Accelerating Binary Biclustering on Platforms with CUDA-enabled GPUs

Jorge González-Domínguez*, Roberto R. Expósito

Computer Architecture Group, Universidade da Coruña, Campus de A Coruña, 15071 A Coruña, Spain

Abstract

Data mining is nowadays essential in many scientific fields to extract valuable information from large input datasets and transform it into an understandable structure. For instance, biclustering techniques are very useful in identifying subsets of two-dimensional data where both rows and columns are correlated. However, some biclustering techniques have become extremely time-consuming when processing very large datasets, which nowadays prevents their use in many areas of research and industry (such as bioinformatics) that have experienced an explosive growth on the amount of available data. In this work we present *CUBiBit*, a tool that accelerates the search for relevant biclusters on binary data by exploiting the computational capabilities of CUDA-enabled GPUs as well as the several CPU cores available in most current systems. The experimental evaluation has shown that *CUBiBit* is up to 116 times faster than the fastest state-of-the-art tool, *BiBit*, in a system with two Intel Sandy Bridge processors (16 CPU cores) and three NVIDIA K20 GPUs. *CUBiBit* is publicly available to download from `https://sourceforge.net/projects/cubibit`.

Keywords: Data Mining, Biclustering, CUDA, GPU, Multithreading

1. Introduction

The data analyzed in many scientific areas are often provided in a two-dimensional way, with information about the magnitude of some attributes (rows) for different samples (columns). Some examples of such areas are gene expression analyses [25], drug activity [19], text mining [3], marketing [16], pattern recognition [5], information networks [13], scientific collaborations [30], analysis of sensor data [20], or social networks [21]. The first step in order to analyze this data usually consists in applying data mining techniques in order to find relevant patterns. The most common data mining approach is probably clustering, which has been applied for decades in order to identify groups of attributes that share certain relationships [14]. However, traditional clustering fails when trying to find patterns

*Corresponding author

Email addresses: jgonzalezd@udc.es (Jorge González-Domínguez), rreye@udc.es (Roberto R. Expósito) among attributes that are only valid for some samples, as usually happens in gene expression analyses. Biclustering techniques are gaining increasing attention over those kind of scenarios as they search for subsets of attributes that are only similar for a subset of samples [26]. Biclusters are represented as two-dimensional submatrices of the original input dataset.

Many alternatives have been designed for biclustering [6], each with different advantages and drawbacks depending on the characteristics of the input datasets. In this work we will focus on those scenarios where the value for each attribute and sample is binary (zero or one). These types of datasets are present on several fields such as gene expression analyses [10] (representing whether a gene is differentially expressed in an individual or not), text mining [22] (each value equal to one indicates that a certain word is included in a text) or marketing [16] (values are one when a customer buys a certain product). In these scenarios the applications can take advantage of the special characteristics of binary datasets to provide relevant biclusters in lower runtime than generic counterparts [27, 29, 8, 15]. Furthermore, a recent experimental evaluation of several biclustering tools using gene expression data has shown that binary-based approaches can also be useful for quantitative data if previously applying a binary discretization [25].

Despite the performance improvement obtained by those tools that are completely focused on binary biclustering, its computational cost is still prohibitive for large datasets. Scientists could take advantage of High Performance Computing (HPC) architecture in order to accelerate the biclustering procedure. This paper presents *CUBiBit*, an application that accelerates the search for binary biclusters in systems with CUDA-enabled GPUs. It is implemented following a hybrid parallel approach that uses CUDA and the multithreading support of C++11 so that: 1) it offers multi-GPU support; and 2) not only the GPUs but also several CPU cores collaborate in the task of finding the biclusters.

The rest of the paper is organized as follows. Section 2 summarizes the related work and the state of the art. Section 3 describes the implementation of our parallel tool. Runtime performance is evaluated and compared in Section 4. Finally, Section 5 concludes the paper and proposes future work.

2. Related Work

The use of HPC facilities in order to accelerate data mining applications is becoming frequent due to the continue increase of dataset sizes. Among the different HPC approaches, acceleration on GPUs is quite popular. These architectures provide computing power that can be equivalent to a medium-sized supercomputer, which is far more costly and thus less accessible to many researchers or scientists. GPU versions of some popular data mining algorithms have been previously implemented, such as association rule mining [7, 9], classification decision trees [11, 12], feature selection [28] or clustering [17, 4].

In literature there are already some previous works that use GPUs for biclustering. Concretely, a GPU version of the geometric biclustering algorithm was satisfactorily applied to find the common patterns of microarray data for neural processing [18], while an OpenCL implementation of a method based on Pearson's correlation can use different hardware accelerators to speed up the biclustering of microarray data too [24]. Nevertheless, none of these

1	0	1	0	1
1	**1**	**1**	0	0
1	**1**	**1**	0	1
1	**1**	**1**	1	0
0	0	0	1	1

Table 1: Example of a binary matrix with one maximal bicluster.

works are focused on binary data, include multi-GPU or multithreading support and their resulting codes are not publicly available for their use. The only parallel implementation for binary biclustering currently available was developed with the MapReduce approach [23], thus its target HPC facility is completely different to that of CUBiBit. Therefore, up to our knowledge, *CUBiBit* is the first available tool for the biclustering of binary data on multi-GPU platforms. GPUs have been efficiently employed to accelerate other bit-wise algorithms but not for biclustering [1, 2, 31].

As mentioned in the previous section, several biclustering approaches have been suggested to deal with binary two-dimensional matrices, *Bimax* [27] being probably the most commonly used one. *CUBiBit* is based on a bit-wise representation already present in *BiBit* [29], which makes these tools faster thanks to efficiently using Boolean algebra operations. Moreover, a recent review of 17 available biclustering methods [25] has proved that the *BiBit* approach obtains accurate results for gene expression data, especially on cases with many large biclusters. This work also shows that this approach can be useful for quantitative data if applying a binarization.

3. CUBiBit Implementation

CUBiBit is a CUDA and C++ based application that searches for biclusters in an input binary matrix M with dimensions $m \times n$. A bicluster consists of a set of rows and columns (R, C) so that all the values within that subset are one (i.e. $\forall i \in R, \forall j \in C, M[i, j] = 1$). Like the state-of-the-art tools, *CUBiBit* only searches for maximal biclusters (i.e. those that are not entirely contained by any other bicluster) with a minimum number of rows (mnr) and columns (mnc) specified by the user through command line. For instance, if mnr and mnc are equal to two, the binary matrix illustrated by Table 1 has only one maximal bicluster of three rows and columns (between the second and fourth rows, and first and third columns) while it contains several smaller biclusters of two rows/columns.

3.1. Bit-wise Data Structure for Binary Biclusters

A bicluster is a submatrix of the initial dataset where the attributes represented by its rows and the samples represented by its columns are correlated. *CUBiBit* employs a bit-wise approach to indicate which rows and columns are included within the biclusters. Concretely, each bicluster is completely represented by two arrays:

- The pattern of the rows ($rowPat$) with m bits. Each bit in position i indicates whether the value of attribute i is included or not in the bicluster.

- The pattern of the columns (*colPat*) with n bits. The bit in position j is set to one when that column j is included in the bicluster.

The terms row and attribute will be henceforth used indistinctly (the same applies to column and sample). Although the basic unit of digital storage is a single bit, the vast majority of current general-purpose processors use byte-addressable memory architecture. This means that the size of the C++ `boolean` datatype must be 1 byte (8 bits). Instead of using one `boolean` to represent each bit of the patterns *rowPat* and *colPat* (thus wasting 7 bits per byte), *CUBiBit* encodes them into arrays of 4-byte integers (32 bits). Thus, the lengths of these arrays are $\frac{m}{32}$ and $\frac{n}{32}$, respectively, with each integer containing the information of 32 rows or columns. Thanks to the use of patterns we can determine the rows and columns that belong to a bicluster just with logical bit-wise operations. For instance, the pattern p of a subset of rows $(r_1, r_2, ..., r_z)$ can be seen as: $p = r_1 \wedge r_2 \wedge ... \wedge r_z$, where $\wedge$ is the binary AND operator of the $\frac{n}{32}$ integers of each row. The *colPat* of a bicluster is defined as the pattern of the group formed by all the rows contained in it. Similarly, the pattern of the group that consists of all the columns within a bicluster defines its *rowPat*.

The concept of bit-wise pattern was already employed in *BiBit* [29] to make this tool significantly faster than the most widely employed tool for binary biclustering: *Bimax* [27]. However, it was only applied to the columns. As will be seen in Section 3.4, *CUBiBit* extends its use also to the rows in order to reduce memory transfers between CPU and GPU.

3.2. General Algorithm

CUBiBit is a command-line application that receives some parameters as arguments such as the path to the input and output files, the minimum number of rows (*mnr*) and columns (*mnc*), the number of CPU threads or the GPU identifiers. The reference manual of our tool not only includes an explanation of all the arguments, but also installation and execution instructions.

Algorithm 1 provides a high-level pseudocode of the approach followed by *CUBiBit*. The execution starts by reading the input matrix and encoding its values into 32-bit integer arrays (Lines 3 and 4, respectively), which is usually negligible in terms of execution time (i.e. input files are relatively small). The first step with a significant computational cost is the bicluster initialization, comprised between Lines 5 and 12 in Algorithm 1. This step is in charge of determining the columns included within each bicluster (i.e. the one values in *colPat*) and needs two nested loops to check all possible attribute pairs. A bicluster formed by these two attributes is valid for the next phase if the pattern fulfills two conditions. On the one hand, the number of columns in the bicluster (i.e. the number of bits equal to one in *colPat*) must be equal or higher than *mnc* (Line 10). On the other hand, no other bicluster must have the same column pattern, as we are looking for maximal biclusters (Line 11). The C++ `set` container is used to save the *colPat* of all the biclusters as it works faster than a `list` for insertions, deletions and searches when each element can be identified by a unique key (logarithmic complexity instead of linear). In this case the key is equal to *colPat*, as no biclusters with the same pattern are allowed. Therefore, the complexity of the bicluster initialization is $O(m^2 \cdot log(m) \cdot n)$.

```
1  INPUT: Path to the files for input ($ifile$) and output ($ofile$)
2  INPUT: Integers $mnr$ and $mnc$ with the minimum number of rows and columns per
     bicluster, respectively
3  Read matrix $M$ of dimensions $m \times n$ from $ifile$
4  Encode $M$ into a 32-bit integer matrix $D$ of dimensions $m \times \frac{n}{32}$
5  Initialize empty bicluster set $S$
6  for Each row $i$ in $D$ from $0$ to $m-2$ do
7      for Each row $k$ in $D$ from $i+1$ to $m-1$ do
8          $colPat$ := row $i \wedge$ row $k$                    #Bit-wise AND of all columns
9          $numCols$ := number of ones in $colPat$
10         if $numCols \geq mnc$ then
11             if No bicluster in $S$ with column pattern equal to $colPat$ then
12                 Insert in $S$ a new bicluster with $colPat$
               end
           end
       end
   end
13 for Each bicluster $b$ in $S$ with $colPat$ do
14     Initialize $rowPat$ with all bits to zero
15     $numRows := 0$
16     for Each row $r$ in $D$ do
17         if $colPat \wedge row\ r == colPat$ then
18             Set the bit $r$ of $rowPat$ to one               # Row included in the bicluster
19             $numRows := numRows + 1$
           end
       end
20     if $numRows \geq mnr$ then
21         Print into $ofile$ the information of $b$
       end
   end
```

Algorithm 1: Pseudocode of CUBiBit.

Nevertheless, the most computationally demanding step starts in Line 13 and consists in completing the information of all the biclusters that were initialized in the previous phase by calculating the $rowPat$ (i.e. determining which rows are included into them). As explained in the previous subsection, *CUBiBit* only needs $\frac{n}{32}$ logical AND operations between *colPat* and the data of the row to determine whether it belongs to the bicluster or not (Line 17). In this case *rowPat* is updated accordingly (Line 18). With nb the number of biclusters initialized in the previous step, the complexity of the bicluster completion is $O(nb \cdot m \cdot n)$. As the number of biclusters can be up to m^2, the maximum complexity is $O(m^3 \cdot n)$.

Finally, only those biclusters with at least mnr rows are printed into the output file. We use the same output format as *BiBit*: one line per bicluster with the information of number of rows, number of columns, rows ids and columns ids separated by semicolons. We refer again to [29] if further explanation about the general algorithm (even with a comprehensible example) is necessary.

3.3. Parallel Bicluster Initialization on Several CPU Cores

The bicluster initialization step, where *CUBiBit* calculates the columns associated to the biclusters, checks every pair of attributes as indicated between Lines 5 and 12 in Algorithm 1. As far as opportunities of parallelization are concerned, different resources could compute different row pairs at the same time. However, we must take into account that several resources simultaneously searching and/or inserting in the `set` container could lead to potential race conditions. Therefore, all the `set` management must be synchronized, thus limiting the amount of resources that can efficiently work in parallel. This means that the bicluster initialization step is not suitable to be parallelized in modern GPUs. Note that these devices currently provide thousands of cores, and the synchronization overhead would prevent to achieve an efficient parallelization in the GPU.

However, powerful multicore CPUs are also available in most current platforms. *CUBiBit* can take advantage of these dozens of CPU resources to parallelize the bicluster initialization step using the built-in multithreading support provided by C++11. Several computation threads can be executed over different CPU cores in order to simultaneously calculate the *colPat* of different attribute pairs. More concretely, *CUBiBit* uses a cyclic distribution over the first loop (see Line 6). It means that the thread with id $tidx$ works over all pairs that start with attributes $tidx$, $tidx + numTh$, $tidx + 2 \cdot numTh$, etc., being $numTh$ the total number of CPU threads. The `set` is stored in a position of the main memory that is accessible to all CPU cores. Moreover, a mutex is employed to serialize the accesses and insertions in the `set` in order to avoid the race conditions when two threads simultaneously try to insert biclusters with the same pattern.

3.4. Parallel Bicluster Completion on Several GPUs

As previously mentioned, the detection of the rows that belong to each bicluster is the most computationally demanding step of the algorithm, especially in those scenarios as gene expression analysis where there are more attributes than samples. All rows must be analyzed for each bicluster that was initialized in the previous step. This phase is very suitable for parallelization as the work for each bicluster is independent. The checking of different rows for the same bicluster can also be developed in parallel. A two-level parallelization approach with CUDA and C++11 multithreading was included in *CUBiBit* so that it is able to exploit several GPUs installed in a shared-memory platform during this step.

All the computation for bicluster completion is included in one CUDA kernel, whose pseudocode is represented in Algorithm 2. The inputs are the encoded data matrix and the *colPat* of each bicluster found in the initialization step, while the outputs are the *rowPat* of those biclusters. After the kernel execution by the GPU, the CPU is responsible of writing into the output file the results of those complete biclusters with at least *mnr* rows. All the

```
1  INPUT: Encoded transposed matrix $D$
2  INPUT: An array with the $colPat$ of the biclusters
3  Copy to shared memory the $colPat$ of the bicluster associated to the block
4  Synchronize the threads of the block
5  # Each thread with index $tidx$ of the block (with $numTh$ threads) in parallel:
6  32-bit integer $myRowPat := 0$
7  for Each integer $i$ from $0$ to $31$ do
8      if Row $(tidx + numTh \cdot i)$ in $D \wedge colPat == colPat$ then
           # The row belongs to the bicluster
9          $myRowPat := myRowPat + 1$                        #Set the bit to 1
       end
10     $myRowPat := myRowPat \cdot 2$           # Move to the next bit to update
   end
11 $rowPat[tidx] := myRowPat$
12 OUTPUT: $rowPat$ of the bicluster
```

Algorithm 2: Pseudocode of the GPU kernel for biclustering completion. Each CUDA block is in charge of one bicluster.

$colPat$ and $rowPat$ values are consecutively stored in memory in order to perform the CPU-GPU transfers at once. Note that the use of 32-bit integer-based patterns, with only one bit to represent whether each attribute or sample belongs to the bicluster, reduces the GPU memory requirements and the performance impact of the memory transfers. Concretely, the total amount of memory transferred from CPU to GPU is $m \cdot n$ bits for the encoded data matrix, and $nb \cdot n$ bits for the $colPat$ of the initial biclusters (nb again being the number of initialized biclusters). The memory transferred from GPU to CPU are the $nb \cdot m$ bits necessary to store the $rowPat$ of those biclusters.

Each bicluster is completed by one CUDA block, and its threads analyze in parallel whether different rows belong to the associated bicluster or not. In case that the amount of biclusters $numBi$ that passed the initialization step is higher than the maximum number of CUDA blocks ($2^{31} - 1$ in current NVIDIA GPUs), *CUBiBit* requires $\frac{numBi}{2^{31}-1}$ calls to the kernel. As can be seen in Line 7, each thread of the block is responsible of 32 rows, i.e., one 32-bit integer element of $rowPat$ (Line 11). Only the logical AND operations shown in Line 8 are necessary to find whether the attributes belong to the cluster. In that case, the bit is set to one (Line 9) and next row is considered (Line 10). As modern NVIDIA GPUs limit the maximum number of threads to 1,024, *CUBiBit* can only work with datasets with a maximum of $1,024 \cdot 32 = 32,768$ samples. Nevertheless, this amount of columns is more than enough on realistic scenarios, especially for gene expression analyses, where the number of attributes is usually significantly higher than the number of samples.

Note a number of optimizations included in our CUDA implementation in order to increase the performance of *CUBiBit*:

- The $colPat$ of the bicluster is copied to GPU shared memory (Line 3). This is a small

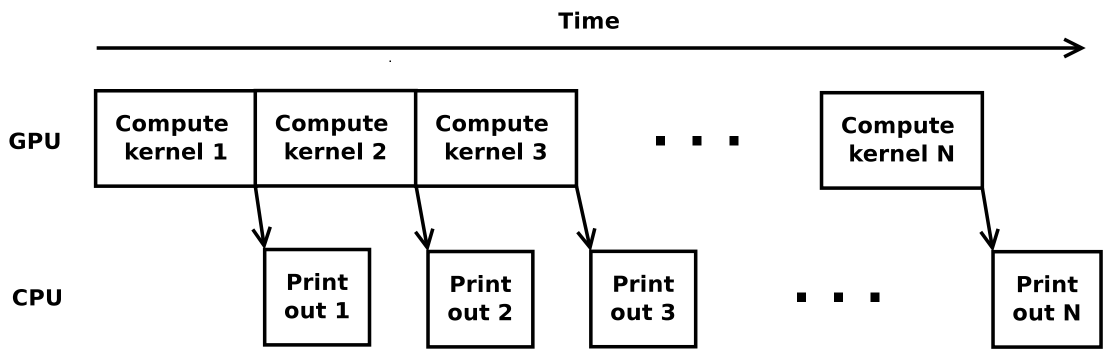

Figure 1: Overlapping between GPU and CPU work. The GPU computes one kernel while the CPU writes the results provided by the previous kernel call.

portion of memory (tens of KB) accessible to all the threads within each block but independent among blocks. This type of memory presents a much higher bandwidth than GPU global memory. Consequently, *CUBiBit* performs highly efficient memory accesses to *colPat* (32 accesses per thread). One synchronization among the threads of the block (Line 4) is necessary to guarantee that the whole pattern has been copied before starting the computation.

- The increase of coalescence in the GPU global memory accesses is one of the most effective and most common optimization techniques for CUDA codes. It consists in combining multiple memory accesses into a single transaction and requires that consecutive threads access consecutive elements in memory. Instead of using a matrix with attributes stored in the rows and samples in the columns (like for CPU), the encoded matrix D transferred to the GPU is transposed in order to increase the coalescence of the memory accesses (consecutive threads simultaneously access consecutive positions in D).

- Each thread $tidx$ is only responsible for the $tidx - th$ 32-bit integer of $rowPat$ (Line 11), which guarantees also coalesced accesses to the output array.

- Data on the CPU is stored in pinned host memory, which usually obtains better I/O performance when copying this data to the device than non-pinned buffers.

- *CUBiBit* makes efficient use of CUDA streams and asynchronous memory transfers in order to overlap the kernel computation on the GPU with the writing of the results performed by the CPU. While the GPU is completing one group of $2^{31} - 1$ biclusters, the CPU writes into the output file the results of the previous group, as illustrated in Figure 1. Therefore, our tool minimizes the impact of the output printing on performance.

Finally, *CUBiBit* can use several GPUs by launching several C++11 threads, each one associated to one different GPU. The biclusters that pass the initialization step are evenly

Name	#SMs	#CUDA cores	Core frequency	Memory size
K20m	13	2496	706 MHz	5 GB
K40c	15	2880	745 MHz	12 GB

Table 2: Specifications of the two types of GPUs used for the experimental evaluation.

distributed among the CPU threads. Each thread is responsible of initializing its correspoding GPU, performing the memory transfers, calling the kernel only for those biclusters distributed to it, and printing the output results. These CPU threads only require synchronization using a mutex to sequentialize the writings into the output file.

4. Performance Evaluation

The performance of *CUBiBit* has been evaluated on a multicore system with two Intel Sandy Bridge processors (in total, 16 CPU cores at 2.20GHz) that contains four NVIDIA Tesla Kepler GPUs: three K20m and one K40c. Table 2 shows some interesting specifications of these GPUs, such as the number of Streaming Multiprocessors (SM) or the number of CUDA cores they provide. Our tool is compiled with NVCC version 8.0.61. The evaluation is focused on performance in terms of execution time, as the biclustering approach of *CUBiBit* is equivalent to the one of *BiBit* and its accuracy was already satisfactorily tested in previous studies [25]. The input datasets were created by randomly generating one and zero values. Note that the execution time of *CUBiBit* and *BiBit* is independent of the origin of the input data. As performance depends on the size of the data and the biclusters, synthetic datasets are as suitable as those with real data for performance evaluation. Similarly to the performance comparison in the aforementioned study, the number of samples is constant (200), while the number of attributes varies from 6,400 to 25,600. The percentage of one values in the simulated datasets is 15%. The results shown in this section were obtained by searching for biclusters with at least 1% of the input matrix dimensions (i.e. 64×2, 128×2 and 256×2 submatrices for the datasets with 6,400, 12,800 and 25,600 attributes, respectively).

Table 3 compares the total runtime of *CUBiBit* and the state-of-the-art tool *BiBit* compiled and executed with the Java runtime version 1.7.0_121. As mentioned in Section 1, up to our knowledge *BiBit* is the fastest available tool for biclustering of binary data. *CUBiBit* runtime includes the impact of intrinsically sequential parts of the code such as I/O operations or CPU-memory transfers. In all the experiments *CUBiBit* makes use of the 16 CPU cores for the multithreaded bicluster initialization step (as shown in Subsection 3.3). As can be seen, our tool is on average 48.80, 73.98 and 61.76 times faster than *BiBit* when exploiting one K20m, three K20m and one K40c GPUs, respectively. As expected, speedups increase with the number of attributes as more work must be performed in parallel (for instance, accelerations obtained on the K40c GPU are 33.92 and 92.00 for 6,400 and 25,600 attributes, respectively).

However, not all the acceleration is achieved thanks to the parallelization of the algorithm, but also because of using C++ more efficient memory management than the Java one

Tool	**GPUs**	**Number of rows**		
		6,400	12,800	25,600
BiBit	-	31.55	214.27	1541.89
seq-CUBiBit	-	10.83	73.64	504.85
	1 K20m	1.06	4.42	22.62
CUBiBit	3 K20m	0.86	3.13	13.20
	1 K40c	0.93	3.61	16.76

Table 3: Runtimes (in minutes) of *CUBiBit* on an Intel Sandy Bridge system with 16 CPU cores and different GPU alternatives: one K20m, three K20m and one K40c. The runtime of a sequential custom-made C++ counterpart (*seq-CUBiBit*) and *BiBit* are also included for comparison purposes. All tools look for biclusters with dimensions at least 1% of the input dataset size. The number of samples and the percentage of one values in the input dataset are constant: 200 and 15%, respectively.

included in *BiBit*. In order to provide an insight about how good is our parallel approach, we have developed a C++ sequential version (denoted as *seq-CUBiBit* in Table 3) that is completely similar to *CUBiBit* but without any parallelization (neither C++11 multithreading in the bicluster initialization nor CUDA kernel for bicluster completion). This sequential implementation, compiled with the GNU compiler version 4.9.2, is on average 2.96 times faster than *BiBit*. By comparing the runtime of *CUBiBit* and this sequential version we can assert that the overall acceleration provided by the parallelization is 16.40, 24.79 and 20.72 using one K20m, three K20m and one K40c, respectively. Note that these speedups are higher for the largest scenario, where the percentage of time spent in the CUDA-parallelized step (biclustering completion, see Subsection 3.4) is higher. For instance, *CUBiBit* using 16 CPU cores and three K20m GPUs is able to find all the biclusters of the dataset with 25,600 rows in around 13 minutes, while the sequential C++ counterpart requires more than 8 hours (speedup of 38.25) and *BiBit* more than one day (speedup of 116.81).

Although the increase of speedup when using several K20m compared to only one GPU does not seem too impressive, it is not due to a bad collaboration between CPU threads and CUDA kernels, but because only the bicluster completion step benefits from it. The graphs in Figure 2 show the runtime breakdown for *CUBiBit* using one and three K20m GPUs, as well as that of the C++ sequential version. While biclustering completion is the most computationally demanding step in the sequential implementation (on average, 93.74% of the total runtime), its impact is reduced to a mere 50.84% using one K20m GPU. Therefore, the potential benefit of using several GPUs is limited to only around half of the execution time, and it is even worse for small datasets (only 33.32% of the runtime is dedicated to bicluster completion with one K20m GPU when analyzing the dataset with 6,400 attributes). The performance breakdown proves that the CUDA optimizations described in Subsection 3.4 make the *CUBiBit* kernel highly scalable. In fact, the average runtime reduction of the bicluster completion step on one K20m GPU compared to the same step in the sequential C++ counterpart is 30.03. This average speedup is increased to 69.39 when using three K20m GPUs (with a maximum speedup of 82.14 for the largest dataset). The benefit of the multithreaded approach implemented in the bicluster initialization step is more limited

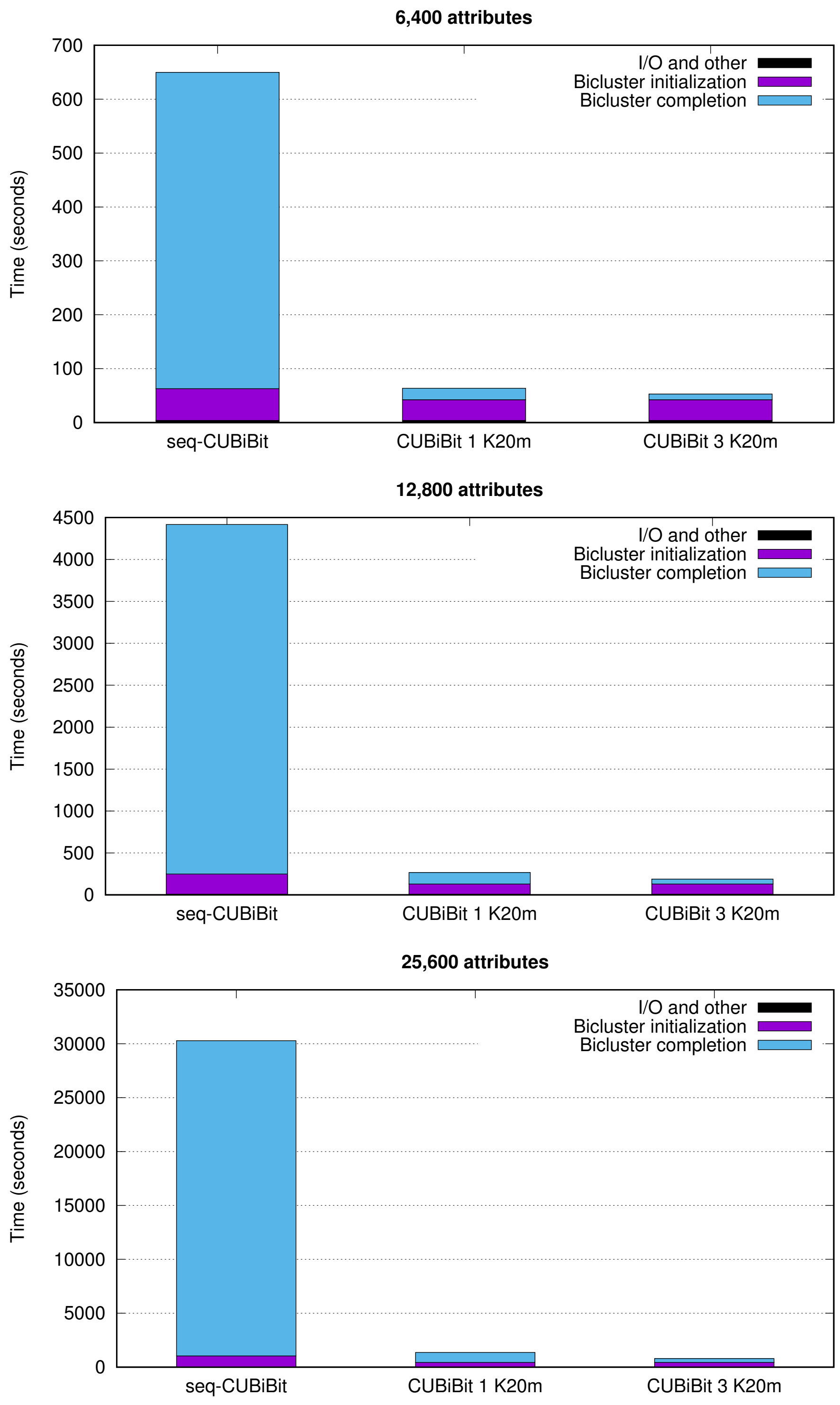

Figure 2: Performance breakdown of *CUBiBit* and the custom-made C++ sequential counterpart. Both versions look for biclusters with dimensions at least 1% of the input dataset size. The number of samples and the percentage of one values in the input dataset are constant: 200 and 15%, respectively.

because the CPU resources are not so powerful as GPU ones. Furthermore, the CPUs require a synchronization when accessing the C++ `set` as mentioned in Subsection 3.3. As expected, the runtime of intrinsically sequential steps such as input reading and memory/structures initialization is constant for *seq-CUBiBit* and *CUBiBit*. The analysis of these results also explains the reason why the speedup obtained by *CUBiBit* increases with the number of attributes: the bicluster completion step has more impact on the runtime of the whole algorithm.

5. Conclusions

This work has presented *CUBiBit*, a high-performance C++-based biclustering tool to process very large binary datasets on multi-GPU platforms. In fact, our tool follows a hybrid parallelization approach that takes full advantage of the abundant computing resources provided by CUDA-enabled GPUs as well as multicore CPUs. The experimental results have shown significant speedups when compared to a representative Java-based biclustering tool, reducing the execution of *BiBit* by up to 116x when using three K20 GPUs and two 8-core CPUs. Moreover, the scalability of *CUBiBit* has been assesed by comparing it to a custom-made C++ sequential implementation. These experiments indicate that our parallelization provides a maximum speedup of 38 when processing a dataset with 25,600 genes. *CUBiBit* is publicly available to download from https://sourceforge.net/projects/cubibit.

As future work, we aim to include in *CUBiBit* a more general (although maybe less efficient) GPU kernel that can work with datasets that contain more than 32,768 samples. The tool should choose the proper kernel to execute in the GPU depending on the size of the input dataset. Moreover, we will explore the use of GPUs to accelerate biclustering techniques for quantitative data.

Acknowledgments

This work was supported by the Ministry of Economy, Industry and Competitiveness of Spain and FEDER funds of the European Union [grant TIN2016-75845-P (AEI/FEDER/UE)], as well as by Xunta de Galicia (Centro Singular de Investigacion de Galicia accreditation 2016-2019, ref. EDG431G/01).

References

[1] Abell, S., Do, N., Lee, J.J., 2016. GPU-LMDDA: a Bit-Vector GPU-Based Deadlock Detection Algorithm for Multi-Unit Resource Systems. International Journal of Parallel, Emergent and Distributed Systems 31, 562–590.

[2] AlBdaiwi, B.F., AboElFotoh, H.M., 2017. A GPU-Based Genetic Algorithm for the P-Median Problem. The Journal of Supercomputing 73, 4221–4244.

[3] Amancio, D.R., 2015. A Complex Network Approach to Stylometry. PLoS One 10, e0136076.

[4] Arefin, A.S., Riveros, C., Berretta, R., Moscato, P., 2012. GPU-FS-kNN: A Software Tool for Fast and Scalable kNN Computation Using GPUs. PloS ONE 7, 1–13.

[5] Blondel, V.D., Guillaume, J.L., Lambiotte, R., Lefebvre, E., 2008. Fast Unfolding of Communities in Large Networks. Journal of Statistical Mechanics: Theory and Experiment 2008, P10008.

[6] Busygin, S., Prokopyev, O., Pardalosa, P.M., 2008. Biclustering in Data Mining. Computers and Operations Research 35, 2964–2987.

[7] Cano, A., Luna, J.M., Ventura, S., 2013. High Performance Evaluation of Evolutionary-Mined Association Rules on GPUs. The Journal of Supercomputing 66, 1438–1461.

[8] Chen, H.C., Zou, W., Tien, Y.J., Chen, J.J., 2013. Identification of Bicluster Regions in a Binary Matrix and Its Applications. PLoS ONE 8, 1–13.

[9] Djenouri, Y., Bendjoudi, A., Mehdi, M., Nouali-Taboudjemat, N., Habbas, Z., 2015. GPU-Based Bees Swarm Optimization for Association Rules Mining. The Journal of Supercomputing 71, 1318–1344.

[10] Eren, K., Deveci, M., Küçüktunç, O., Çatalyürek, Ü.V., 2013. A Comparative Analysis of Biclustering Algorithms for Gene Expression Data. Briefings in Bioinformatics 14, 279–292.

[11] Jian, L., Wang, C., Liu, Y., Liang, S., Yi, W., Shi, Y., 2013. Parallel Data Mining Techniques on Graphics Processing Unit with Compute Unified Device Architecture (CUDA). The Journal of Supercomputing 64, 942–967.

[12] Jurczuk, K., Czajkowski, M., Kretowski, M., 2016. Evolutionary Induction of a Decision Tree for Large-Scale Data: a GPU-Based Approach. Soft Computing 21, 7363–7379.

[13] Kaya, M., Alhajj, R., 2014. Development of Multidimensional Academic Information Networks with a Novel Data Cube Based Modeling Method. Information Sciences 265, 211–224.

[14] Kerr, G., Ruskin, H.J., Crane, M., Doolan, P., 2008. Techniques for Clustering Gene Expression Data. Computers in Biology and Medicine 38, 283–293.

[15] Lee, S., Huang, J.Z., 2014. A Biclustering Algorithm for Binary Matrices Based on Penalized Bernoulli Likelihood. Statistics and Computing 24, 429–441.

[16] Li, T., 2005. A General Model for Clustering Binary Data, in: 11th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining (KDD 2005), Chicago, IL, USA. pp. 188–197.

[17] Li, Y., Zhao, K., Chu, X., Liu, J., 2010. Speeding Up K-Means Algorithm by GPUs, in: 10th IEEE International Conference on Computer and Information Technology (CIT 2010), Bradford, United Kingdom. pp. 115–122.

[18] Liu, B., Xin, Y., Cheung, R.C., HongYan, 2014. GPU-Based Biclustering for Microarray Data Analysis in Neurocomputing. Neurocomputing 134, 239–246.

[19] Liu, J., Wang, W., 2003. Op-cluster: Clustering by Tendency in High Dimensional Space, in: 3rd IEEE International Conference on Data Mining (ICDM 2003), Melbourne, FL, USA. pp. 187–194.

[20] Liu, Y., Nie, L., Han, L., Zhang, L., Rosenblum, D.S., 2015. Action2Activity: Recognizing Complex Activities from Sensor Data, in: 24th International Joint Conference on Artificial Intelligence (IJCAI 2015), Buenos Aires, Argentina. pp. 1617–1623.

[21] Liu, Y., Zhang, L., Nie, L., Yan, Y., Rosenblum, D.S., 2016. Fortune Teller: Predicting Your Career Path, in: 13th AAAI Conference on Artficial Intelligence (AAAI 2016), Phoenix, AR, USA. pp. 201–207.

[22] Mimaroglu, S., Uehara, K., 2007. Bit Sequences and Biclustering of Text Documents, in: 7th International Conference on Data Mining (ICDM 2007), Omaha, NE, USA. pp. 51–56.

[23] Nisar, A., Ahmad, W., Liao, W.K., Choudhary, A., 2015. An Efficient Map-Reduce Algorithm for Computing Formal Concepts from Binary Data, in: 2015 IEEE International Conference on Big Data (IEEE Big Data 2015), Santa Clara, CA, USA. pp. 1519–1528.

[24] Orzechowski, P., Boryczko, K., 2015. Rough Assessment of GPU Capabilities for Parallel PCC-Based Biclustering Method Applied to Microarray Data Sets. Bio-Algorithms and Med-Systems 11, 243–248.

[25] Padilha, V.A., Campello, R., 2017. A Systematic Comparative Evaluation of Biclustering Techniques. BMC Bioinformatics 18, 55.

[26] Pontes, B., Giráldez, R., Aguilar-Ruiz, J.S., 2015. Biclustering on Expression Data: a Review. Journal of Biomedical Informatics 57, 163–180.

[27] Prelic, A., Bleuler, S., Zimmermann, P., Wille, A., Bühlmann, P., Gruissem, W., Hennig, L., Thiele, L., Zitzler, E., 2006. A Systematic Comparison and Evaluation of Biclustering Methods for Gene Expression Data. Bioinformatics 22, 1122–1129.

[28] Ramírez-Gallego, S., Lastra, I., Martínez-Rego, D., Bolón-Canedo, V., Benítez, J.M., Herrera, F.,

Alonso-Betanzos, A., 2017. Fast-mRMR: Fast Minimum Redundancy Maximum Relevance Algorithm for High-Dimensional Big Data. International Journal of Intelligent Systems 32, 134–152.

[29] Rodríguez-Baena, D.S., Pérez-Pulido, A.J., Aguilar-Ruiz, J.S., 2011. A Biclustering Algorithm for Extracting Bit-Patterns from Binary Datasets. Bioinformatics 27, 2738–2745.

[30] Viana, M.P., Amancio, D.R., da F Costa, L., 2013. On Time-Varying Collaboration Networks. Journal of Informetrics 7, 371–378.

[31] Zhang, J., Zhu, Y., Pan, Y., Li, T., 2016. Efficient Parallel Boolean Matrix Based Algorithms for Computing Composite Rough Set Approximations. Information Sciences 329, 287–302.