Parallel Data Mining Algorithms: Frequent Pattern and Classification Problems

Dardan Xhymshtiti
University of Colorado Denver
Department of Computer Science
Denver, Colorado
dardan.xhymshtiti@ucdenver.edu

Taulant Ymeri
University of Colorado Denver
Department of Computer Science
Denver, Colorado
taulant.ymeri@ucdenver.edu

Abstract—Data Mining and Machine Learning algorithms, are very useful nowadays in extracting insights from data. In this project report, we evaluated the possibility of parallelizing Naïve Bayes classification algorithm, and Apriori technique in GPU. Naïve Bayes is a supervised classification algorithm used for classifying data records to one or another class. On the other hand, Apriori is a Frequent Pattern Mining technique used to find and extract the frequent patterns from a transactional dataset. Both algorithms find a wide area of usage, starting from classification of customers whether they will by a specific product or not based in the purchase history to the finding which items in a store are mostly sold together. The evaluation of these algorithms showed an increase of the execution time of Apriori, and an improvement of Naïve Bayes.

Keywords—Frequent Pattern Mining, Classification, CUDA

I. INTRODUCTION

As the use of technology is spread in every possible field, the amount of data generated is growing rapidly. Starting from the daily transactions processed in big markets, spatial data generated by GPS devices and marketing data on the internet, to the smallest transactions processed in big markets, spatial data generated by GPS devices and marketing data on the internet, we need hardware and software support to better store, manage and benefit from this data. Considering the hardware support, nowadays exist hard drives with capacity over 10Tb which enable storing large amount of data, SSD which provide a faster read/write access to the stored data, till data centers which provide a large capacity of space for the users to store data over the internet. The data collected in these storages is usually raw data, and to extract knowledge, we need algorithms that are capable of learning and applying new processes without being explicitly programmed for all of them. These algorithms are called learning algorithms.

A machine learning algorithm that we will leverage in this paper, is Naïve Bayes. Naïve Bayes is a classification algorithm which applies Bayes formula to calculate the probabilities of specific attribute values being in a specific class, and later use these probabilities to predict the classes for the incoming tuples. This algorithm is a supervised one, which means the training data consist of labels that tell in which class each item belongs to.

The second algorithm leveraged is Apriori. Apriori is a data mining technique used in transactional datasets to extract the most frequent patterns. Apriori is not a learning algorithm (i.e., cannot adapt itself in the data not seen before).

Our main idea in this project is to parallelize these two algorithms using GPU as a running platform. We are motivated by the amount of data being increased day by day, and the need for high-performance methods and techniques in terms of speed and accuracy, that faster and better extract knowledge from data. We have run our algorithms both in Hydra cluster and in our laptops, and the results seems to be promising.

The outline of this paper is as in follow: in section 2 we will discuss about the related work in this field; in section 3 a parallel implementation of Naïve Bayes is discussed; in section 4 a parallel implementation of Apriori is discussed; in chapter 5 we present the results from our evaluation; in step 6 we present the future work and the optimizations that can be done on top of the algorithms implemented.

II. RELATED WORK

Most of parallel algorithms developed for FPM, follow an Apriori approach. Such an instance is GPApriori[2] where the candidate generation step is performed in CPU, whereas the support counting step is performed in GPU. The advantages of using Apriori is that it is easy to understand, and easy to apply in transaction data. The drawbacks of using this algorithm are that is slow because the candidate generation step is very time consuming, and for each candidate generated we have to scan the data set to determine the support count of that candidate. Other implementations like FP-Growth[2] algorithm, uses FP-Tree to make the frequent pattern mining faster and easier. An FP-Tree consist of unique items as nodes together with their support counts as attribute for every node. Algorithms using this approach work better than Apriori in large datasets, because they store less information in the memory, and the extraction of frequent patterns is faster because we scan the data set only once to map the items in the tree. The downside of this approach is that its nature it is not efficient to be implemented on GPU. That because, FP-Tree is created recursively and later we need to perform tree traversals while extracting the frequent patterns.
These properties of FP-Trees, do not fit well with the underlying architecture of the GPU. Later algorithms like CGMM[3] combines features from Apriori, FP-Growth, to create an algorithm which performs both the candidate generation and support counting step in GPU. This algorithm uses a heuristic approach which during the runtime it knows whether to use CPU or GPU to mine frequent patterns, based on the density of the data.

For ADC (Automatic Document Classification) there exist numerous implementations of different parallel algorithms. Starting from one of the simplest one which is the K-Nearest Neighbor algorithm which considers each test document \( x \) as an independent instance and uses \( k \) training documents closer to \( x \) (the neighborhood) to classify it [4]. Other approaches for this kind of problem which are dimensionality reduction by features selection[5], whose goal is to keep only the features that better “define” the documents, and data indexing [6], [7], whose goal is to represent the data in a more direct and efficient way.

The authors of [8], present a study about how several data mining techniques may be implemented using GPUs, more specifically the CUDA architecture, to classify documents. Also, one interesting study regarding ADC is the paper [9], which presents a parallel implementation in CUDA for meta-learning approach Random Forest, with a significant reduction in execution time. Another algorithm is the parallel version of SVM[10] where the authors introduce several techniques to accelerate SVM in GPUs, including spare matrix format to enhance performance.

### III. APRIORI ALGORITHM

Say that we have a transactional dataset as the one below.

<table>
<thead>
<tr>
<th>Table 1. Transactional dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction ID</td>
</tr>
<tr>
<td>T1</td>
</tr>
<tr>
<td>T2</td>
</tr>
<tr>
<td>T3</td>
</tr>
<tr>
<td>T4</td>
</tr>
</tbody>
</table>

Finding frequent patterns means finding the patterns of items that are the most represented in the dataset. From the table above a single item e.g., (1) we call 1-item-set, a set of two items we call 2-item-set e.g., (1,3), and a set of \( n \) items we call \( n \)-items-set. The above table shows a horizontal representation of the data where in the ID column we store the IDs of the transactions whereas in the Items column we store the IDs of the items bought at a specific transaction. In the Apriori algorithm we usually decide a threshold value called \( \text{min\_sup} \) which represents a critical point where a transaction is with interest to us or not. In the first step of Apriori algorithm, we scan the dataset, extract all the unique items, and count how many times they are presented in the dataset. By doing that we got the following table:

<table>
<thead>
<tr>
<th>Table 2. Support counts for each item</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

Next, we check if the support counts pass the \( \text{min\_sup} \) which is a user-defined variable. Let us say that the \( \text{min\_sup} = 2 \). In this case, all the support counts in the table above pass \( \text{min\_sup} \), and all the corresponding 1-item-set are considered candidates for the next step.

The next step is candidate generation step. In this step, we combine the items with each other to generate 2-items-sets from 1-items-sets. By doing so, we get the following table:

<table>
<thead>
<tr>
<th>Table 3. 2-items-set candidates with corresponding support counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>1,2</td>
</tr>
<tr>
<td>1,3</td>
</tr>
<tr>
<td>1,4</td>
</tr>
<tr>
<td>1,5</td>
</tr>
<tr>
<td>2,3</td>
</tr>
<tr>
<td>2,4</td>
</tr>
<tr>
<td>2,5</td>
</tr>
<tr>
<td>3,4</td>
</tr>
<tr>
<td>3,5</td>
</tr>
</tbody>
</table>

Again, we check if the supports of these 2-items-sets pass the \( \text{min\_sup} \) or not. The 2-items-sets that pass the \( \text{min\_sup} \) are considered candidates for the next 3-items-sets, whereas those who do not are pruned. Without any doubt, we can perform pruning in those candidates, because we follow the Apriori property which states: all the sub-items-sets of an items-set are frequent if the items-set is frequent. Starting from the candidate generation from the 2-items-set to 3-items-set, we need to be careful in applying the Apriori property. Now we cannot combine all the items with each other as we did in the first step. Now we combine only the items that have the same suffix of items. For example, the 2-items-set (1,2) can be combined with (1,3) because the suffix (1) is similar, whereas (1,2) cannot be combined with (2,3) because they do not have a similar suffix (i.e., (1) is different from (2)). By pruning the items with less minimum support than 2 and performing the candidate generation step we get the following table:

<table>
<thead>
<tr>
<th>Table 4. Frequent patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>2,3</td>
</tr>
<tr>
<td>2,3,5</td>
</tr>
</tbody>
</table>

By performing pruning, we can see that the only remaining pattern is (2,3,5) represented 2 times in the dataset in transactions T2 and T3. This pattern for us is the most frequent pattern in the dataset. As we saw from the example above, the
Apriori uses an iterative approach of candidate generation and support counting which leads to the frequent patterns.

IV.I VERTICAL REPRESENTATION OF THE DATASET
As we discussed before, Apriori algorithm has a weak performance for two main reasons:

1. The candidate generation step is very computation intensive,
2. For every candidate generated we must scan the dataset to find its support count.

There is a quick way how we can skip the support counting step. That is by storing the data in vertical representation rather than in horizontal one. The table below shows a vertical representation of the table above:

<table>
<thead>
<tr>
<th>Items ID</th>
<th>Tid_List</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T1, T3</td>
</tr>
<tr>
<td>2</td>
<td>T2, T3, T4</td>
</tr>
<tr>
<td>3</td>
<td>T1, T2, T3</td>
</tr>
<tr>
<td>4</td>
<td>T1</td>
</tr>
<tr>
<td>5</td>
<td>T2, T3, T4</td>
</tr>
</tbody>
</table>

In this representation, we store the IDs of items in the left side, and in the right side we store the transactions where this item find itself in. The way how the algorithm works is that we simply intersect the Tid_Lists of the items combined. For example, if we have a candidate like (1, 2) we intersect Tid_lists of items with id 1 and 2:

\[(T1, T3) \cap (T2, T3, T4) = (T3)\]

By doing this, we know that transaction items (1,2) are purchased together in the transaction T3. We do not need to scan the database for the support counting step. We know that items 1 and 2 are purchased only once. Next, we check the \( \min_{sup} \), we prune the candidates that do not exceed this value, generate new candidates in the same way we did in the example in section 4, until we find the frequent patterns. More on this you can find at Apriori[].

IV.II BIT VECTOR REPRESENTATION OF TID_LIST
It is true that vertical representation of the data improves the performance of sequential Apriori algorithm. But in case that we want to implement Apriori in GPU, there is a problem if we want to use the vertical representation. That is, the intersection of two tid_lists. Let us first consider how we can implement intersection in sequential:

- Say that we have \( A = \{a_1, a_2, ..., a_n\} \) and \( B = \{b_1, b_2, ..., b_m\} \). One way to implement the intersection in sequential is to represent \( A \) and \( B \) as vectors, take \( a_i \) and scan through \( b_1, b_2, ..., b_m \) to check if we find any watch (i.e., \( a_i = b_j \)). If we find one, we add that one in the result vector otherwise we do nothing. The worst-case execution time of this algorithm is \( O(n) \) because for every element \( a_i \) of \( A \) where \( 0 \leq i \leq n \) we must scan \( b_1, b_2, ..., b_m \) in \( B \) until \( b_k = a_i \). The worst case is where we cannot find any match for the specific element \( a_i \). This approach is time consuming both in CPU and GPU.

- The other way is to represent \( A \) and \( B \) as maps. Thus, if we want to find a match of \( a_i \) in \( B \)’s elements, we need only \( O(1) \) time. Even though this approach gives good performance in CPU, we cannot use it in GPU because GPU support only arrays as data structures and not maps.

- The third approach is to represent Tid_list as a bit vector. Say for example that we have the Tid_list (T2, T3, T4). We can represent it as:

\[
\begin{bmatrix}
0 & 1 & 2 & 3 & 4 \\
0 & 0 & 1 & 1 & 1 \\
\end{bmatrix}
\]

Figure 1. Bit vector representation of tid_list

That is, we create a vector of 0 values of the size equal to the \( \max \) id in the tid_list (which is 4 in our example), and we place 1 in every cell where the corresponding index exist in the tid_list. The dataset table now becomes like:

Table 6. Bit vector vertical representation of transactions

<table>
<thead>
<tr>
<th>Items ID</th>
<th>Tid_List</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0101</td>
</tr>
<tr>
<td>2</td>
<td>00111</td>
</tr>
<tr>
<td>3</td>
<td>0111</td>
</tr>
<tr>
<td>4</td>
<td>01</td>
</tr>
<tr>
<td>5</td>
<td>00111</td>
</tr>
</tbody>
</table>

Now, the intersection of two bit arrays \( A_{bit} \) and \( B_{bit} \) works such that for every \( i \), where \( 0 \leq i \leq \min(n,m) \), we perform \( a_i \land b_i \). For example, the intersection of (T1, T3) with (T2, T3, T4) is converted to the intersection of:

\[
A_{bit}: \quad \begin{bmatrix}
0 & 1 & 2 & 3 \\
\downarrow & \downarrow & \downarrow & \downarrow \\
0 & 1 & 2 & 3 \\
\end{bmatrix}
\]

\[
B_{bit}: \quad \begin{bmatrix}
0 & 1 & 1 & 1 & 0 \\
\downarrow & \downarrow & \downarrow & \downarrow \\
0 & 1 & 2 & 3 \\
\end{bmatrix}
\]

\[
A_{bit} \land B_{bit}: \quad \begin{bmatrix}
0 & 1 & 1 & 1 \\
\downarrow & \downarrow & \downarrow & \downarrow \\
0 & 1 & 2 & 3 \\
\end{bmatrix}
\]

Figure 2. Intersection between two bit vectors

As we can see from the figure above, we discarded using the bit value with index 4 at vector \( B_{bit} \), because it does not have a similar representative at vector \( A_{bit} \).

Representing tid lists as bit vectors, makes the intersection process easier. All what we need to do is to perform OR binary operation and get the result. This method fits well with the underlying vectorize architecture of GPU but it also introduces some other problems. While we perform the intersection faster, we allocate more space in memory to store an array. Say, for example, we have a tid_list like (T1, T2, T3). A bit vector representation of this tid_list is \( (0,1,0,0,0,0,0,0,0,1) \). As we can see the vector is very sparse, and much of the memory used to store this bit vector is wasted. In this case, we did a tradeoff between speed of performance and size of memory allocated. We can solve the sparse vector problem by maintaining a linkedlist, but linked lists are not supported on GPU. In the
sequential and parallel implementation of Apriori algorithm, I follow this approach of presenting tid_lists as bit vectors, where every bit that is set (i.e., bits that are 1), tells that the corresponding index is presented in tid_list.

IV. SEQUENTIAL IMPLEMENTATION OF APRIORI
Sequential implementation of Apriori algorithm, follows the bit vector representation of tid_list approach that we discussed in the section IV.III. Using this approach in sequential implementation, causes the algorithm to run slower and allocate more space in memory. Even though there are other ways how we can make the sequential algorithm run faster (e.g., by representing tid_lists as maps as discussed before), we followed the bit vector representation because we wanted to reuse much of the code that we write in sequential in our parallel implementation. Although the algorithm might run slower in sequential, the design of data structures used, fits very well and provides a greater performance when implemented Apriori in GPU.

```
procedure GENERATECANDIDATES(k-items-candidates)
  1. for i = 0 to candidates.length do
  2.    for j = i+1 to candidates.length do
  3.      if cand[i] and cand[j] have same n-1 ele. then
  4.        cand[i].append(cand[j][n])
  5.        intersect cand[i].tid_list with cand[j].tid_list
  6.    end if
  7.  end for
  8. end for
  9. return (k+1 items candidates)
```

Figure 3. Procedure for generating candidates

Algorithm Sequential_Apriori

Input: dataset D, min_sup
Output: frequent pattern list L

1. candidates = 1-item-set candidates
2. while L > 0 do
3.   GENERATECANDIDATES(candidates)
4.   if candidate_sup_count > min_sup then
5.      Consider candidate for the next step
6. else
7.   prune candidate
8. end
9. end while
10. return (L)

Figure 4. Sequential Apriori algorithm

We take as input a itemset databases and minimum support value. In the step 1 of the algorithm in figure 2, we scan through the database, find all the distinct items and count how many times they present there. These values we save in a map called candidates and we consider them in the next generation of 2-items-sets. In the step 3 we use the procedure from fig.1 to generate k+1-items-set frequent patterns, in step 4 we check if their frequency count passes the minimum support or not. In case that a pattern support count passes the minimum support, we consider it as a candidate for the next step, otherwise we prune it. We continue like this until we find all the frequent pattern, and return them as result.

V. PARALLEL IMPLEMENTATION OF APRIORI
Apriori presents one of the basic ways of finding frequent patterns in a transactional dataset. Even though its idea is simple, parallelizing it in GPU is not so easy. From [ ] we know that GPU has an SIMT architecture, and for a problem to benefit by the speed of GPU, it should have some behaviors that are suitable with the architecture of GPU. For example, for a problem that performs a single function many times over different data, we can easily run it faster by implementing in GPU. But for a problem that has a sequential nature, we can increase its execution time rather than decrease when implementing in parallel. Implementing Apriori in GPU, can easily fall in the second case, where GPU implementation can show weaker performance than a CPU one.

In our implementation of Apriori, CPU and GPU have different tasks. We use CPU for generating candidates and pruning whereas GPU to perform support counting. In the following subsections, we will discuss about challenges during the implementation and algorithm description.

VI. II CHALLENGES DURING IMPLEMENTATION
Parallel implementation of Apriori was a challenge on itself. We have categorized these challenges as in follow:

1. **Problem reformulation**
   Before starting the implementation, the challenge was to reformulate Apriori algorithm in terms of vectors. Because GPU consist of a vectorizer architecture, to utilize the resources at its best, we had to perform this reformulation. Turning Apriori to a vector problem, we had to manipulate a lot with indices and array sizes.

2. **Array data structure**
   To implement Apriori we used CUDA programming model from Nvidia. CUDA does not support any other data structures except arrays. The problem with arrays is that they have static sizes, and you must know exactly how much memory to allocate in GPU for the program to run correctly.

3. **Thread and memory management**
   Determining the number of blocks and threads was another challenge that we faced during the implementation phase. Launching less threads than needed, often resulted in bad outcomes. Another one is that the time did not promise us to make the algorithm more memory-efficient, so Apriori that we implemented transfers a lot of data between host and device memories, what makes it run slower in some cases.
V. II REFORMULATING APRIORI IN TERMS OF VECTORS

As we know, Apriori uses an iterative approach. In the first iteration, it starts by extracting all the unique items (1-item-set) in a dataset, and for each of them, finds the transactions that it appears in. The number of transactions represents its support count. Then, if this support count is greater than a defined minimum support threshold, we consider that specific candidate for the next round of candidate generation. In the second round, we use (1-item-set) items for example {A, B, C} to generate 2-item-set candidates {AB, AC, BC} and continue the same process.

In our case candidate is performed in CPU, but the intersection of bit vector tid_list of candidates and support counting is performed on GPU. To better capture how we proceeded with our implementation, we will provide an example. Say that we have the dataset the following:

<table>
<thead>
<tr>
<th>Items ID</th>
<th>Tid_List</th>
<th>Support Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0101</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>00111</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0111</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>00111</td>
<td></td>
</tr>
</tbody>
</table>

The above table represents 1-item-set candidates. We use CPU to parse the data and to bring the data to that position. We represent the above table by using a map where Items ID are considered as key whereas Tid_List are considered as values. On the next step, we use CPU to perform candidate generation. After CPU generates the 2-item-set candidates the map looks like:

<table>
<thead>
<tr>
<th>Items-set</th>
<th>Tid_List</th>
<th>Support Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,2</td>
<td>waiting_for_gpu</td>
<td>waiting_for_gpu</td>
</tr>
<tr>
<td>1,3</td>
<td>waiting_for_gpu</td>
<td>waiting_for_gpu</td>
</tr>
<tr>
<td>1,4</td>
<td>waiting_for_gpu</td>
<td>waiting_for_gpu</td>
</tr>
<tr>
<td>1,5</td>
<td>waiting_for_gpu</td>
<td>waiting_for_gpu</td>
</tr>
<tr>
<td>2,3</td>
<td>waiting_for_gpu</td>
<td>waiting_for_gpu</td>
</tr>
<tr>
<td>2,4</td>
<td>waiting_for_gpu</td>
<td>waiting_for_gpu</td>
</tr>
<tr>
<td>2,5</td>
<td>waiting_for_gpu</td>
<td>waiting_for_gpu</td>
</tr>
<tr>
<td>3,4</td>
<td>waiting_for_gpu</td>
<td>waiting_for_gpu</td>
</tr>
<tr>
<td>3,5</td>
<td>waiting_for_gpu</td>
<td>waiting_for_gpu</td>
</tr>
</tbody>
</table>

Now the map consists just of keys generated by CPU, and we wait for GPU to compute Tid_Lists and support count. To make the GPU compute Tid_Lists and support count first we need to transfer some information data to the GPU. To generate this information data we use the information from table 1 and table 2. This information data consists of some vectors that we will describe below:

- **α**: Bit array representation of tid_lists

  Let us name the table in 1 as \( \tau_1 \) and table 2 as \( \tau_2 \). In this step, we use \( \tau_1 \) which represents the 1-item-set candidates. In general, we use the \( k \)-items-set table to generate a bit array needed to compute the tid_list for \( (k+1)\)-items-set on GPU. Now, we create an empty bit array \( \alpha \) of size equal to the length of all tid_list bit vectors together of \( \tau_1 \). In mathematical terms this size is represented as:

  \[
  a_{\text{size}} = \sum_{i=0}^{\tau_1_{\text{size}}} \text{tid_list}[i].\text{size}
  \]

  Next, we populate the empty array by tid_list’s bit vectors in the order they appear in the dataset. In few words, we convert tid_list from a table structure to an array structure. The resulted bit array looks like:

  ![Figure 5. Bit vector \( \alpha \)](image)

  This bit vector \( \alpha \) is now ready for the GPU, but without some more information that we will show in the follow, it is useless.

- **β_1**: Array of starting tid_list’s indices

  When using GPU for calculation, mostly we should work with indices. In our case, having the \( \alpha \) bit array we generated before, we need some indices information of where does a specific tid_list begins. Thus, by using \( \alpha \) it is easier to get this information. In figure 6 we highlight \( \alpha \) starting indices of tid_lists.

  ![Figure 6. Starting indices of tid_lists in \( \alpha \)](image)

  We store the indices in an array denoted by \( \beta_1 \) represented in the figure below:

  ![Figure 7. \( \beta_1 \) array storing the starting indices of tid_lists in \( \alpha \)](image)

- **β_2**: Array of ending tid_list’s indices

  We generate another array that represents ending indices of tid_list in \( \alpha \). In the figure 8 we highlight the indices which represent ending indices.

  ![Figure 8. Ending indices of tid_list in \( \alpha \)](image)

  We store the indices in an array denoted by \( \beta_2 \) represented in the figure below:

  ![Figure 7. \( \beta_2 \) array storing the ending indices of tid_lists in \( \alpha \)](image)

- **γ_1**: Array representing first items in an item-set

  Now we are ready to complete our information data that we need to send to GPU memory. Till now we have \( \alpha \), \( \beta_1 \) and \( \beta_2 \). From the table 8, we can see the items-set \{1,2\}, \{1,3\}, \{1,4\},... \{3,5\}. We call the value indices 0,0,0,0,1,1,1,2,2 as first items and 1,2,3,4,2,3,4,3,4 as
second items. Now we create an array $y_1$ which stores first items.

<table>
<thead>
<tr>
<th>Inds</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 9. $y_1$ array storing first items of items-sets of $\tau$

As a note, you should be careful when we have for example 3-items-set candidates like {1,2,3}. In this case first items we consider {1,2} and second item {2}. In general, we consider $[a_1, a_2, ..., a_{k-1}]$ as first candidate and $[a_k]$ as second candidate. The size of this array is defined by the formula:

$$y_{1\text{size}} = \sum_{i=1}^{\tau.\text{size}} \tau.\text{size} - i$$

This formula is used to find the array size only in the first step when we have 1-item-sets.

- $y_2$: Array representing second items in an item-set

Same as the above point, but now we store in an array denoted by $y_2$, the second items.

<table>
<thead>
<tr>
<th>Inds</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 10. $y_2$ array storing second items of items-sets of $\tau$

- $\rho$: Next step result array

The above information data that we provided, are sufficient for the GPU to calculate the intersection and support count of candidates. We need to have another array where the GPU will store the result after it finishes. We call this array as result array and we denote it by $\rho$. Arrays have static sizes so we need to predefine the size of result before we allocate memory on GPU. To find this size we use the candidates generated in table $\tau_2$ and bit vector tid_list from $\tau_1$. The array size corresponds to the formula:

$$\rho_{\text{size}} = \sum_{a,b \in \tau_2} \min(a_{\text{size}}, b_{\text{size}})$$

In our ongoing example, this array will have the size equal to 33 and it looks like:

<table>
<thead>
<tr>
<th>Inds</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

| Bit | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
|     | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
|     | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Figure 10. Result array $\rho$

$\rho_{\text{idx}}$: starting result array indices

This array consists of indices that tell us at what position should we store the result in the result array for specific tid_list bit vectors. In our example this array will consist the following values:

<table>
<thead>
<tr>
<th>Inds</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td>10</td>
<td>14</td>
<td>16</td>
<td>19</td>
<td>20</td>
<td>25</td>
</tr>
</tbody>
</table>

| Bit | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|     | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|     | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Figure 10. Result array $\rho$

- $\omega$: Support count result array

Support count is an array where GPU stores the support count for every candidate generated. The size of this vector is equal to the size of table $\tau_2$. For our example, this is looks like:

<table>
<thead>
<tr>
<th>Inds</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

| Bit | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
|     | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
|     | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Figure 10. Support count array

Now from the step $k$ of candidate generation, we have all the information needed to allocate memory in GPU, easier thread management, and arrays to store result for the intersection and support count of step $k + 1$.

VI. II PARALLEL APRIORI ALGORITHM DESCRIPTION

The parallel approach that we took to implement Apriori in GPU, is not one of the most efficient ways. Apriori follows an iterative process, so for every step we transferred the information data to GPU, and got back the results. The number of steps that Apriori performs depends on the minimum support threshold value that we define and the size of the data set. For this reason our approach is not memory efficient, which sometimes takes more time than sequential one, just to constructs all of the information data.

**Algorithm Parallel_Apriori**

**Input:** dataset $D$, min_sup

**Output:** frequent pattern list $L$

1. $candidates = 1$-item-set candidates
2. $candidates = 2$-item-set-candidates-without-tid_list
3. $info\_data = GENERATEINFODATA(candidates)$
4. while $L > 0$ do
   5. GENERATECANDIDATES(candidates)
   6. kernel SUPPORTCOUNT($info\_data$)
   7. if candidate_sup_count > min_sup then
      8. Consider candidate for the next step
   9. else
      10. prune candidate
11. end
12. update candidates
13. end while
14. return ($L$)

Figure 10. Parallel Apriori algorithm
Parallel Apriori algorithm takes as input a transactional dataset and a minimum support value. It maintains a frequent patterns list that gets updated in every iteration. In the step 1, we generate the 1-item-set candidates together with their corresponding tid_list bit vectors in CPU. In the step 2, we generate 2-items-set candidates without their corresponding tid_list in CPU, and in step 3 we have all the parameters needed to construct the information data to transfer and calculate the next k-items-set step in GPU. In step 4, until the candidate list is not empty we generate the next step candidates, we transfer the information data to GPU, calculate the intersection and support count and return the results to CPU. In step 7-10 in CPU we prune the candidates that have lower support count than the defined minimum support, in step 12 we update the candidates list and get in the same procedure again.

In step 3, we call `GENERATEINFODATA(candidates)` procedure which does all what we explained in the previous subsection. It generates the bit array $\alpha$, array of starting tid_list indices $\beta_1$, array of ending tid_list indices $\beta_2$, array of first elements $\gamma_1$, array of second element $\gamma_2$, the result array $\rho$, result array starting indices $\rho_{idx}$ and the support count vector $\omega$. All of these arrays in the step k, are filled with data expect the result and support count arrays which will get filled from the GPU. We transfer to GPU memory $\alpha, \beta_1, \beta_2, \gamma_1, \gamma_2, \rho_{idx}$ and return from the GPU $\rho$ and $\omega$.

In step 4, we call `SUPPORTCOUNT(info_data)` procedure which gets executed in the GPU. Now in this step, all the arrays that we generated and transferred to the GPU will get sense. To have a more clear understanding of Naïve Bayes classifier, we will take an example. Let’s imagine that we have a dataset on fruits. The attributes are: weight, color, and freshness. The classes are: yes, no.

To have a more clear understanding of Naïve Bayes classifier, we will take an example. Let’s imagine that we have a dataset on fruits. The attributes are: weight, color, and freshness. The classes are: yes, no.

First, let us say that we have generated all the all the arrays $\alpha, \beta_1, \beta_2, \gamma_1, \gamma_2, \rho_{idx}$. Because arrays have static sizes, and we have all the sizes of these arrays calculated, we allocate enough memory in GPU to these arrays. The figure 15 explains how the calculation of result array $\rho$ and support count array $\omega$ are calculated.

In the beginning of GPU execution, we lunch enough thread to handle the execution. In this case, we will discuss the execution trace of thread $t_0$. For all the other threads the execution trace is similar.

In the step 1, thread $t_0$ accesses $\gamma_1$ cell with index 0, and get the value 0, which represents the first item index of table $\tau$. In the step 2, thread $t_0$ accesses $\gamma_2$ cell with index 0, and get the value 1 which represents the second item_set index of table $\tau$.

In the step 3, we access the cells of the array $\beta_1$ with indices 0 and 1. Indices 0 and 1 we got by the 2 previous steps. Indices 0 and 1, point to values 0 and 4 respectively.

In the step 4, we access the cells of the array $\beta_2$ with indices 0 and 1. Indices 0 and 1, point to values 3 and 8 respectively.

At the step 5, with the values we got from steps 3 and 4, thread $t_0$ considers the range of values 0-3 and 4-8 of array $\alpha$ as bit vectors of tid_list of items 1 and 2 in $\tau$. That is, we got 0101 and 0011 ready be in the result array starting from the index 0 until index 3.

In the step 8 we store the number of bits that are set in 0011. The result is 1. The same process is followed by all the other threads. In the end, we return $\rho$ and $\omega$ to host memory, prune the candidates with less support count than minimum support and start the next step.

### VI. NAÏVE BAYESIAN CLASSIFIER

Bayesian classifiers are statistical classifiers. They can predict class membership probabilities such as the probability that a given tuple belongs to a particular class.

Bayesian classification is based on Bayes’ theorem (named after Thomas Bayes). Studies comparing classification algorithms have found a simple Bayesian classifier known as the Naïve Bayesian classifier to be comparable in performance with decision tree and selected neural network classifiers. Bayesian classifiers have also exhibited high accuracy and speed when applied to large databases.

Naïve Bayesian classifiers assume that the effect of an attribute value on a given class is independent of the values of the other attributes. This assumption is called conditional independence. It is made to simplify the computations involved and, in this sense, is considered “naïve”.

To have a more clear understanding of Naïve Bayes classifier, we will take an example. Let’s imagine that we have a dataset on 1000 pieces of fruit. The fruits are: a Banana, Orange and some other fruit, and we also imagine that we know 3 features of each fruit, whether the fruit is long or not, sweet or not and yellow or not.
<table>
<thead>
<tr>
<th>Fruit</th>
<th>Long</th>
<th>Sweet</th>
<th>Yellow</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banana</td>
<td>400</td>
<td>350</td>
<td>450</td>
<td>500</td>
</tr>
<tr>
<td>Orange</td>
<td>0</td>
<td>150</td>
<td>300</td>
<td>300</td>
</tr>
<tr>
<td>Other</td>
<td>100</td>
<td>150</td>
<td>50</td>
<td>200</td>
</tr>
<tr>
<td>Total</td>
<td>500</td>
<td>650</td>
<td>800</td>
<td>1000</td>
</tr>
</tbody>
</table>

**TABLE I. TRAINING DATA SET**

From the table above we know that 50% of the fruits are bananas, 30% are oranges, and 20% are other fruits.

Based on the training set that we have we can say the following:

- From the 500 bananas, 400 (0.8) are Long, 350 (0.7) are Sweet and 450 (0.9) are Yellow
- From the 300 oranges, 0 Long, 150 (0.5) are Sweet and 300 (1) are Yellow
- From the 200 other fruits, 100 (0.5) are Long, 150 (0.75) are Sweet, and 50 (0.25) are Yellow.

From the above data, we now have gathered the probabilities of each of the features for each fruit. Now, if we get a testing record (a fruit), with the features: Long, Sweet and Yellow, our model can classify this fruit based on this formula:

\[
P(X|G_i) = P(G_i) \prod_{k=1}^{n} P(x_k|G_i)
\]

Using the formula above, we select the class/category which has the highest probability given the features of the fruit that we need to predict to which class it belongs. For example in our case for the test data with the features long, sweet and yellow, we have these probabilities:

\[
P(X | Banana) = 0.5 \times 0.8 \times 0.7 \times 0.9 = 0.252
\]
\[
P(X | Orange) = 0
\]
\[
P(X | Other) = 0.2 \times 0.5 \times 0.75 \times 0.25 = 0.1875.
\]

From the results above, we conclude that the test record, belongs to the class Banana, since it has the highest probability of the three other classes.

**VII. INTRODUCTION TO ADC AND GPU-NB**

The exponential growth of data nowadays, has given rise to interesting problems: there is more data than we actually can analyze without sophisticated tools. Many of these approaches are concerned with the organization and mining of useful knowledge from this enormous amount of information. Among these approaches that are applied today the most widely used are the ones that include machines learning and data mining techniques such as automatic document classification (ADC).

The idea behind ADC techniques is to create effective models that are able to classify documents in an automated way, thus minimizing human intervention. The techniques of ADC are widely used today in spam filtering, organization of topic directories and digital libraries, identification writing styles or authorship and so on.

ADC uses the supervised learning paradigm in which a classification model is created based on the previous data, in this case we call it a training set. We then use this model to classify unseen documents (the testing set).

For our project we have implemented Naïve Bayes’ian classifier for ADC. The implementation has been done in GPU, named GPU-NB and we have also implemented the CPU version of GPU-NB which we call the CPU-NB.

The implementation of both GPU-NB and CPU-NB consists of some main phases. The first phase consists of parsing the data from the data set, and dividing the parsed data into the training and testing set. The second phase consists of two sub phases: calculation of the documents term frequencies and the calculation of the probability of the terms within the classes. In the third phase, we test our classification model with a testing set.

The GPU-NB is based on the Naive Bayes more specifically on the Multinomial Naïve Bayes which is a statistical language modeling mostly used for text classification. Based on this approach which calculates the score of a class \( c_i \) as the probability that a document \( d_i \) belongs to that class \( c_i \).

After this, a ranking of classes is created and the classifier will classify the document to the class which is in the first position of the rank. Now, let \( P(c_i|d_i) \) be the probability that a test document \( d_i(a_1, a_2, ..., a_j) \) belongs to class \( c_i \) where \( (a_1, a_2, ..., a_j) \) represents the terms/words in a document \( d_i \).

This probability is calculated using the Bayes Theorem:

\[
P(c_i|d_i) = \frac{P(c_i)P(d_i|c_i)}{P(d_i)}
\]

Based on the Naive Bayes algorithm, we assume that all the terms are independent and don’t have any correlation between each other. This approach allows us to calculate the probability of each term \( a_j \) independently from the other terms. Thus we can define \( P(c_i|d_i) \) as:

\[
P(c_i|d_i) = \frac{P(c_i)\prod_{a_j \in d_i}P(a_j|c_i)}{P(d_i)}
\]

To estimate \( P(a_j|c_i) \) we define it as:

\[
P(a_j|c_i) = \frac{T_{c_i}(a_j) + 1}{\sum_{v \in V} T_{c_i}(v) + 1}
\]

where with \( T_{c_i}(a_j) \) we define the number of occurrences of term \( a_j \) in class \( c_i \) and \( \sum_{v \in V} T_{c_i}(v) \) we define the summation of the occurrences of all terms in a document of class \( c_i \). The \( +1 \) in the numerator and denominator has been added do avoid division by zero which is called Laplace smoothing technique. And the \( V \) in the formula is for the term vocabulary (the terms of a document).

The classification is done by the highest conditional class probability \( c_i \) for a given document \( d_i \). To avoid the problem of the probability that a document with a large number of terms to become quickly close to zero, what we do is we use the sum of logarithms instead of multiplication of probabilities. So, the maximization performed in our algorithm is:
\[ c_{map} = \arg \max_{c_i \in C} [\log P(c_i) + \sum_{1 \leq j \leq |V|} \log P(a_j | c_i)] \]

VIII. IMPLEMENTATION OF CPU-NB

The implementation of CPU-NB is a sequential version of GPU-NB. That is why the CPU-NB served us as a foundation of the GPU-NB since some approaches used in CPU-NB have been acquired by GPU-NB. The algorithm of CPU-NB goes as follow:

Algorithm CPU-NB

**Input:** path P
1. \( D \) – LoadDataset(P);
2. \( SD \) – SplitDataset(D);
3. \( K \) – 10; /k-fold cross validation value;
4. \( Mic.F1 \) – vector < >;
5. \( \text{while} \ 10 < K:\)
6. \( Tra \) – SD.traindataset(K);
7. \( Tes \) – SD.testdataset(K);
8. \( \text{PreProcessTraining(Tra)}; \)
9. \( TrainModel \) – trainNaiveBayes(Tra);
10. \( Results \) – testNaiveBayes(TrainModel, Tes);
11. \( Mic.F1.add(Results) ; \)
12. \( \text{end while} \)
13. ShowResults(Mic.F1);

Algorithm 1 CPU-NB

From the pseudo code above, we can see that the first step of our algorithm is to load the dataset. After the dataset is loaded, we proceed by splitting it into the training dataset and the testing dataset. The splitting of the dataset is done using the K-Fold cross validation, that is a technique where we take the dataset and divide it into \( k \) equal subsets (in our case we have used \( k = 10 \)). Then the algorithm will run \( k \times \text{times} \) (line 5), where in each time one of the \( k \) subsets is used as a testing set and the other \( k -1 \) are used as training set. The results of each iteration are saved and in the end we use the Mic.F1 score metric to show the quality of our classification model (line 13).

IX. IMPLEMENTATION OF GPU-NB

As aforementioned, the implementation of CPU-NB helped us to create the foundation of GPU-NB algorithm. There are three CUDA kernels implemented in our algorithm. We have tried to use as much parallelism as we could by minimizing the data dependencies with preprocessing of the data sets on the CPU. The following details are about each kernel implemented in our algorithm:

- **CalcKernel** – in this kernel we compute the frequency of each term for each class. In this kernel we send all the distinct terms from the training dataset. For each term, this kernel issues a thread which will compute the frequency of that term for each of the classes that we have in the training set. This kernel will create a matrix of \( \text{Term} \times \text{Class} \). This kernel advantage is that there are no dependencies here and everything here is in parallel.

- **ProbabilityKernel** – the probability kernel takes the matrix that was generated by the CalcKernel and computes the probability of each term on each class by issuing one thread for each term. The probability is computed by dividing the frequency of the term in the class with the total number of terms in that class. Once again, this kernel has no dependencies and everything is parallelized. The output of this kernel will again be a matrix of size \( \text{Terms} \times \text{Classes} \).

- **TestingKernel** – this kernel is the classification kernel. The difference between this kernel and the other kernels is that here we will not issue one thread for each term, instead we will issue one thread for each document, which is responsible for computing the probability of the document on each class. This kernel will return the probabilities of all the test documents for each class and the final results will be computed on the CPU part.
Algorithm 1 GPU-NB

**Input:** path P
1. D – LoadDataset(P);
2. SD – SplitDataset(D);
3. K – 10;fk-fold cross validation value;
4. Mic.F1 – vector<>;
5. while K < 10:
6.   Tra – SD.trainDataset(K);
7.   Tes – SD.testDataset(K);
8.   PreProcessTraining(Tra);
9.   CM - CalcKernel<<< >>> ;
10.  PM - ProbabilityKernel <<<CM >>> ;
11.  TestMatrix – TestKernel <<<Tes, PM>> ;
12.  Result – computResult(TestMatrix);
13.  Mic.F1.add(Result);
14. end while
15. ShowResults(Mic.F1);

**Algorithm 2 GPU-NB**

From the Algorithm above and the workflow figure, we load the dataset in the beginning and then divide it in k subsets (we use again the k-fold cross validation technique like in the CPU-NB version). Before we start calling the kernels, we have to preprocess the data for the GPU. The problem here is that the data that needs to be preprocesses as arrays. The details how the kernels work is written below:

**CalcKernel** – this kernel has these parameters:
- Dev_calcMatrix – output matrix
- DocumentArray – concatenated documents (ordered based on the class they belong)
- DistinctTermsarray – distinct terms
- DistinctClassArray – distinct classes
- StartingPointOfClassDoc – index of the documents of a class start in DocumentArray
- EndingPointOfClassDoc – index of the documents of a class end in DocumentArray
- SizeOfdistinctClassArray – this tells us how many classes are in total
- SizeOfTerms – the size of total distinct terms
- SizeOfMatrix – the size of the total matrix
- SizeOfDocs – the total number of terms in the DocumentArray structure.

What we do in the CalcKernel, we create a thread for every Term in the DistinctTermsArray. We have to do a check if the index of the thread isn’t higher than the SizeOfTerms, that’s why we have also send to the device the SizeOfTerms variable. Next, we check the index of the thread, we proceed by taking the term that belongs to the thread. This we have done by for example: if there are 2000 distinct terms in the training dataset, then 2000 threads will take each one a term based on their index. This solution was accomplished because the index of the threads starts from 0, until 1999.

After the thread has taken his term, it will proceed by first checking in each class the frequency that the term appears and save it in the Dev_CalcMatrix.

**ProbabilityKernel** – this kernel has these parameters:
- Dev_probMatrix – the output Matrix
- DistinctTermsarray – distinct terms
- DistinctClassArray – distinct classes
- ClassSummationTerms – the sum of all the frequencies of the terms for each class
- CalcMatrix – the matrix that was the output of the first kernel
- SizeOfdistinctClassArray – this tells us how many classes are in total
- SizeOfTerms – the size of total distinct terms

The probability kernel, also as mentioned before, issues a thread for each term. Each thread, will compute the probability of the term it has by dividing it with the total number of frequencies of the class it is computing from the ClassSummationTerms structure. The result of the thread will be save in the Dev_probMatrix.

**TestingKernel** – this kernel has these parameters:
- Dev_results – the output matrix
- ProbMatrix - the probability matrix (output of the ProbabilityKernel)
- DistinctTermsarray – distinct terms
- ConcatenatedTestDocs – the concatenated documents of the test set
- IndexOfClassTerms – this one is preprocessed in the CPU. This structure holds the information for every term of the ConcatenatedTestDocs, where we can find it in the ProbabilityMatrix.
- StartingPointOfClassDoc – index of the documents of a class start in ConcatenatedTestDocs
- EndingPointOfClassDoc – index of the documents of a class end in ConcatenatedTestDocs
- SizeOfdistinctClassArray – this tells us how many classes are in total
- SizeOfDocs – the total number of terms in the test set
- SizeOfConcatenatedDocs – the total number of documents in the test set.

In this kernel, we issue a thread for each of the documents. That threads, takes each of the terms of that documents, finds the probability of them in the ProbMatrix and then sums up the logarithm value of that probability. That thread does this for each class that exists for the document that it has acquired in the beginning. The results are saved in the Dev_Resul matrix.
To assess the performance of the Apriori and Naïve Bayes parallel algorithms we performed several experiments. These experiments we run in Dozer with AMD Opteron™ Processors 6274 1400MHz, 128 GB of RAM using NVIDIA GPU Tesla K40c (2880 CUDA Cores), CUDA version 3.5 with 3004 MHz memory clock rate.

For the Apriori algorithm, we run it both in sequential and parallel with the datasets:

<table>
<thead>
<tr>
<th>Dataset</th>
<th># of items</th>
<th>Average Length</th>
<th># of Transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chess</td>
<td>76</td>
<td>37</td>
<td>3196</td>
</tr>
<tr>
<td>Retail</td>
<td>16470</td>
<td>10.3</td>
<td>88126</td>
</tr>
<tr>
<td>Mushroom</td>
<td>119</td>
<td>23</td>
<td>8124</td>
</tr>
<tr>
<td>Pumsb</td>
<td>2113</td>
<td>74</td>
<td>49046</td>
</tr>
</tbody>
</table>

The results that we took are presented in the following graphs:

As we can see from, from our results in Apriori, the sequential has a better execution time rather than the parallel one. There are a few reasons that we think this happens:

- Bit vector representation
  Tid_list that we represent as bit vectors usually they are sparse. For example \{1, 2, ..., 200, 201\} we represent as an array of \{0,1,1,....,1,1\}. There is a lot of processing happening in 0 values which takes to much time. We can improve this in the future by removing the 0 and modeling a linked list by two arrays where one of them shows the bit vector and the other one shows the index of only 1s.

- Apriori has an iterative nature, which in case that we want to run in GPU, we have to construct and pass a lot of information from CPU to GPU and the other way around, which is really time consuming. In the future, we should find different approaches to keep some information all the time in memory rather than sending it in every iteration.
Here we present the document collection used for the two algorithms, the accuracy of the classification models and the execution time of CPU-NB and GPU-NB. The main idea of our implementation was to compare the sequential version of Naïve Bayes with the parallel version implemented for NVIDIA GPU’s.


We have used five data sets for our experiments. The datasets we used are:

- *Reuters_ny.dat* – this set contains documents of news from Reuters
- *20news.dat* – this set contains documents from newsgroups
- *ACM_TF.dat* – contains documents from ACM digital
- *Acl_bin.dat* – contains product reviews from Amazon
- *ML.dat* – contains medical documents

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Classes</th>
<th>Number of Attributes</th>
<th>Number of Documents</th>
<th>Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>ML</td>
<td>7</td>
<td>268,783</td>
<td>861,454</td>
<td>30,805</td>
</tr>
<tr>
<td>ACM_TF</td>
<td>11</td>
<td>56,449</td>
<td>24,897</td>
<td>27,687</td>
</tr>
<tr>
<td>20news</td>
<td>20</td>
<td>61,050</td>
<td>18,805</td>
<td>129,511</td>
</tr>
<tr>
<td>Reuters_ny</td>
<td>8</td>
<td>24,985</td>
<td>8,184</td>
<td>42,203</td>
</tr>
<tr>
<td>Acl_bin</td>
<td>2</td>
<td>1,110,351</td>
<td>27,677</td>
<td>181,509</td>
</tr>
</tbody>
</table>

2. Metrics used

To measure the accuracy of the algorithms we have used the standard metric *microF*_1. *MicroF*_1 measures the global effectiveness in terms of decision made by the classifier. We have also mentioned before, we have used the k-fold cross-validation technique for our algorithms.

3. Accuracy comparison between CPU-NB and GPU-NB

The accuracy results of the two implementations of Naïve Bayes are shown below, and we will also give explanation about the reasons for these results.

<table>
<thead>
<tr>
<th>Metric</th>
<th>ML.dat</th>
<th>Mic.F1(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GPU</td>
<td>71.4481</td>
</tr>
<tr>
<td></td>
<td>CPU</td>
<td>42.2508</td>
</tr>
<tr>
<td></td>
<td>difference</td>
<td>-29.197</td>
</tr>
<tr>
<td></td>
<td>ACM_TF.dat</td>
<td>GPU</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CPU</td>
</tr>
<tr>
<td></td>
<td></td>
<td>difference</td>
</tr>
<tr>
<td></td>
<td>20news.dat</td>
<td>GPU</td>
</tr>
</tbody>
</table>

From the experiments above we can see that when using some of the datasets the difference between GPU and CPU was significant. We have researched about this phenomenon, and we believe that the reason why we have better accuracy on the GPU than in CPU is because how the hardware of these platform handle floating point.

We can see that the *ACM_TF* dataset has low accuracy in both platforms, and the reason is that this dataset has the lowest density from all of the other datasets (see table at Document Collection), and we believe that this had an impact on the Naïve Bayes classifier.

4. Runtime comparison between CPU-NB and GPU-NB

Below are the execution time of the two algorithms using the five datasets mentioned earlier.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>CPU-NB runtime</th>
<th>GPU-NB runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>reuters_ny.dat</td>
<td>14 sec</td>
<td>15 sec</td>
</tr>
<tr>
<td>20news.dat</td>
<td>147 sec</td>
<td>76 sec</td>
</tr>
<tr>
<td>ACM_TF.dat</td>
<td>47 sec</td>
<td>28 sec</td>
</tr>
<tr>
<td>acl_bin.dat</td>
<td>186 sec</td>
<td>933 sec</td>
</tr>
<tr>
<td>ML.dat</td>
<td>625 sec</td>
<td>1689 sec</td>
</tr>
</tbody>
</table>

![Figure 16. Execution time: CPU-NB vs. GPU-NB](image-url)
From the results above, we can see that GPU-NB has outperformed the CPU-NB in three datasets: *reuters_ny*, *20news*, and *ACM_TF*, whereas the CPU-NB showed better runtime in two other datasets: *acl_bin* and *ML*.

The runtime of the algorithms was measured from the beginning of the algorithm until the end. The reason why we chose this form to measure is that both of the algorithms have their downside. The CPU-NB does the preprocessing faster, but the problem is it takes longer time to do the computation of the training and testing. The GPU-NB takes more time for preprocessing the dataset, but it is faster when it comes to do the training and the testing of Naïve Bayes. So, for this reason we wanted to include both of their disadvantage in the runtime measurement.

XI. CONCLUSION

In this work, we have implemented sequential and parallel versions of Apriori algorithm and the Naïve Bayes classifier. With this project, we wanted to compare sequential algorithms with parallel algorithms to see the performance differences. We have shown that the Apriori algorithm will not improve if implemented in parallel on GPU, instead it executed slower than the sequential implementation. We have also shown that the GPU implementation of Naïve Bayes showed better accuracy because of how the GPU hardware handles floating point, and also we have shown that the GPU implementation executed faster.

XII. FUTURE WORK

For the future work, we plan to modify the Apriori algorithm in order to run faster. One idea is to model a map by using two vectors, which we believe will increase the performance. Furthermore, we plan to modify also the kernel in Apriori in such a way that it keeps some data that it constantly needs in the GPU rather than sending them back and forth from CPU to GPU and the other way around.

XIII. REFERENCES


[3] J.W.H. J.P "Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree”


