FPGA-based OpenCL Accelerator for Discovering Temporal Patterns in Gene Expression Data Using Biclustering

Rui P. Duarte, Álvaro Simões, Rui Henriques
INESC-ID and Dep. of Computer Science and Engineering
Univ. de Lisboa, Portugal
rui.duarte,alvaro.simoes,rmch@tecnico.ulisboa.pt

Horácio C. Neto
INESC-ID and Dep. of Electrical and Computer Engineering
Univ. de Lisboa, Portugal
hcn@inesc-id.pt

ABSTRACT

Biclustering is a prominent task for the analysis of biological data, including gene expression data from microarray experiments or molecular concentrations from mass spectrometry. Biclustering performs simultaneous clustering on biological entities and samples, enabling the discovery of non-trivial regulatory modules with putative biological functions. Although several biclustering algorithms have been proposed, existing algorithms are challenged by efficiency aspects given the inherent NP-hard nature of this task and the high-dimensionality of biological data.

This work proposes principles to accelerate state-of-the-art biclustering algorithms, with a focus on the QUalitative BIClustering (QUBIC) algorithm given its inherent efficiency and ability to identify statistically significant biclusters. To this end, this work introduces a parallel architecture for QUBIC based on an heterogeneous system composed by a FPGA and a CPU that communicate through the OpenCL framework. The proposed architecture is sound and surpasses the need for dedicated electronic solutions. Results show a reduction up to 29x in the execution time of the functions accelerated on the FPGA, when compared to a software-only implementation.

CCS CONCEPTS

• **Computing methodologies** → **Massively parallel algorithms**; **Parallel programming languages**; • **Computer systems organization** → **Multicore architectures**;

KEYWORDS

FPGA, OpenCL, Biclustering, QUBIC, Gene Expression, Parallel Programming, Hardware/Software Accelerators

ACM Reference Format:
Rui P. Duarte, Álvaro Simões, Rui Henriques and Horácio C. Neto. 2018. FPGA-based OpenCL Accelerator for Discovering Temporal Patterns in Gene Expression Data Using Biclustering. In *PBio 2018: 6th International Workshop on Parallelism in Bioinformatics (PBio 2018), September 23, 2018, Barcelona, Spain.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3235830.3235836

PBio 2018, September 23, 2018, Barcelona, Spain

ACM ISBN 978-1-4503-6531-4/18/09…$15.00
https://doi.org/10.1145/3235830.3235836

1 INTRODUCTION

The lower costs associated with microarray technology (DNA chips) to profile gene expression, as well as mass spectrometry to measure molecular concentrations, create unprecedented opportunities to study the role of a vast number of biological entities from samples collected under different conditions or from large populations of individuals. Gene expression data is typically represented by a matrix where each entry represents the expression level of a specific gene on a given sample. Figure 1 shows an illustrative outcome of DNA microarray technology.

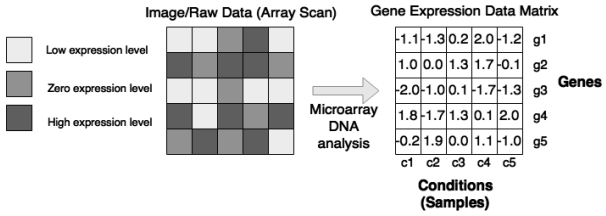

Figure 1: Mapping DNA chips into gene expression data.

The need to analyze biological data with increasing dimensionality is shaping the newly developed data mining methods, showing an underlined focus towards the discovery of local regulatory patterns [14]. Recent studies in genomics, proteomics and metabolomics show that biclustering is a well-established and powerful way of exploring biological data [1, 15]. Given a real-valued matrix, biclustering seeks to find sub-matrices (biclusters), subsets of rows with a coherent pattern across subsets of columns. In the context of gene expression data, a bicluster defines a local regulatory pattern characterized by a subset of genes with coherent expression values on a subset of samples.

Given the combinatorial nature of biclustering, the input size of biclustering problems grows in quadratic order in the number of rows or columns of a dataset [15]. In this context, exhaustive biclustering searches become impractical since gene expression matrices have thousands of genes [2]. As such, in order to cope with computational complexity of biclustering tasks, biclustering algorithms place constraints on the performed searches [3]. The commonly placed constraints (including restrictions on the allowed structure, coherence and optimality of biclustering solutions) impact the quality and relevance of the found biclusters, and are generally insufficient to guarantee the scalability of the searches [15].

In this context, there has been an increasing number of efforts to accelerate biclustering algorithms recurring to parallelization and

distribution principles [2, 13], as well as some initial efforts to establish dedicated hardware computing architectures [2, 13]. Despite the efforts, the current research leaves unexplored the potential of Field-Programmable Gate Array (FPGA)-based architectures for accelerating biclustering algorithms.

In an effort to address existing limitations, this work aims to study the potential of accelerating biclustering algorithms within a combined hardware/software architecture, designed and configured in an FPGA-based heterogeneous system using the OpenCL framework.

For this purpose, a specific class of state-of-the-art biclustering algorithms – clique-based biclustering algorithms – are selected as candidate for this study as they show an already competitive efficiency profile. Clique-based biclustering algorithms formulate the biclustering problem as a task of discovering (bi)cliques in (bipartite) graphs mapped from the input data. In particular, the QUalitative BIClustering (QUBIC) algorithm is here considered given its superior efficiency against peer biclustering algorithms and effectiveness to identify statistically significant patterns from gene expression data. Although QUBIC searches are known to be highly efficient as they rely on effective anti-monotonic principles, the performance of QUBIC still has severe efficiency bottlenecks for biological data with thousands of biological entities [4].

Collected results of the accelerated QUBIC implementation against its original (software-only) implementation show significantly efficiency gains in execution times, evidencing the relevance of the proposed contributions.

This paper is organized as follows. *Section 2* presents the background on biclustering and OpenCL. *Section 3* introduces the proposed method to accelerate a specific class of biclustering algorithms. *Section 4* discusses the results obtained, thus, demonstrating performance improvement. Conclusions are drawn in *Section 5*.

2 BACKGROUND

2.1 Biclustering

Definition 2.1. Given a real-valued matrix, $A=(X, Y)$, with a set of rows $X=\{x_1, .., x_n\}$, a set of columns $Y=\{y_1, .., y_m\}$, and entries $a_{ij}\in\mathbb{R}$ that relate row i and column j:

- A **bicluster** $B = (I, J)$ is a $r \times s$ submatrix/subspace of A, where $I = (i_1, .., i_r) \subseteq X$ is a subset of rows and $J = (j_1, .., j_s) \subseteq Y$ is a subset of columns;
- The **biclustering task** aims to identify a structure of biclusters $\mathcal{B} = \{B_1, .., B_p\}$ such that each bicluster $B_k = (I_k, J_k)$ satisfies specific criteria of *homogeneity* and *statistical significance* [16].

The concept of biclustering, firstly introduced in [6], is now largely applied for biological data analysis with particular incidence over gene expression data [7, 17–21]. Contrasting with clustering algorithms, that group all genes or conditions with similar behavior, biclustering identifies subgroups of genes that exhibit similar patterns of activity over specific subsets of conditions. Biclustering is particularly relevant across biological domains since most cellular processes: i) are controlled by compact groups of genes; and ii) are only active in a restricted set of conditions. Figure 2 shows the difference between subspaces from clustering and biclustering methods. On the left, each gene in a cluster is defined by all the conditions (gene clusters). Similarly, in the middle, each condition in a cluster is represented using all the genes (condition clusters). On the right, and unlike the clustering methods, the goal of biclustering is to identify a group of genes that reveal the same behavior only under a specific subset of conditions by performing simultaneous clustering on both dimensions. Moreover, the genes or conditions from the biclusters can belong to multiple clusters (overlapping regions). Figure 3 instantiates the biclustering notation.

The *homogeneity* criteria defines the degree of similarity between the elements of a subspace in accordance with a cost function or heuristic. The homogeneity criteria determines the coherency, quality and structure of the target biclustering solution. The *coherency* of a bicluster is defined by the observed correlation of values (Def. 2.2). Biclusters can follow dense, constant, additive, multiplicative, plaid or order-preserving coherence, either across rows or columns [17]. The *quality* of a bicluster is defined by the type and amount of accommodated noise. The *structure* is defined by the number (either fixed, parameterizable or variable), size and positioning of biclusters. A flexible structure is characterized by an arbitrary-high number of (possibly overlapping) biclusters. The *statistical significance* of a bicluster measures how its probability to occur deviates from expectations (against a null data model).

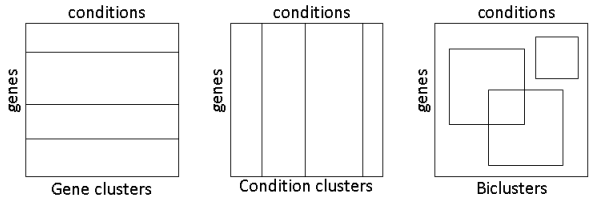

Figure 2: Clusters of a gene expression matrix using clustering (left and center), and biclustering methods (right).

	C1	C2	C3	C4	C5
G1	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}
G2	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}
G3	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}
G4	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}
G5	a_{51}	a_{52}	a_{53}	a_{54}	a_{55}

X = {G1, G2, G3, G4, G5}
Y = {C1, C2, C3, C4, C5}
I = {G2, G3, G4} J = {C2, C3, C4}

Cluster of conditions (X,J) = {C2, C3, C4}
Cluster of genes (I, Y) = {G2, G3, G4}
Bicluster (I,J) = {{G2, G3, G4}, {C2, C3, C4}}

Figure 3: Biclustering terminology in a gene expression data context.

Definition 2.2. Let the elements in a bicluster $a_{ij} \in (I, J)$ have *coherency across rows* given by $a_{ij}=k_j\times\gamma_i+\eta_{ij}$, where k_j is the expected value for column j, γ_i is the adjustment for row i, and η_{ij} is the noise factor. Given a dataset A and a specific coherency strength $\delta \in [0, max_A-min_A]$, $a_{ij}=k_j\times\gamma_i+\eta_{ij}$ where $\eta_{ij}\in[k_j-\delta/2, k_j+\delta/2]$. The γ factors define the coherency assumption: *constant* when $\gamma=1$ and **multiplicative** otherwise ($\gamma\neq1$).

2.2 Accelerators for pattern discovery in gene expression

Despite the necessity of accelerating biclustering algorithms with principles ranging from high performance computing to heterogeneous architectures, only few recent works explore these opportunities [1, 2, 2, 5, 13, 22].

Orzechowski et al. [4] established major principles to consider when developing efficient biclustering algorithms on acpGPU, including efficient access to global memory; coalescing and tiling; reuse of local/shared memory; load balancing between barriers; and avoidance of branches in code. A dedicated biclustering algorithm, well-prepared to mine biclusters with monotonically increasing values on columns, was recently published in accordance with these principles [5].

Arnedo-Fdez et al. [13] gathered initial research evidence on the relevance of using GPUs to accelerate biclustering algorithms in the presence of very large data sets. In their work, the FLOC algorithm – a probabilistic move-based biclustering algorithm which strictly requires biclusters to have a low mean squared residue – is considered. Similarly, Bhattacharya and Cui [1] accelerate biclustering searches based on the largest Condition-dependent Correlation Subgroups (CCS) for each gene in the gene expression dataset.

González-Domínguez and Expósito [2, 3] proposed two versions for accelerating the BiBit biclustering algorithm by exploring the computational capabilities of CUDA-enabled GPUs and modern distributed-memory systems, which provide several multi-core CPU nodes interconnected through a network. Despite its relevance, BiBit is only prepared to discover biclusters from binarized data.

In Liu et al. work [2], an intrinsic parallel architecture with appropriate GPU mapping was applied over the Geometric BiClustering (GBC) algorithm. Three different implementations are assessed, showing a significant speedup of the GPU-based GBC algorithm against a highly optimized CPU-based version.

Despite being pioneer works, the aforementioned GPU-accelerated implementations consider biclustering algorithms that are unable to guarantee the statistical significance of the outputs and show inherent inefficiencies against state-of-the-art biclustering searches [4]. These observations motivate the relevance of extending the studied principles to alternative, state-of-the-art biclustering algorithms

2.3 FPGA-Based OpenCL Accelerators

In the emerging era of multicores, it's necessary to create parallel programs that use all cores simultaneously [8]. Furthermore, it was identified the necessity to create a standard model for designing programs that will execute across different devices like CPUs, GPUs, FPGAs and DSPs. OpenCL [9] is a programming language, based on C programming language, for heterogeneous parallel computing on cross-vendor and cross-platform hardware. OpenCL favors the implementation of parallel algorithms that can be ported between platforms with minimal recoding. OpenCL abstracts low level hardware routines and the devices physical memory, offering execution models to support massively-parallel code execution. The advantage of this abstraction layer is the ability to scale code from general purpose CPUs (Intel and AMD) to massively-parallel GPUs or FPGAs.

OpenCL is based in three models: Platform, Execution and Memory Model [10]. The OpenCL Platform Model consists of a CPU host connected to one or more devices such as CPUs, GPUs or hardware accelerators (e.g. FPGAs and DSPs). Each OpenCL device consists of one or more Compute Units (CU), which are divided into one-to-many Processing Elements (PE), as shown in Figure 4. The host is responsible for identifying the compute devices and selecting the ones that will do the computation.

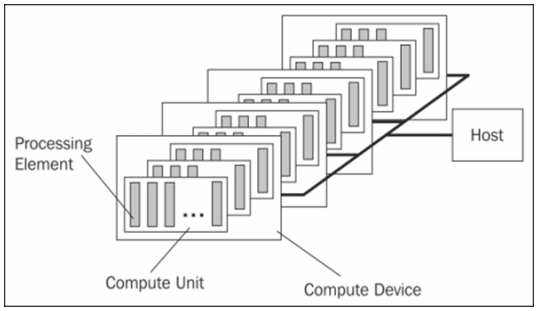

Figure 4: OpenCL Platform Model.

The Platform Model abstracts the way in which OpenCL interacts with the hardware. The relationship between the elements of the platform model and the hardware in a system can be a fixed property of the device, or may be a dynamic feature of a program, depending on how a compiler optimizes the code to best utilize physical hardware. This means that the specific definition of compute units may differ depending on the hardware vendor [12]. For FPGAs, the mapping is more flexible because the number of compute units are assigned by the programmer itself or by the compiler that determines how many resources are needed to implement the hardware circuit of the kernel.

The OpenCL Execution Model comprises two components: OpenCL kernels and host programs. Kernels are the basic unit of executable code that runs on one or more OpenCL devices. Kernels are like C functions that can be data-parallel or task-parallel.

After the host submits a kernel, an index space is created, defining where the function will be executed. The kernel will be executed on each point of the defined index space, which is called *NDRange* and it can be up to three dimensions (x, y, z). The points belonging to the index space running the kernel function are called work-items. Together, work-items are grouped into independent groups called work-groups. The size of the work-groups must be specified by the programmer.

The OpenCL memory model defines how data is transferred between a device and the host computer, and how its stored within a device. This model is defined to abstract the physical configurations of the device memory. Memory in OpenCL is divided into two regions, host memory and device memory. The host memory is directly available to the host, and the data moves from host to devices by calling functions inside OpenCL API. The device memory is directly available to the work-items executing a kernel, within a device. In addition, the memory device is hierarchically composed by four different memory regions:

- Global Memory: Accessible to the host by read and write commands and visible to all work-items in a context;
- Constant Memory: Region of global memory memory that remains constant. This region is read-only for the work-items executing a kernel, but host can read and write from this region;
- Local Memory: Region local to a work-group and only accessible by the work-items of the same work-group;
- Private Memory: Region specific to a work-item, data inside this region is invisible to other work-items.

2.4 QUBIC

Despite the diversity of biclustering algorithms [17], only few are able to place non-trivial homogeneity criteria without hampering efficiency for medium-sized matrices, as well as guarantee the statistical significance of the discovered biclusters [4, 16]. The QUalitative BIClustering (QUBIC) algorithm [4] is one of such cases. QUBIC maps real-valued matrix into an integer matrix, and approximates the underlying value distributions to provide a statistical ground for the biclustering task.

QUBIC is capable of identifying biclusters with coherent patterns known as scaling patterns. These scaling patterns are revealed using the multiplicative model (a_{ij}=k_j×γ_i+η_{ij} in accordance with Def.2.2). Still regarding homogeneity, QUBIC is further able to handle noisy data and allow overlapping areas between biclusters.

To assess the statistical significance of biclusters, QUBIC assumes that the values follow a Gaussian distribution to test their unexpectedness.

QUBIC algorithm starts by quantizing the data matrix to obtain a qualitative representation of the original values. Upper and lower boundaries on the Gaussian distribution are considered to determine what is a differential value. These boundaries can be either defined by the user or dynamically inferred from the observed values. In biological data contexts, these boundaries allow to identify genes that are differentially expressed or proteins and metabolites with unexpected concentrations. If values are within the previously introduced boundary range, then the genes are marked as unchanged. Values that are recognized as changed are marked as up-regulated if they have a positive differential expression, or as down-regulated otherwise. A ranking step is applied for decomposing the changed values into r categories for positive regulation ({1 .. r}) and negative correlation ({-r .. -1}).

After this step, QUBIC tests the similarity between a set of candidate rows (initial seeds) and all remaining rows, where similarity is computed in accordance with the number of columns that satisfy a multiplicative model. To accelerate this step, QUBIC sorts matrix entries to focus on pairs of rows that share a significant amount of columns with differential values. The algorithm then constructs a heavy graph G, where the *nodes* correspond to the data rows (genes), the *edges* connect two rows (genes) and their *weight* is given by the degree of similarity between each pair of rows. Given the fact that only pairwise similarities are considered, G is thus a bipartite graph.

QUBIC proceeds by iteratively adding rows to the initial seeds. This translates into a search problem for all the heavy subgraphs present in the built bipartite graph. By placing a threshold θ on what is considered to be a strong relation between rows, the problem of finding all heavy subgraphs can be mapped into the problem of finding bicliques. The θ parameter is defined by the user and can be varied to control the tolerance to noise.

Despite its inherent advantages, QUBIC is not scalable since the task of identifying bicliques (subset of nodes in which all adjacent nodes are highly connected) is known to be NP-hard. In this context, additional efficiency opportunities need to be explored in order to guarantee that the QUBIC becomes computationally tractable for large matrices.

To address this limitation, the following sections explore and measure the impact of using a dedicated FPGA-based architecture to accelerate QUBIC and similar biclustering algorithms based on discovery of (bi)cliques from adjacency matrices.

3 ALGORITHM PROFILING

In heterogeneous computing, it is necessary to evaluate the balance in the workload between the devices because portions of the accelerated function, so that it can be performed more efficiently.

In this work the `gprof` profiler was used to measure the execution times of each function on the QUBiC baseline implementation available. The analysis performed by the profiler revealed that even when the `-O3` optimization is enabled, the `make_graph` function from QUBIC takes more than 50% of the total execution time, thus making it a good candidate to accelerate in hardware. The output of the profiler for the `-O1` optimization is presented below.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ks/call  Ks/call  name
 72.93    869.74   869.74        1     0.87     1.17  make_graph
 21.29   1123.65   253.90 40094490     0.00     0.00  isInStack
  2.44   1152.74    29.10 20550010     0.00     0.00  intersect_row
  0.62   1160.19     7.45 19990000     0.00     0.00  fh_extractminel
  0.34   1164.22     4.03        1     0.00     0.29  cluster
  (...)
```

4 PROPOSED SOLUTION

This section describes the parallelization process on some parts of the QUBiC algorithm, and the approaches taken for improving its performance. In addition, it is explained how the kernel was implemented using the OpenCL FPGA SDK from Altera.

4.1 Parallelization

The parallelization process targets the `make_graph` function of QUBIC – the most time consuming function in QUBiC according to the `gprof` profiler. The goal of this function is to generate the seeds (pairs of genes strongly correlated across conditions) for the subsequent clustering phase of QUBiC by computing the correlation strength between every pair of genes from the expression matrix. An illustration of this function (based on set-intersections) is represented in Figure 5 .

Figure 5 illustrates the gene comparison, from the left, to the right. On the left, gene 1 conditions are compared against conditions for genes 2, 3 and 4. Therefore, there are 3 comparisons being made on the second loop. The total number of gene comparisons between gene 1 and the rest of the genes is 15. For loading the gene conditions, the kernel performs 20 requests to the FPGA memory, that is, 5 for each gene in the comparison. Afterwards, in the middle, gene

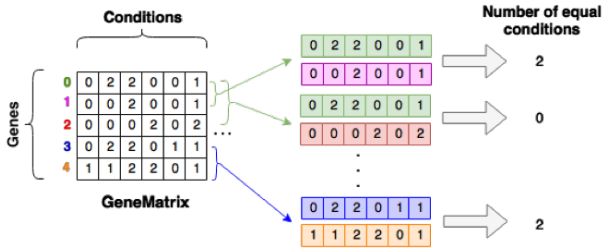

Figure 5: Example of the behavior of `make_graph` function.

2 is compared against genes 3 and 4, with a total of 2 comparisons and 15 load operations. Finally, on the right, only 1 comparison is made, and it is between genes 3 and 4, corresponding to 10 load operations. For this example, the total number of comparisons and load operations are given by equations 1 and 2:

$$NumComputations = \frac{4(4-1)\times 5}{2} = 30 \quad (1)$$

$$NumLoads = (\frac{4(4-1)\times 5}{2} + (4-1)) \times 5 = 30 \quad (2)$$

A score counter stores the number of equal conditions between two genes. After the count, a seed is created with the compared pair of genes and their counter, and then the seed is inserted on a Fibonacci heap only if its counter is greater than a minimum value. The pseudo-code in Figure 7 shows a part of the `make_graph` function.

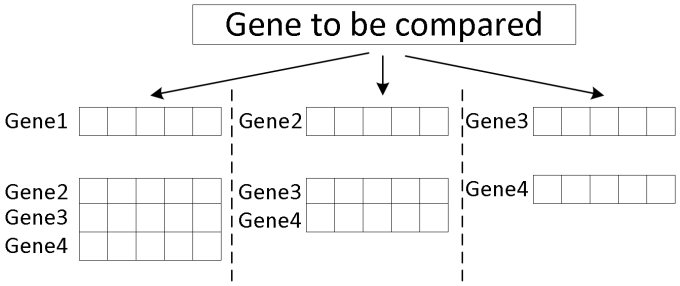

Figure 6: Example of the number of gene comparisons and load operations made in a 4x5 matrix from the make_graph function.

The condition for inserting a seed on the Fibonacci heap, shown on Figure 7, is based on a minimum size threshold, which is increased every time the heap reaches its maximum size (number of nodes in the heap). OpenCL does not allow dynamic memory allocation, imposing the use of a static data structure to store the seeds. Hence, we decided to preserve the Fibonacci heap because it will require a restructuring of the code to accommodate a new data structure (and the corresponding new functions) to maintain the original functionality. Considering the previous OpenCL restriction, and the update on the minimum value when inserting a seed, the portion of the code that counts the number of equal conditions for each gene and the generation of the seed is made in the FPGA,

```
Function Make_Graph(Rows, Columns)
    Data: GeneMatrix
    Result: FibHeap
    for i = 0 to Rows do
        for j =i+1 to Rows do
            for w = 0 to Columns do
                if GeneMatrix[i][w] = GeneMatrix[j][w] and
                 GeneMatrix[i][w] ≠ 0 then
                    counter++;
                end
            end
            if counter > minimum then
                seed = {i, j, counter};
                put seed in FibHeap;
            end
        end
    end
end
```

Figure 7: Pseudo-code of Make_graph function.

while the verification for inserting the seeds on the heap is done by the CPU.

Following the work-balance between the host and device, it is important to define the number of comparison operations done by the `make_graph` function before addressing the communications between the devices. The number of comparison operations taken by the `make_graph` function is defined by the following expression:

$$\frac{N\times(N-1)}{2}\times M \quad (3)$$

In Equation 3, N and M are the genes and conditions from the input matrix, and $(N\times(N-1))/2$ represents the number of genes pairs compared, resulting from the two outer loops in Figure 7. The $(N-1)$ means that the current gene to be compared does not compares with himself, resulting from the initialization of the second loop. This component is divided by two since the values of previous comparisons involving the same pair of genes are preserved.

A key aspect in parallelizing algorithms across different computational units is data transfers. Communications and synchronization between these units represent a bottleneck for getting a faster parallel implementation. The objective is to minimize the number of communications.

Hence, in the `make_graph` function, to reduce the communications, only two transfers are made. The first one is to send the matrix from the host, containing all genes and conditions, and the second one is to receive the seeds from the FPGA. Furthermore, Equation 3 calculates how many load requests are made. The number of load operations performed in the `make_graph` kernel to the FPGA memory is calculated by the following formula:

$$\left(\frac{N\times(N-1)}{2} + (N-1)\right)\times M \quad (4)$$

where: N and M are the genes and conditions from the input matrix, and $(N\times(N-1))/2+(N-1)$ represents the number of genes loaded

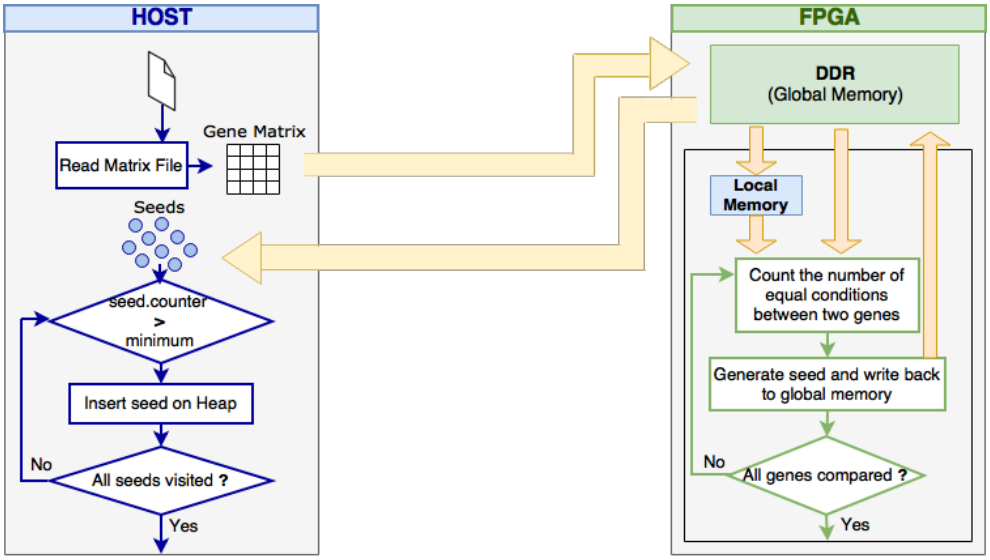

Figure 8: The program flow between the CPU and FPGA device.

from the memory. An example of the number of gene comparisons and load operations is shown in Figure 6.

Another key aspect when improving programs for better performance is examining how the data is accessed. The goal is to optimize the memory accesses by considering the data locality and reuse. Although, there's a need to consider the trade-offs between the overhead of move the data into local memory. So, in cases where reads/writes are made frequently at the same location, data can be reused and benefit from a better place to store it for faster accesses.

For the case of `make_graph` function, accessing the global memory for every two genes to be compared has high-latency compared to on-chip memory. To overcome this overhead, the gene to be compared with the rest of the genes was stored in the local memory, because this gene doesn't change on every iteration. The program flow, between the CPU and FPGA device, is illustrated in Figure 8.

In heterogeneous systems, the memory, on the device (FPGA), is made of banks (hardware units). Successive data elements are placed in adjacent banks to avoid bank conflicts when a group of threads access consecutive elements. If multiple accesses occur on the same bank, then the bank with most conflicts imposes the latency. Memory accesses patterns have a great impact on the memory bus utilization, and low utilization leads to low performance. The goal is to have high-throughput instead of low-latency by making coalesce memory accesses and avoiding bank conflicts. Coalescing the memory accesses optimizes the data transfers to/from the global memory, which translates into less contention on this resource.

The memory access pattern of the QUBiC algorithm favors the coalescence because all the conditions are needed and accessed in a successive way, also the values are stored in contiguous memory positions, on the same bank of memory. To reduce the number of requests to the memory, multiple columns are transferred with one request (burst-coalesced request) instead of one condition per request.

The number of columns received per request depends on the level of parallelization defined, and how many data elements fit into one word from global memory. The same is applied when generating a seed after the counting of equal conditions. When a seed is generated, containing the two genes and the score (number of equals conditions), it is wrapped into a vector containing the three fields. This type of data vector leads to a single write on the memory instead of three.

OpenCL programming model offers two types of parallelism, data and task parallelism. For the data parallel approach, multiple work-items, or threads, ran over the dataset and execute the same operation. This SIMD-based parallel processing is used for loops without data dependencies through the loop iterations. However, if the loop contains data dependencies, makes it difficult, or even impossible, to parallelize it in a data parallel fashion because involves sharing resources by the work-items and additional synchronization methods. So, instead of running multiple work-items on the data, it is possible to have a single work-item called OpenCL task to deal with data-dependence. Although, the performance is affected due to the number of work-items decrease.

Pipeline-parallel architectures can exploit the parallelism by running parallel instructions in a single work-item through the pipelining of loop iterations (Pipeline Parallelism). Loop pipelining can execute each iteration at every clock cycle if there are no dependencies between iterations. The launch of a new iteration per clock cycle is called Initiation Interval (II). If the II is optimal (II=1), then all the iterations are overlapped and execute the same way as launching multiple work-items (parallel threads) in a pipelined fashion, as it is shown in Figure 9.

When the II is above one, then it takes more clock cycles for an iteration to start. This happens because the next iteration depends on the result of the previous iteration, also the latency associated with the computation of the results from one iteration can affect even more the II.

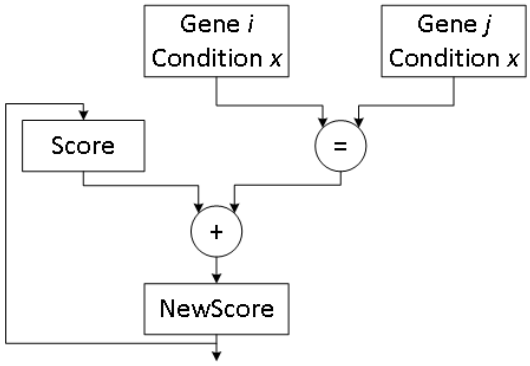

Figure 9: Kernel with no loop unroll optimizations.

Regarding the `make_graph` function, the kernel is executed using the OpenCL task because of the dependency over the current score counter. The counter accumulates the number of equal conditions between two genes at every iteration.

4.2 Implementation Details

This section describes the platform and the techniques used in the Intel OpenCL FPGA SDK to make kernel functions, and what modifications were made on the baseline code to accommodate the new parallel implementation.

The hardware platform used is a DE5-Net board from Terasic, with a Stratix V GX (5SGXEA7N2F45C2) FPGA, which includes 4GB of DDR3 memory. In addition, the FPGA has 622k logic elements, 50-Mbit of M20K memory blocks and 256 DSPs. The board communicates with the CPU via an 8x PCI-e interface. The host machine is composed of a 3.4GHz i5-3570 (4 Cores) Intel Core processor with 16 GB of RAM. The data transfer performance for this FPGA board is summarized in Table 1.

Measure	Speed [MB/s]
Write	2187.3
Read	2977.6
Throughput	2582.4

Table 1: Different data transfer speeds between the host computer and the DE5 Net FPGA board.

To take full advantage of the Direct Memory Access (DMA), data needs to be 64-bit aligned from the host, so that the memory transfers between devices are more efficient (they are sent in bursts, through PCI-e). If the data is non-aligned, then it increases the latency into the data transfers because the DMA is not used.

To improve the performance of the kernel, the loop unroll technique was used to decrease the number of iterations of the kernel execution. The goal of this technique goal is to reduce the loop overhead by eliminating/minimizing the loop control instructions that are made in each iteration. For accomplish the goal, the loop body is replicated, which comes with the penalty of having more hardware resources derived from expansion of the loop pipeline. Also, according to the number of iterations unrolled, the number of gene comparisons and load operations increases per clock cycle. Figure 10 shows an example of the kernel architecture using the loop unroll optimization.

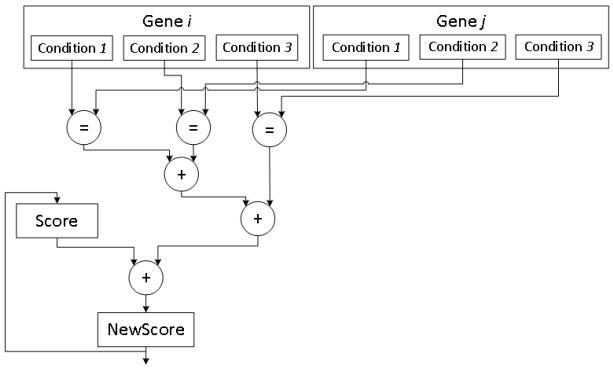

Figure 10: Kernel pipeline when using the loop unroll technique by three iterations.

Within the kernel of the `make_graph` function, the loop for caching the gene to be compared over the N-1 genes, and the loop for counting the equal conditions between genes were unrolled for higher throughput. Nevertheless, with increase on the number of memory accesses, the load operations are coalesced to improve the performance, by loading more input data with one/less requests.

In OpenCL, to unroll a loop, the unroll bounds need to be constant or the unrolling factor must be explicit. For this reason, a fixed number of columns to be compared was defined, that also needs to be equal to the unroll factor.

Associated with the constant unroll factor and to unroll the exact number of iterations, it was defined a fixed size for the number of conditions computed in parallel per iteration. This way, the block size of conditions is equal to unrolling factor. Also, the number of conditions in the input matrix needs to be multiple of the unrolling factor chosen. In the host side was implemented a function that rounds the number of conditions in the matrix to a multiple of the unroll factor number, which will lead to a minor overhead because of the extra computation for the additional conditions.

Another improvement was made regarding the baseline code with respect to the "if" clause present in the original code. The if clauses were removed and transformed into a combinational logic function that stores the result of an AND operation between the two boolean algebra formulas.

5 RESULTS

This section describes the results obtained from the heterogeneous and parallel implementation of the `make_graph` function using the Intel FPGA OpenCL SDK and a single FPGA board.

The kernels were evaluated on a DE5-Net FPGA board from Terasic. The parallel version of the `make_graph` function, running on the FPGA, is compared against the serial version running on the CPU and compiled with the `-O3` optimization flag. The performance results were obtained from measuring the execution times and calculating the speedups for different inputs and loop unroll factors.

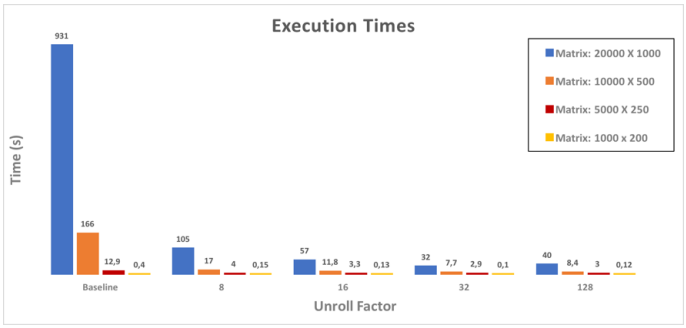

Figure 11: Execution times of the baseline code and `make_graph` kernels with different unroll factors.

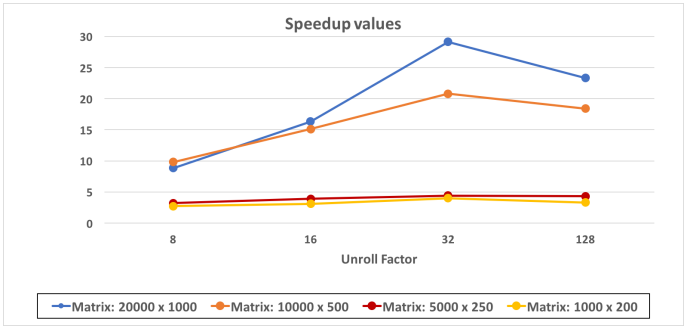

Figure 12: Speedups of the `make_graph` function with different unroll factors.

To this end, we generated multiple synthetic datasets resembling gene expression data using the biclust package from R. We varied the number of genes in these artificial matrices up to twenty thousand to guarantee that each input matrix and the associated QUBIC's seeds fit into the FPGA DDR memory.

The seeds generated by the kernels of the `make_graph` function correspond to the same seeds produced by the baseline QUBIC implementation, thus guaranteeing the correct comparison of the original and enhanced algorithms.

From Figure 11, it can be observed that the parallel execution times are lower than the baseline implementation for different inputs sizes. For the largest input size it achieved an execution time 29x inferior compared to the baseline code. In Figure 12 , by analyzing the speedup values it is possible to conclude that for maintaining/improving the performance with the input size growth, it is necessary to increase the hardware resources of a circuit to accommodate more operations per clock cycle.

To evaluate the impact of the growth in the number of genes and conditions, it was made multiple input matrices with fixed sizes on each dimension of the matrices. Therefore, the Figure 14 shows that the execution times are more affected by the increment of the number of conditions than the increase of genes.

The results from Figure 12 also indicate a performance degradation for an unroll factor greater than 32. When loop unrolling technique is used, the number of hardware resources utilized growths with the increment of the unroll factor, which leads to bigger circuit.

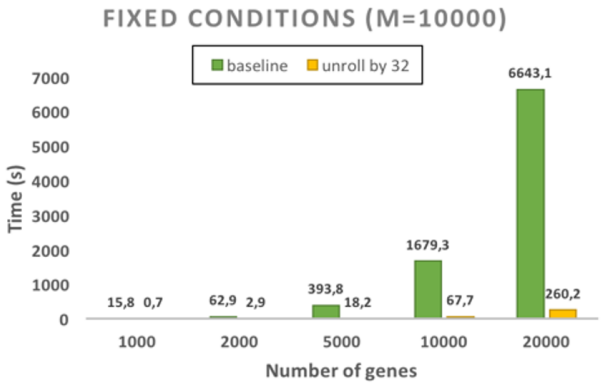

Figure 13: Comparison between the executions times of baseline code and a kernel with 32 unroll factor for multiple matrices with fixed size on both dimension.

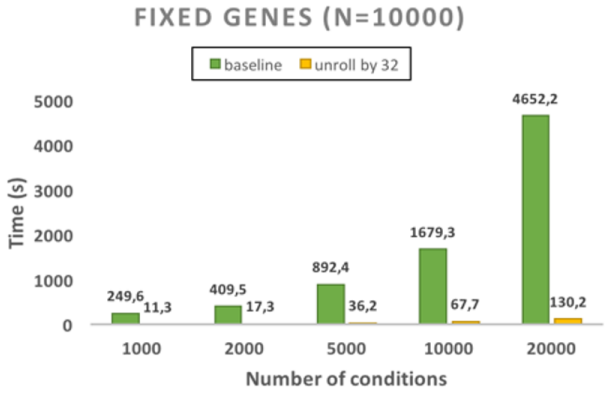

Figure 14: Comparison between the executions times of baseline code and a kernel with 32 unroll factor for multiple matrices with fixed size on both dimension.

Nevertheless, an increase in the circuit area can potentially have a negative impact on the clock frequency. When the circuit area expands, the longest combinatorial path between flip-flops, RAM, and I/O also increases. The increase of this path known as "critical path" is one of the bottlenecks of a system, because will take more time to propagate the signals through the circuit.

From Table 2, between the loop unroll factor of 32 and 128, there's an increase of 7% in ALUTs, 3% FFs and 2% of RAM usage, leading to a decrease of 58,7 MHz in the clock frequency.

Another aspect that needs to be considered when using the loop unroll technique is the required memory bandwidth to maintain the computation resources fully occupied. Increasing the loop unroll factor value will require more memory bandwidth because there are more load/store operations.

The algorithm data access pattern can cause a bottleneck, if the requests exceed the available global memory bandwidth, therefore the memory bandwidth utilization was analyzed for different unroll factors.

Unroll Factor	ALUTs	FFs	RAMs	DSPs	Clock [MHz]
32	73872 (16%)	89024 (9%)	531 (21%)	4 (2%)	244.9
128	106217 (23%)	108466 (12%)	579 (23%)	4 (2%)	186.2

Table 2: Circuit area usage and clock frequency for two kernels with different unroll factors.

Unroll Factor	Load conditions BW [MB/s]	Occupancy	Clock Frequency [MHz]
8	3977.7	99 %	250.8
32	15182.2	96 %	244.9
128	11885.1	25 %	186.2

Table 3: Circuit area usage and clock frequency for two kernels with different unroll factors and an input matrix of size 20000x1000.

Table 3 was obtained from the kernel profiling report, it reveals the bandwidth used for three different unroll factors and an input matrix size of 20000 genes by 1000 conditions. In this table, the parameters are:

- Unroll factor - degree of loop parallelization;
- Load conditions - total bandwidth used from global memory to load the conditions (DDR memory);
- Occupancy - percentage of the profiled time the kernel pipeline is full
- Clock frequency - clock frequency that the kernel operates

From the results in Table 3 it is possible to conclude that the kernel with the unroll factor of 32 is the best one as it uses the available bandwidth of the FPGA global memory. The bigger difference relies on the occupancy percentage between 32 and 128 unroll factors. In the 128 unroll factor, the high amount of parallel memory accesses makes the loop-iterations to not enter the pipeline consecutively due the insertion of bubbles into the pipeline. A decrease on the kernel clock frequency with a low occupancy reveals a drop in the frequency of requests to the memory, which causes the bandwidth to decline as demonstrated by the results of Table 3.

The 32 unroll factor value is optimal because it fully saturates the memory bandwidth. In the case where the unroll factor surpasses this optimal value it over saturates the memory bandwidth, then the kernel performance degrades due to the high contention on the memory resources.

6 CONCLUSION

This work presented a new parallelization architecture and the corresponding OpenCL implementation to accelerate clique-based biclustering algorithms. To this end, we considered different hardware/software acceleration strategies using an heterogeneous system composed by a FPGA and a CPU. These acceleration principles were applied over the QUBIC algorithm, given its state-of-the-art stance regarding effectiveness and efficiency. By incorporating parallelization principles in one of the most computational intensive parts of the algorithm, significant performance improvements were observed. The collected results stress the relevance and potential of the proposed contributions for biclustering high-dimension biological data.

ACKNOWLEDGMENT

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013. The authors would like to thank Altera University Program for the donation of the FPGA board.

REFERENCES

[1] Oghabian, A., Kilpinen, S., Hautaniemi, S., Czeizler, E.: Biclustering Methods: Biological Relevance and Application in Gene Expression Analysis. PLoS one 9(3), e90801 (2014).
[2] Liu, B., Xin, Y., Cheung, RC., Yan, H.: GPU-based biclustering for microarray data anal-ysis in neurocomputing. Neurocomputing 134(0), 239-246 (2014).
[3] Cheung, K., White, K., Hager, J., et al. YMD: a microarray database for large-scale gene expression analysis. In: Proceedings of the American Medical Informatics Association, pp. 140-4. Hanley and Belfus, Inc., San Antonio, Texas (2002).
[4] Li, G., Ma, Q., Tang, H., et al.: QUBIC: a qualitative biclustering algorithm for analyses of gene expression data. Nucleic Acids Res 37(15), e101 (2009).
[5] Madeira, S., Oliveira, A.: Biclustering Algorithms for Biological Data Analysis: A Survey. IEEE/ACM Trans. Computational Biology and Bioinformatics, vol. 1, 24-45 (2004)
[6] Hartigan, J.A.: Direct clustering of a data matrix. J. Am. Stat. Assoc. (67), 123-129 (1972).
[7] Cheng, Y., Church, G.M.: Biclustering of Expression Data. In: Proc. Eighth Int'l Conf. In-telligent Systems for Molecular Biology (ISMB '00), pp. 93-103, 2000.
[8] Altera: Implementing FPGA Design with OpenCL Standard. White Paper, Intel Altera, 2013
[9] Munshi, A., Gaster, B., Mattson, T., Fung, J., Ginsburg, D.: OpenCL Programming Guide. 1st edn. Addison-Wesley Professional, 2011.
[10] The Khronos Group Inc: The open standard for parallel programming heterogeneous sys-tems (OpenCL). Specification v2.2, 2016.
[11] AMD: Training guide: Introduction to OpenCL programming, 2010
[12] Tompson, J., Schlachter, K.: An Introduction to the OpenCL Programming Model. Digital version, 2012.
[13] Arnedo-Fdez, J., Zwir, I., Romero-Zaliz, R.: Biclustering of very large datasets with GPU technology using cuda. In: Proceedings of V Latin American Symposium on HPC, 2012
[14] R. Henriques, C. Antunes, and S. C. Madeira, "A structured view on pattern mining-based biclustering," *Pattern Recognition*, 2015.
[15] R. Henriques and S. C. Madeira, "Bicpam: Pattern-based biclustering for biomedical data analysis," *Algorithms for Molecular Biology*, vol. 9, no. 1, p. 27, 2014.
[16] ——, "Bsig: evaluating the statistical significance of biclustering solutions," *Data Mining and Knowledge Discovery*, 2017. [Online]. Available: https://doi.org/10.1007/s10618-017-0521-2
[17] S. C. Madeira and A. L. Oliveira, "Biclustering algorithms for biological data analysis: A survey," *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, vol. 1, no. 1, pp. 24–45, Jan. 2004.
[18] A. Freitas, W. Ayadi, M. Elloumi, J. Luıs, and J.-K. H. Oliveira, "Survey on biclustering of gene expression data," *Biological Knowledge Discovery Handbook*, pp. 591–608, 2012.
[19] K. Eren, M. Deveci, O. Küçüktunç, and Ü. V. Çatalyürek, "A comparative analysis of biclustering algorithms for gene expression data," *Briefings in bioinformatics*, vol. 14, no. 3, pp. 279–292, 2013.
[20] M. Charrad and M. B. Ahmed, "Simultaneous clustering: A survey," in *Pattern Recognition and Machine Intelligence*. Springer, 2011, pp. 370–375.
[21] K. Sim, V. Gopalkrishnan, A. Zimek, and G. Cong, "A survey on enhanced subspace clustering," *Data Mining and Knowledge Discovery*, vol. 26, no. 2, pp. 332–397, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10618-012-0258-x
[22] E. Mejía-Roa, C. García, J. I. Gómez, M. Prieto, F. Tirado, R. Nogales, and A. Pascual-Montano, "Biclustering and classification analysis in gene expression using nonnegative matrix factorization on multi-gpu systems," in *Intelligent Systems Design and Applications (ISDA), 2011 11th International Conference on*. IEEE, 2011, pp. 882–887.

REFERENCES

[1] Anindya Bhattacharya and Yan Cui. 2017. A GPU-accelerated algorithm for biclustering analysis and detection of condition-dependent coexpression network modules. *Scientific Reports* 7, 1 (2017), 4162.

[2] Jorge González-Domínguez and Roberto R Expósito. 2018. Accelerating Binary Biclustering on Platforms with CUDA-enabled GPUs. *Information Sciences* (2018).
[3] Jorge González-Domínguez and Roberto R Expósito. 2018. ParBiBit: Parallel tool for binary biclustering on modern distributed-memory systems. *PloS one* 13, 4 (2018), e0194361.
[4] Patryk Orzechowski and Krzysztof Boryczko. 2015. Effective biclustering on GPU-capabilities and constraints. *Prz Elektrotechniczn* 1 (2015), 133–6.
[5] Patryk Orzechowski, Moshe Sipper, Xiuzhen Huang, and Jason H Moore. 2018. EBIC: an evolutionary-based parallel biclustering algorithm for pattern discovery. *Bioinformatics* (2018).