S-step* will append the itemset $\{i\}$ to s_a. To perform it, the method is based on these two operations.

First, we need to generate a bitmap from $B(s_a)$ such that all bits less than or equal to k[1] are set to *zero*, and all bits after k are set to *one*. The authors call this bitmap a $transformed\ bitmap$. After that, it is necessary to make a logical AND between the transformed bitmap and the item bitmap. The resultant bitmap has the properties described above and it is exactly the bitmap for the generated sequence. In the real application, the transformation is done using a lookup table.

[1] k is the index of the first bit with value *one* in $B(s_a)$

4.7.2 I-step Processing

Suppose to have bitmaps $B(s_a)$ and $B(i)$ and that we have to perform a *I-step* on s_a using i. This *I-step* will generate a new sequence s_g by adding item i to the last itemset of s_a. The bitmap for s_g should have the property that if a bit has value *one*, then the corresponding transaction j must contain the last itemset in s_g, and all of the other itemsets in s_g must be transactions before j. Now, we should consider the resultant bitmap $B(s_r)$ obtained by doing a logical AND among $B(s_a)$ and $B(i)$. A bit k in $B(s_r)$ have value *one* if and only if bit k in $B(s_a)$ is one and bit *key* in $B(i)$ is also *one*. For bit k of $B(s_r)$ to be *one* the transaction j that corresponds to bit k must, therefore, containing both the last itemset in s_a and the item i. In addition, all of the other itemsets of s_a must appear transaction before j. It follows that $B(s_r)$ satisfies each of the requirements that $B(s_g)$ must satisfy; $B(s_r)$ is therefore exactly the bitmap for the generated sequence.

Chapter 5

gpuSPAM

"Program testing can be used to show
the presence of bugs, but never to show their absence"

Edsger Dijkstra

5.1 An overview of gpuSPAM

The basic idea behind gpuSPAM is based on gpuDCI [19], where the application starts the computation on CPU because the dataset is too large to fit it in the GPU's global memory. After the S-step pruning and I-step Pruning as described above, the pruned dataset will be moved to the GPU just after the S-step and I-step Processing. The GPU successively will compute the support computation; but still, after swapping to GPU, the CPU will manage the patterns, the candidate generation and store patterns that are frequent according to the support computed by the GPU.

As in *gpuDCI*, there are several conditions that we need to satisfy for having a good usage of the GPU. In particular, there are three important aspects related to processor utilization, memory access patterns and blocking operations. For the first one we need to be sure that every core is having enough workload and also enough resources for executing the assigned computation. The second condition is decisive because we must ensure coalesced access to global memory by aligning memory access to avoid serialization and get the whole process rather slow. The last goal is to minimize the number of synchronizations, especially that kind of operations that is causing global synchronization such as memory transfer, kernel launches and, of course, other blocking operations.

In this work, we analyze two different approaches called *Candidate-wise* and *Transaction-wise*. These two strategies resulted a good compromise between speedup and gpu-workload, with different datasets. For both of them we will briefly describe the general features.

5.2 Candidate-wise parallelization

In this approach each GPU multiprocessor works on the intersection and count operations related to a different candidate, whereas the cores of the same multiprocessor work on the same intersection or candidate. In general, thread blocks are taking care of a block of candidate that are processed one by one, whereas threads are in charge of the either intersection or count among a portion of bitmap. Remark that in this case the chunk of threads that are operating on the bitmap are all in the same multiprocessor accessing to the same, fast, shared memory.

There is a problem that it needs to handle up for implementing this strategy: managing the cache used to store intermediate intersection results. As in DCI [20][21] and the original SPAM, the candidates are examined in lexicographical order (see the description above), thus, using a *stack* we can delete the item when it is no more needed (see Implementation section).

The solution that we adopted in gpuSPAM is to assign a collection of candidates to the same thread block, by setting an independent cache for each block. Afterwards, we assign a contiguous sub-sequence of candidates to each chunk of threads increasing the cache reuse.

5.3 Transaction-wise parallelization

In this strategy the GPU cores, on one's own of the GPU multiprocessor they belong to, work on the same intersection or count operation. Each thread executes exactly the operations as described above for Candidate-wise approach; to put it better, all of them are in charge of an interleaved chunk of the bitmap, in such a way that threads having contiguous indexes are working on consecutive parts of the bitmap. Thus, the contiguous blocks processed by the same thread, are separated by a predeterminated sequence of $blocks \times threads$ decided at kernel launch time.

The chunk of threads that are involved in the operations on a bitmap are not in the same gpu-multiprocessor, so they do not have to access to the same shared memory as the previous approach. For this cause, the reduction has to be processed in two steps: after each thread has completed the count on its portion of bitmap, the counter of each thread is added on the local reduction, and then the partial sums for each multiprocessor are summed to obtain the global reduction.

To satisfy one of the conditions for having a good usage of the GPU (cores workload), the number of thread blocks must be at least the same as the number of GPU multiprocessors. After, to ensure that the cores of each multiprocessor are active, the global memory access should be overlapped with computation. Empirically, the amount of GPU's memory required by this approach is principally determinated by the size of the pruned dataset ($number\_of\_items \times size(bitmap)$) plus the size of the intermediate result cache ($max\_pattern\_length \times size(bitmap)$). If there is more space for more than one cache, we can use the previous strategy, which allows an higher utilization of the cores even when bitmaps are not huge.

5.4 Implementation

In this section is described the implementation of the strategies applied to SPAM. As said above, the given descriptions are more related to gpuDCI. In this work, they are slightly different, but still with the same principles. We will split each approach in two different parts: the intersections of the bitmaps and the re-use of the partial data during GPU computations.

5.4.1 Bitmaps Intersection and Counting

The SPAM application provides five different type of Bitmaps. To have an efficient computation from GPU side, as soon as we pruned the dataset after the $S-pruning$ and $I-pruning$ steps we start to use the GPU global memory avoiding data transfers. Indeed, this operation allowed us to hide the *memory latency* caused by the global memory. Exploiting the *shared memory* we drastically accelerate the performance of the whole application.

The work starts from *Counting Operation*, where the system is finding the *support* in number of customers. This means that the method Count is WALKING the memory counting the support going through the whole bitmap. Afterwards, the application creates the *SBitmap* which allows to accomplish the Candidate Generation performing the *s-step process* described above. Thus, for Candidate-wise parallelization method, we are going to execute the counting support in batches and fetch blocks of results at the end of the computation. The strategy uses a *cache* where we save the temporary values of the intersection in global memory (both reading and writing), and then the shared memory for performing the counting support. Basically, the cache is actually just a vector of integers (device vector) and its behavior is like a *stack* because when a candidate has been examined and it is no longer needed in the array we simply "popped" it out. In the actual version we are not performing the *pop* method, but we simply overwrite the previous element.

Transaction-wise parallelization method, instead of using shared memory for storing temporary results, is using global memory. But for boosting the computation and keeping the coalescing during the intersection, we needed to apply a *Reduction*.

5.4.2 ANDing and Counting operations

This section is the most important part for both Candidate and Transaction strategies. Indeed, as the pseudocodes show, they are quite similar among each other, but at the same time, they are using different parameters and arrays indexes.

The following pseudocode shows the Candidate wise approach:

Algorithm 5.1 Candidate wise

```
1: tid = threadIdx
2: __shared__ count[threadIdx] = 0
3: while tid ≤ bitmapSize do
4:     if (candidate not examined) then
5:         count[threadIdx] += (bitmap1[tid] & bitmap2[tid])
6:     else
7:         skip candidate
8:     end if
9:     i += blockDim
10: end while
11: result = Reduce()   ▷ Reduce the count and store the result in global memory
```

Down here there is shown the pseudocode of Transaction wise method:

Algorithm 5.2 Transaction wise

```
1: tid = threadIdx + blockDim * blockIdx
2: __shared__ count[threadIdx] = 0
3: while tid ≤ bitmapSize do
4:     if (candidate not examined) then
5:         count[threadIdx] += (bitmap1[tid] & bitmap2[tid])
6:     else
7:         skip candidate
8:     end if
9:     i += blockDim * gridDim[1]
10: end while
11: blockCount[blockIdx] = Shared_Reduce()[2]   ▷ Reduce the count and store the
    temporary result into the shared memory
12: // The last finished block loads temporary
13: // results from shared memory and reduce the count
14: // and store the result into the global memory
15: result = Global_Reduce()[3]
```

[1] where gridDim is the number of blocks in one axis

[2] Reduction process is applied to the shared variable

[3] Reduction applied to the counting device vector; It contains the final result of the counting operation. As said previously, it is storing data in global memory

5.5 Reduction

The aim of this section is to explain how to have an efficient data parallel reduction[1]. The reduction is a tree-based approach used within each thread block. As Figure 4.1 shows, each thread block reduces just a portion of the array; in this way, every multiprocessor of the GPU will keep busy allowing us to process very long arrays with the minimum effort.

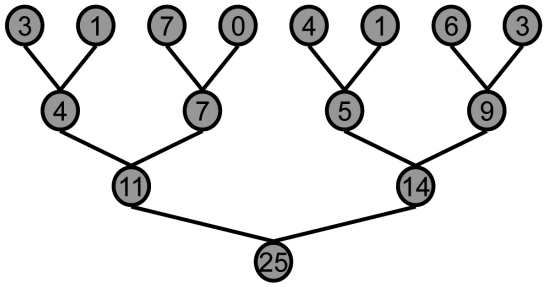

Figure 5.1: Tree-based approach for reduction.

The main problem is how we do communicate partial results between thread blocks. Unfortunately, CUDA doesn't provide a *Global Synchronization* because, according with [22], it is expensive to build in hardware for GPUs with high processor count and, very important, it would force the programmer to run fewer blocks to avoid deadlock, which may reduce overall efficiency. For such a reasons, we decompose the problem into multiple kernels.

Basically, the main goal of this optimization is because we are trying to reach the GPU peak performance. During our tests, we used *Bandwidth* as metric for memory-bound kernels. The hardware used for these tests is the $NVIDIA^{®}\ GTX\ 580$ which has a maximum Bandwidth peak of 192,4 GB/s. Of course we did not reach the maximum but we arrived quite close.

5.5.1 Parallel Reduction Complexity

The complexity of the reduction algorithm is quite simple to understand.

$\log(N)$ parallel steps, where each step S takes $N/2^S$ independently from the kind of operations. So far, we have that a *Step Complexity* is $O(\log(N))$.

But, for $N = 2^{D2}$, it performs $\sum_{S\in[1...D]} 2^{D-S} = N - 1$ operations. This formula produces a *Work Complexity* of $O(N)$ time - which is efficient but it does not perform more operations than a sequential algorithm.

With P[3] threads physically in parallel, the *Time Complexity* is $O(\frac{N}{P} + \log(N))$, where in a thread block, $N = P$, obtaining $O(\log(N))$ time. As Figure 4.2 shows the time complexity is better than the sequential one.

[1]This reduction is the general form that Nvidia provides to figure out the whole process and have the best performance. In gpuSPAM we used exactly the same form

[2]Where D is simply a number; more often is greater then S

[3]number of threads decided at kernel launching time; P should be as same number as multiprocessors inside the GPU

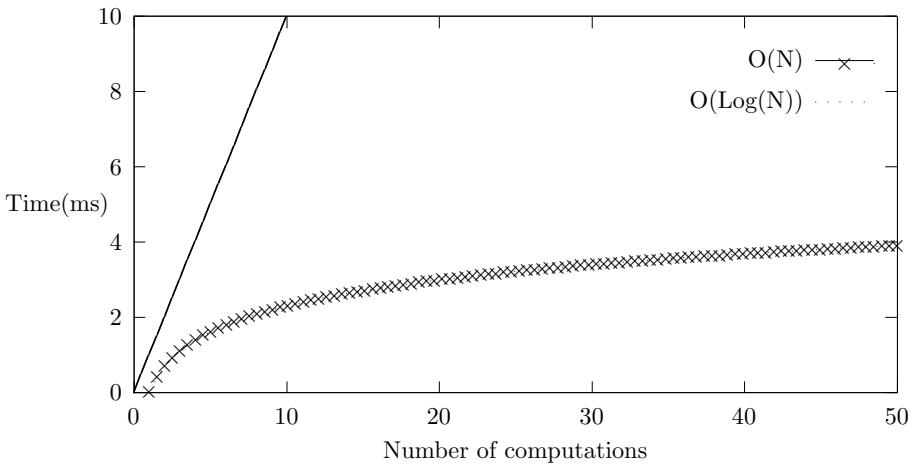

Figure 5.2: Difference between two complexity.

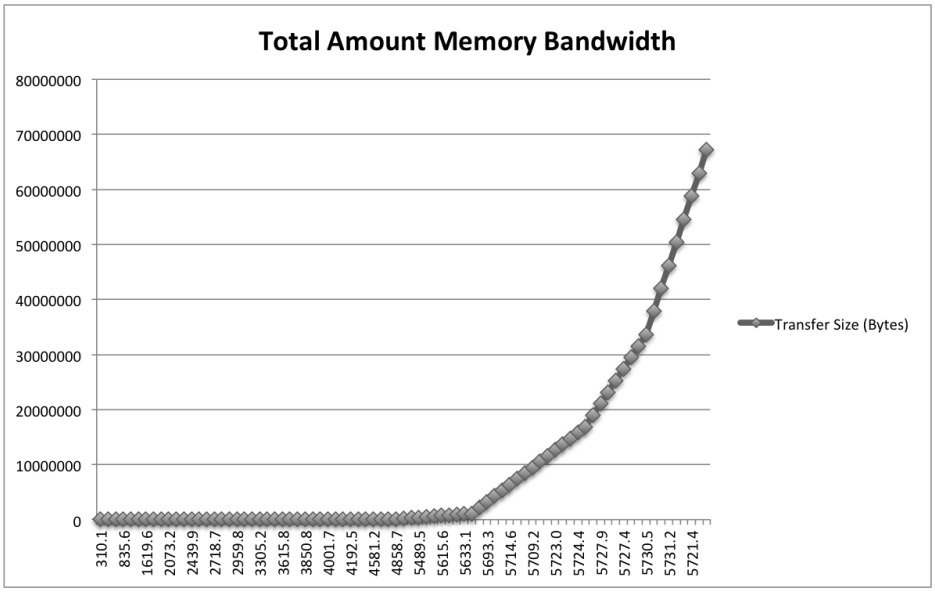

Figure 5.3: Amount of Bandwidth using reduction in the application calculated in MB/s.

5.6 The gpuSPAM Application

In this section we report how we modified the original Spam and how we adapted the GPU side into the program. Several problems we meet related with classes: for instance, we figured out that we could exploit the Abstract class *SeqBitmap* also from GPU side, because, as we reported above, after the second scan we move the dataset into the GPU global memory. In this way we allow the GPU to make every kind of calculation without wasting time for transferring data going and forth. In the beginning we started simply to make ANDing and Counting operation copying data in the GPU and moving the calculation back to CPU for performing the mining. But with this solution we realized that the whole program takes too much time for moving data between the host and the device.

So, basically, the application reads the dataset and save everything in memory RAM and then we copy the data into the GPU global memory. Afterwards we start to perform the mining operation. Several kernels are launched and all of them have the same amount of blocks and threads-per-block. We chose this configuration to allow the GPU to get the best performance and avoid to waste SM[4] resources (those thoughts are applied for both candidate and transaction wise parallelization).

5.6.1 Classes Diagrams

The next picture depicts the Classes Diagrams of both applications because we thought that it is necessary for the reader to understand the main differences between them.

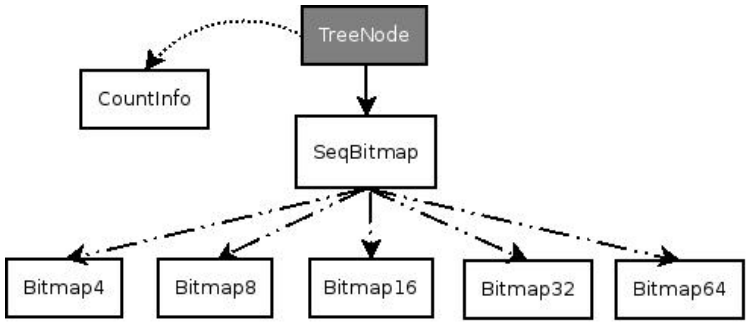

Figure 5.4: Classes diagram of SPAM

[4]The streaming multiprocessor (SM) contains 8 streaming processors (SP). These SMs only get one instruction at time which means that the 8 SPs all execute the same instruction. This is done through a warp (32 threads) where the 8 SPs spend 4 clock cycles executing a single instruction on multiple data (SIMD).

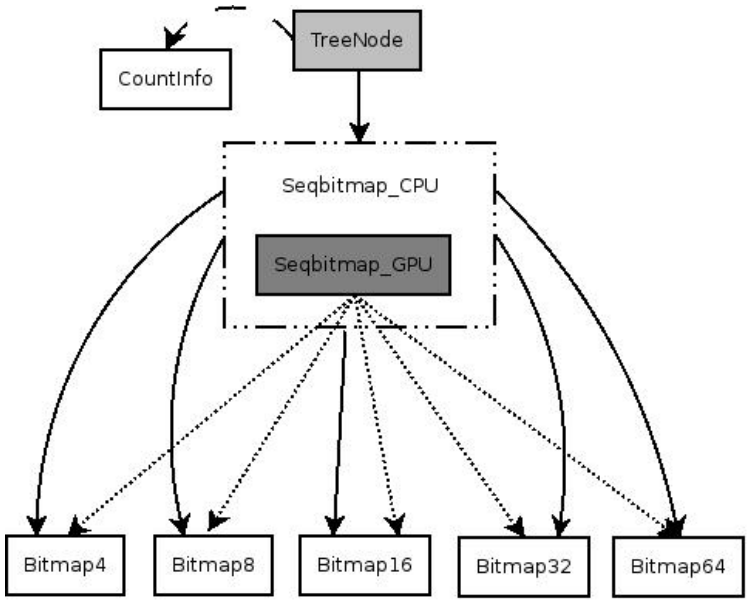

Figure 5.5: Classes diagram of gpuSPAM

As the reader can evaluate, we added a *sub-class* called *SeqBitmap_GPU* where the compiler nvcc of CUDA can understand what kind of method and private variables we can use directly into the GPU. We would remind that, source files for CUDA applications [23] consist of a mixture of conventional C++ "host" code, plus GPU "device" (i.e. GPU-) functions. The CUDA compilation trajectory separates the device functions from the host code, compiles the device functions using proprietary NVIDIA compilers/assemblers, compiles the host code using a general purpose C/C++ compiler that is available on the host platform, and afterwards embeds the compiled GPU functions as load images in the host object file. In the linking stage, specific CUDA runtime libraries are added for supporting remote SIMD procedure calling and for providing explicit GPU manipulation such as allocation of GPU memory buffers and host-GPU data transfer.

In the following code, we show how we implemented the new cuda-class. Basically it is a caveat because the cuda compiler cannot actually make any external linkage because it does not allow it[5], so we introduced a MACRO in the header file. This macro is a pre-processor definition where both compiler nvcc and the general C/C++ one, can use the methods of SeqBitmap class.

```
#ifdef __CUDACC__
#define CUDA_CALLABLE_MEMBER __host__ __device__
#else
#define CUDA_CALLABLE_MEMBER
#endif
```

Of course, `CUDA_ CALLABLE_ MEMBER` must put before the the implementation of the method.

[5]According with NVIDIA, the external linking will be a future feature

5.7 Data Flow Chart of Mining process

Here we report the Data Flow Chart of Mining method. Of course, it is not the full one since it contains some things that are not necessary to understand the process. Basically, the function repeat the following process several times, depending on the dataset. The Counting, ANDing and Reduction operations are already into GPU.

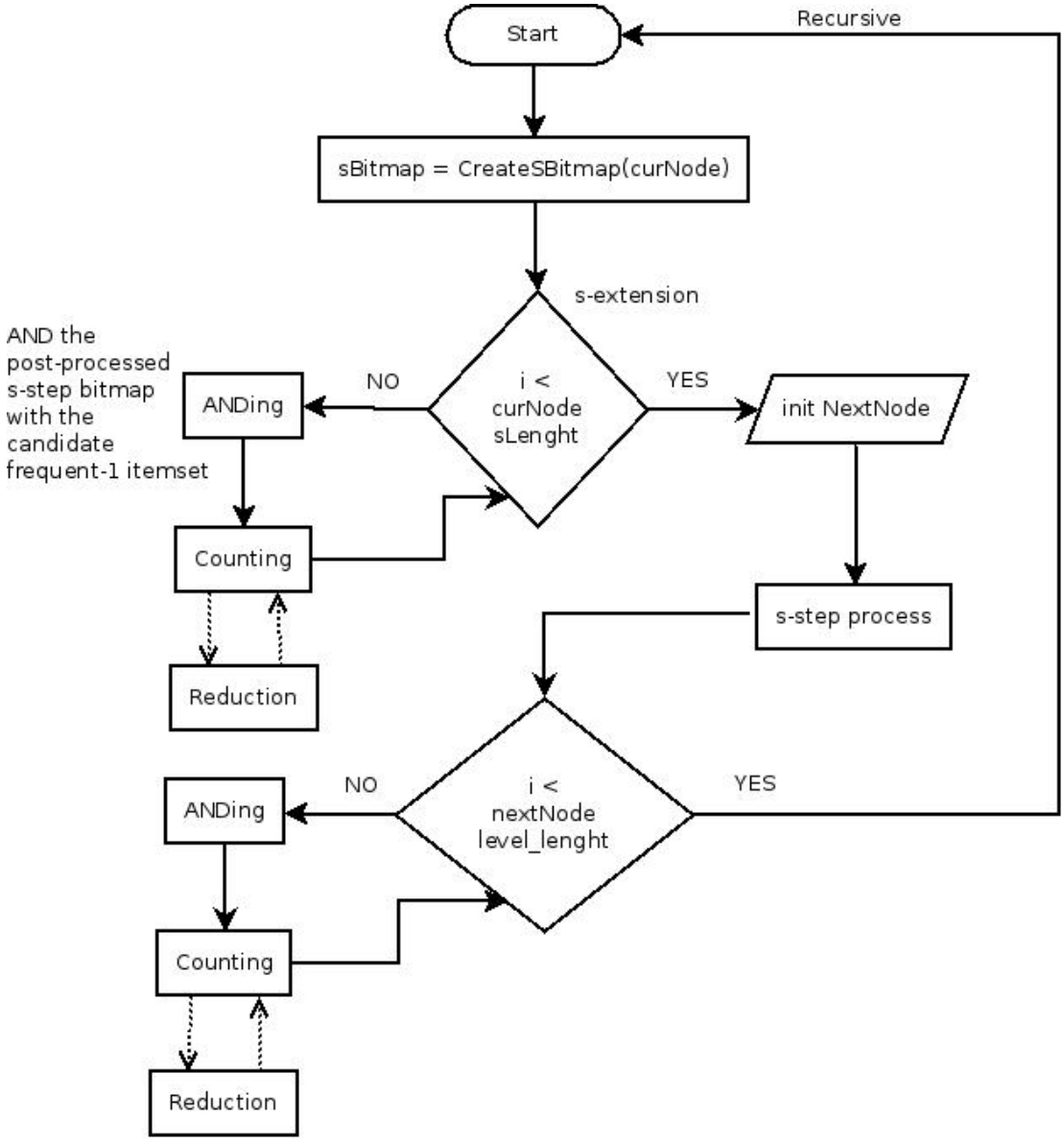

Figure 5.6: Data Flow Chart of Finding Sequential Patterns for both S-Step and I-Step

Reduction operation is not all the time called. That's because, Bitmap4 for example does not need it since the counting operation has just to compute a for loop with two iterations (see Results Chapter). So, if we decide to use the Reduction method, instead of having a speed-up we have as not as good performance without it. For this reason, in the data flow chart we use those kind of "arrows". However, for which kind of Bitmap has to be created is taken from *Seqbitmap_GPU* as the previous

Class Diagram shows.

5.8 Testing

In this section we present the main code that we used for testing the application. Nvidia provides directly some timers for CUDA. Indeed, thanks to those functions, we could mesueare with a high precision, the amount of time that each kernel was taking.
The *CUDA Event API* provides a collections of function which allows the programmers to calculate *events*, such as:

- mesuare the elapsed time for CUDA calls (kernel) with clock cycle precision
- query the status of an asynchronous CUDA call
- block CPU until CUDA calls prior to the event are completed

Mainly, these functions are used just for testing because according with the third point above here, the CPU is being stopped because CUDA calls about events. So, they must deleted before to release the code.

Here the general code for measuring the events:

```
double elapsedTime;          // varaible for elapsed time;
                             // double for having high float precision
cudaEvent_t start, stop;     // special variables for
                             // calculating the events

cudaEventCreate(&start);     // ceate an object for both start
cudaEventCreate(&stop);      // and stop event

// Recording the time from the very beginning when the kernel
// has been called to when it stops as well

    cudaEventRecord(start, 0);
        Kernel<<<grid, block>>>(...);
    cudaEventRecord(stop, 0);

// Synchronize the record event
cudaEventSynchronize(stop);

// Calculate the elapsed time
cudaEventElapsedTime(&elapsedTime, start, stop);

// Release memory
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

With those simple methods, we could simply calculate the amount of time needed.

Instead, as the reader will evaluate from the next chapter, we had to calculate even the time taken from CPU. For affording this, we did not use a similar approach as for the GPU, because we would not useneither the *clock()* method nor *difftime()* of C++ library. Indeed, the first one return the clock ticks elapsed since the program was launched, then, the other one, calculate the elapsed time from two different times (as *cudaEventRecord* does) but some issues were encountered becuase most of them are in *nsec*.

So, we decided to use the Visual Studio Performance Profiling Tools, where we could see the exactely time tike by every single function and many informations. Honestly, in the very beginning of the project we used this powerful tools for understanding which were the methods that were taking the biggest amount of time.

Chapter 6

Experimental Results

"People think that computer science is the
art of geniuses but the actual reality is
the opposite, just many people doing things that
build on each other, like a wall of mini stones. "

Donald Knuth

In the following part of the section, we describe the behavior exhibited by gpuSPAM in our experiments. We launched the application with different datasets and using different parameters for the kernels. The main goal of these tests, is to understand how GPUs can really improve the speed up of the computations. We split the results in two parts: one for the candidate-wise parallelization and the other one for the transaction-wise parallelization. In the next section we describe the test environment and the datasets.

6.0.1 Test environment and datasets

The experiments where executed on a workstation equipped with an Intel Xeon Quad Core CPU @ 2.67GHz, 8 GB of RAM, 64-bit architecture and a NVIDIA GTX 580.
GPU features: 16 multiprocessors (512 cores) GPU Processor clock @ 1544MHz, 1.5GB device memory and Cuda device capability 2.0 Fermi architecture. The operating system was *Windows* 7 with *Visual Studio* 2010 as IDE and as GPU compiler we used *nvcc* which it is part of the Cuda Toolkit 4.1. During debugging tests we used the *NVIDIA Parallel Sight* 2.1 for GPU side, then we used the Debugger of Visual Studio for CPU side.

In our experiments we generated numerous synthetic datasets using the IBM AssoGen program [15].

The amount of block for each operation was 256 and 256 threads per block. Actually we were giving a small part of the maximum capacity.

6.0.2 Synthetic data generation

The table in the bottom represents which kind of datasets we used for our tests. The reader needs to pay attention on the number of customers, because the application can support maximum 64 transactions for each customer.

Tot. # Transactions	Minimun support
10K	0.5
50K	
100K	
200K	
400K	
800k	
1M	
5M	
20M	

Table 6.1: Summary of used datasets

6.0.3 Graphs of results

In the next pages, the reader can see how much time gpuSPAM takes[1] for computing a certain amount of Transactions. In [14] there are the results[2] of the original application. Each function of the original application has been calculated again because we used a different workstation as the reader can notice from the following table. The reference to Result Section of SPAM above is cited just becuase it can be useful for the reader for having a complete landscape of the results.

Operation	GPU CW Time	GPU TW Time	CPU Time
ANDing	$\backsim 182\,ns$	$\backsim 323\,ns$	$\backsim 477\,ns$
Counting	$\backsim 543\,ns$	$\backsim 587\,ns$	$\backsim 788\,ns$
CreateSBitmap	$\backsim 643\,ns$	$\backsim 678\,ns$	$\backsim 894\,ns$

Table 6.2: Time taken for all operations of Bitmap 32

The values in the table have been calculated doing an average operation each time we launched the application. Basically, we divided the total amount of the time taken of the operation by the total number that it is called. [3] In every test, we used

[1] Those results are obtained calculating the time that every kernel operation takes. Then the time taken, has been divided by the number of blocks per threads

[2] See results chapter in the article

[3] We repeated this calculation for ANDing, Counting and CreateSBitmap - The results are related to the GPU Kernel

the same chunk of blocks and threads-per-block. We used the Bitmap 32 because we could effectively see the difference with the CPU and even becuase it is one of the most used with large datasets.

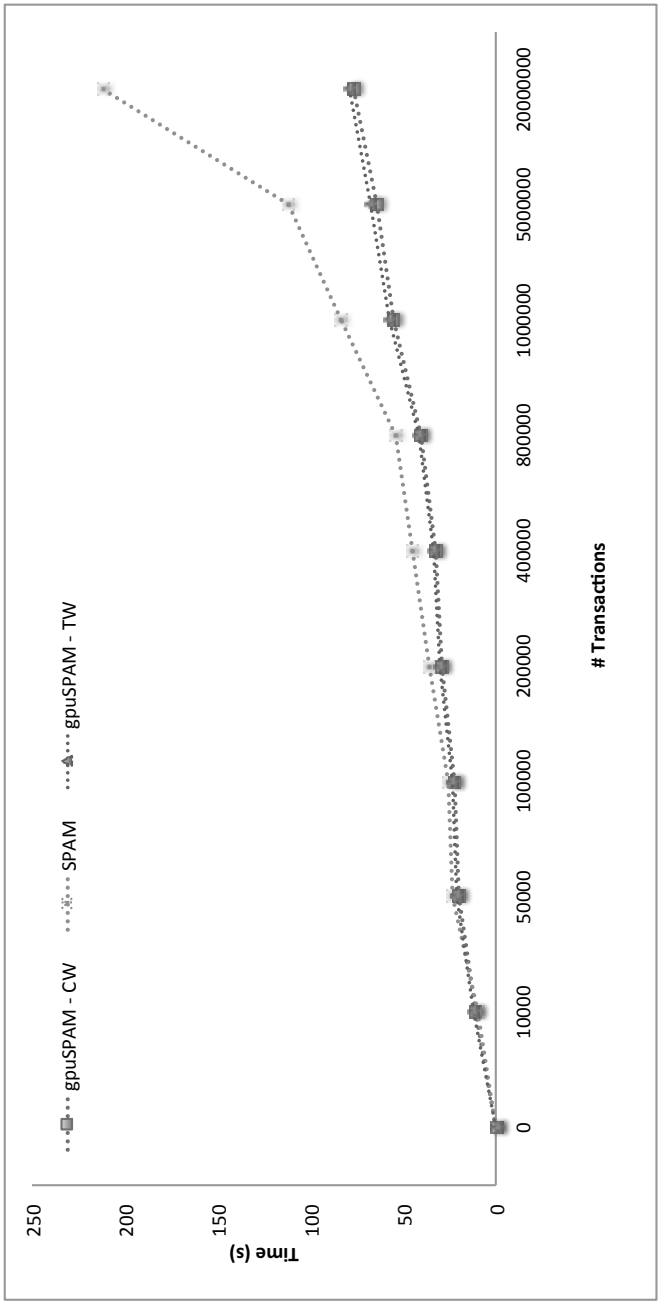

Figure 6.1: Time taken comparing the two applications

As the reader can evaluate, GPU can improve the program especially when we are going to compute a huge amount of transactions. This happens because SPAM is very optimized for CPU computation, but when it is performing a very big dataset, especially when Bitmap 32 and 64 operations are called, the GPU takes advantages of parallelism and reductions as well. Of course, also with the other kind of Bitmaps the device is performing them in parallel but they do not take so much lesser time

than the original ones.

In the following table, we show the time taken for all operations of each Bitmap. The example is based on the dataset with 1 milion transactions and using Candidate wise:

Operation	Bitmap4	Bitmap8	Bitmap16	Bitmap32	Bitmap64
ANDing CPU	18 ns	32 ns	147 ns	477 ns	798 ns
ANDing GPU	23 ns	39 ns	149 ns	182 ns	202 ns
Counting CPU	64 ns	78 ns	188 ns	788 ns	982 ns
Counting GPU	71 ns	88 ns	183 ns	543 ns	610 ns
CreateSBitmap CPU	58 ns	61 ns	173 ns	894 ns	1.33 µs
CreateSBitmap GPU	62 ns	65 ns	177 ns	643 ns	953 ns

Table 6.3: Time taken for all operations of each Bitmap

Truth be told, with the Bitmaps 4 and 8, GPU takes more time to make the computations. That's because we are moving data into shared memory which of course need some time. But as we said above, with Bitmaps 32 and 64 we have a good improvement.
Figure 6.2 is referred to the table above.

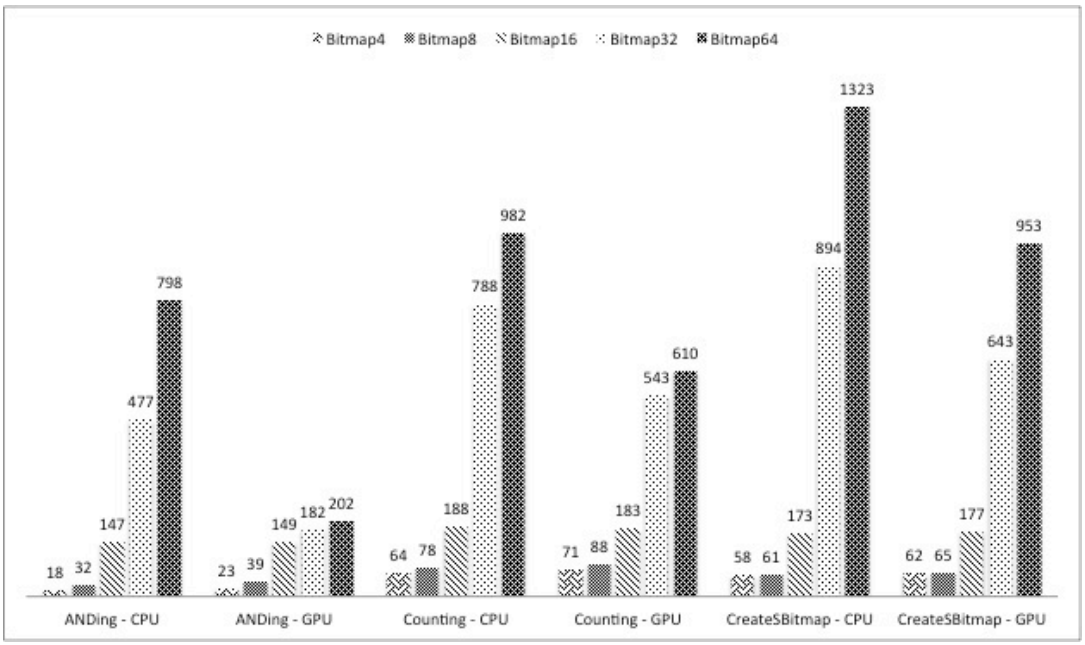

Figure 6.2: Comparation between operations and Bitmaps

Chapter 7

Conclusions

"Stay Hungry, Stay Foolish"

Steve Jobs
Graduate speech at Stanford University (2005)

The analysis that has been proposed about this parallel data mining algorithm on graphic processor, aims to demonstrate that GPUs can compute very large datasets reducing drastically the time's taken from CPU. Indeed, after the evaluation of the results obtained calculating the amount of time that each operation needs to be computed, confirm that GPUs can be exploited for computing a huge amount of data.
Referring to the introduction where we explained how SPAM works, the example related to the Supermarket, we can think now gpuSPAM as a "Super Fidelity Card" where even the customer is able to know which are the most bought items. This example helps the reader to have an idea about this work. Indeed, if just the supermarket checks which are the most bought items of each customers, it takes more time rather than every customer checks them by itself. This is because, gpuSPAM works in parallel and not sequentially.

7.0.4 Future works

This thesis gives just the first step for a better implementation of the original application. Indeed, we just convert the functions in GPU and analyzed the performance for each operation. So, basically it provides an idea of the potential of graphic cards. In the future it will be possible to use this work and improve the code that we have written so far. In this way people can create a brand new algorithm (maybe using both CPU and GPU at the same time) for improving it. Another possible feature could be an algorithm which exploits also multi-CPU and multi-GPU.

Bibliography

[1] D. Hand, H. Mannila, and P. Smyth, *Principles of Data Mining*. The MIT Press, 2001.

[2] M. D. Dikaiakos, D. Talia, and A. Bilas, *Knowledge of Data Mining in GRIDs*. Springer Science+Business Media, LLC, 2007.

[3] D. B. Kirk and W. mei W. Hwu, *Programming Massively Parallel Processors - A Hands-on Approach*. Morgan Kaufmann, 2010.

[4] M. J. Atallah and M. Blanton, *Algorithms and theory of computation handbook*. CRC Applied Algorithms and Data Structures series, Chapman Hall, 1998.

[5] N. Corporation, *CUDA C Programming Guide*. Santa Clara, CA 95050, 4.2 ed., May 2012.

[6] N. Corporation, *Fermi Compute Architecture Whitepaper*. Santa Clara, CA 95050, 1.1 ed., January 2009.

[7] J. D. Ullman, "http://infolab.stanford.edu/ ullman/mining/assocrules.pdf."

[8] C. Silverstein, S. Brin, R. Motwani, and J. Ullman, "Scalable techniques for mining causal structures," *VLDB Journal Articles*, 1998.

[9] C. Lucchese, *High Performance Closed Frequent Itemsets Mining inspired by Emerging Computer Architectures*. PhD thesis, University Ca' Foscari of Venice, 2008.

[10] S. Orlando, "http://www.dais.unive.it/ dm/new_slides/3_asso_dwm.pdf."

[11] W. Fang, K. K. Lau, M. Lu, X. Xiao, C. K. Lam, P. Y. Yang, B. He, Q. Luo, P. V. Sander, and K. Yang, "Parallel data mining on graphic processors," *HKUST-CS08-07*, October 2008.

[12] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," *Proceedings of the 20th VLDB Conference Santiago*, 1994.

[13] C. Silvestri, *Distributed and Stream Data Mining Algorithms for Frequent Pattern Discovery*. PhD thesis, University Ca' Foscari of Venice, Dipartimento di Informatica Universita' Ca' Foscari di Venezia, Via Torino, 155 30172 Venezia Mestre – Italia, 2006.

[14] J. Ayres, J. Gehrke, T. Yiu, and J. Flannick, "Sequential pattern mining using a bitmap representation," *ACM 1-58113-567-X/02/0005*, 2002.

[15] R. Agrawal and R. Srikant, "Mining sequential patterns," *ICDE '95: Proceedings of the Eleventh International Conference on Data Engineering*, vol. IEEE Computer Society, March 1995.

[16] R. Srikant and R. Agrawal, "Mining sequential patterns: Generalizations and performance improvents," *EDBT '96: Proceedings of the 5th International Conference on Extending Database Technology: Advances in Database Technology*, vol. Springer-Verlag, March 1996.

[17] R. J. Bayardo, "Efficiently mining long patterns from databases," *SIGMOD 1998*, pp. 85–93, 1998.

[18] D. Burdik, M. Climlim, and J. Gehrke, "Mafia: A maximal frequent itemset algorithm for transactional databases," *Proceedings of the 17th International Conference on Data Engineering*, April 2001.

[19] C. Silvestri and S. Orlando, "gpudci: Exploiting gpus in frequent itemset mining," *20th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, 2012.

[20] C. Lucchese, S. Orlando, and R. Perego, "kdci: a multi-strategy algorithm for mining frequent sets," *FIMI Workshop*, 2003.

[21] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri, "Adaptive and resource-aware mining of frequent sets," *IEEE ICDM*, pp. 338 – 345, 2002.

[22] M. Harriss, "Optimizing parallel reduction in cuda," tech. rep., NVIDIA Developer Technology, 2010.

[23] NVIDIA, "http://moss.csc.ncsu.edu/ mueller/cluster/nvidia/2.0/nvcc$_2$.0.*pdf*."