UNIVERSITEIT TWENTE.

Electrical Engineering
Dependable Integrated Systems MSc

FPGA Acceleration of a hierarchical clustering and data fusion application for disease subtype discovery

Master thesis

Examination Committee
Dr. Ir. Nikolaos Alachiotis
Dr. Ir. Marco Gerards
Dr. Ing. Anastasia Lavrenko

Student
Andrei-Octavian Voicu-Spineanu

Year 2021

Table of Contents

List of figures

List of tables

List of acronyms

- HC – Hierarchical Clustering
- TCGA – The Cancer Genome Atlas
- HLS – High-Level Synthesis
- FPGA – Field-Programmable Gate Array
- RTL – Register-Transfer Level
- OS – Operating System
- GBM – Glioblastoma multiform
- KIRC – Kidney renal clear cell carcinoma
- COAD – Colon adenocarcinoma
- LIHC – Liver hepatocellular carcinoma
- SKCM – Skin cutaneous melanoma
- OV – Ovarian serous cystadenocarcinoma
- SARC – Sarcoma
- AML – Acute myeloid leukemia
- BIC – Breast cancer
- DNA – Deoxyribonucleic acid
- RNA – Ribonucleic acid
- SRAM – Static Random-Access Memory
- SDRAM - Synchronous Dynamic Random-Access Memory
- URAM – Unified Random-Access Memory
- BRAM – Block Random-Access Memory
- FF – Flip-Flop
- LUT – Look-up table
- DSP – Digital Signal Processor

1. Introduction

This document represents the final report of my Master thesis which concludes the 2 years Master program at University of Twente, Dependable Integrated Systems Master of Sciences.

The main subject of this thesis represents a new method of integrative clustering for disease subtyping of cancer patients, which is based on recent advancements in multi-omics clustering. According to Pfeifer and Schimek [1] this method's behavior has the potential of aiding in cancer progression and eventually treatment, due to the advantage of individual contribution of each omic to the data fusion process. This technique, called *HC-fused,* alongside other state-of-the-art data integration methods, were applied to data sets from TCGA (The Cancer Genome Atlas) [2]. *HC-fused* proved superior performance to some types of cancer, best results being obtained for KIRC (kidney renal clear cell carcinoma) and SARC (sarcoma).

Based on information from Pfeifer and Schimek [1] and Nie et al. [3] one goal of integrative analysis of datasets is to involve features from different sources. Each source (e.g. DNA sequence or RNA expression) comes with its own specific properties which may result in a more effective clustering performance. By working with these different sources of data, heterogeneous datasets are being formed. However, the clustering methods which act upon these datasets present a major challenge due to their inadaptability to large scale multi-view data. This challenge can be observed in the numerous computations that are needed to perform the integrative clustering algorithms, which are completed in a very long execution time.

1.1 Motivation

The goal of this project is to **improve the overall execution time of the *HC-fused* algorithm** through the implementation of a faster C++ code which will be further used in designing an efficient and compatible hardware using High-Level Synthesis which will decrease even more the execution time of the algorithm.

Through the application of *HC-fused* or other similar algorithm, Pfeifer and Schimek [1] state that the cancer progression of patients may be better understood and thus disease subtypes for each individual patient will be highlighted and eventually better treatments can be found. One major advantage that this method presents is the individual contribution of each omic (such as DNA or RNA) to the final output of the method – a fused network which integrates the similarities of patients into clusters. It has been observed that a much better and conclusive clustering result is yielded by an increased number of algorithm iterations, 10 iterations being the minimum amount and 100 being the recommended one. As a consequence a long time is taken by the numerous and repetitive computations from inside the *HC-fused* method. This time also adds up with the relatively slow performance that *R* programming language has to offer. Additionally, the way the algorithm is able to scale with the number of patients and the number of omics will also be assessed.

The **motivation** of this work is to overcome the algorithm's slow execution to enable more iterations per method run, a better clustering performance and further, a more accurate disease subtype discovery. In order to perform this, the slow execution and poor scalability with the number of patients and omics problems will be addressed. The overall goal is accomplished by optimizing the *HC-fused* algorithm by means of both software and hardware techniques. The initial target R code, which is presented by Pfeifer and Schimek in [1] will be adapted, translated to C++ and further used in designing

a hardware via high-level synthesis, capable of improving even more the efficiency of the method. The final result should present a major improvement in execution time, while also being able to scale accordingly with the number of patients and the number of omics.

Although R represents an efficient programming language for data analysis and statistics, it still has the disadvantage of being rather slow in execution time. Converting the algorithm to C++ and applying a series of optimizations should indicate a large improvement in terms of execution time, the C++ compilers being able to translate the code directly into machine code. As a consequence, this project will report on the potential performance of porting the R code to C++. Additionally, it was observed that the R code also scaled poorly with the input data types. Increasing the number of patients or the number of omics from the network caused a substantial growth in the execution time, making it inefficient to test large datasets. Due to this, this report will assess the scaling efficiency of the C++ method with the number of patients and data types. Furthermore, once it is sensed that the software optimizations no longer can greatly affect the execution speed of the algorithm, the C++ code will be used in developing a hardware accelerator via an FPGA using High-Level Synthesis. Finally, the impact of the hardware acceleration on the execution time will be measured and discussed.

1.2 Scientific contribution

The *HC-fused* [1] hierarchical clustering and data fusion algorithm's performance was improved during this project. The software optimizations that were implemented after the translation of the method to C++ have greatly decreased the execution time. The two datasets that were fed to the algorithm for testing were made of 105 patients with 2 types of omics and 849 patients with 3 types of omics respectively. Running the latest and most improved version of the method showed an execution time 784 times faster for the smaller dataset and 3384 times faster for the larger one. The difference between the two results indicate that the scaling of patients and number of omics is no longer considered a problem, based on the tests.

Using an adapted version of the fastest C++ implementation, an accelerator hardware architecture was designed, capable of improving the execution time of the method even more. Prior to the design of the hardware, the profiling process has indicated the two functions that occupy the majority of execution time for the algorithm. Based on these results and on further considerations, three different hardware designs were being created, the user having the possibility to use them individually as separate IP cores. Running the algorithm using these accelerators for the designated parts should show great benefits in terms of efficiency. The first accelerator helped in speeding up the computation of the Similarity Matrix by 1.36 times, while the other two were used in accelerating the extraction of indexes from the Similarity Matrix. The resulting estimated time was added up for these two designs since only their combined functionality can be compared with the software one, the acceleration ratio being for this case 1.5 times faster. By combining the software and hardware results, for the dataset of 105 patients and two omics, the algorithm now executes 10 iterations 965 times faster.

1.3 Thesis organization

This chapter offered a small overview of the thesis premises and results, indicating the scope of the project, assessed research questions and main improvements implemented at the end of working period. The Chapter 2 presents the starting point of this project – the *HC-fused* algorithm. It will cover the workflow of the method alongside preliminary results and comparisons with other similar methods of this type. Furthermore it revises the necessities of the algorithm and the potential promising benefits

that it is capable of. Chapter 3 will shortly go over projects similar to this one, offering a broad view of the current computational stage and the results that can be obtained using some hardware techniques. It will offer short overviews of the method's applications and results in terms of either timing, resources or both. Chapter 4 will fully describe the software optimizations that were implemented and the results in time for each stage. This section also includes the interface between the initial R algorithm and the hardware platform, C++ being compatible with both of them. The fastest version is being further used in HLS in designing the hardware capable of accelerating the algorithm. Furthermore, chapter 5 describes the hardware architecture that was chosen in the design process, presenting reasons and potential benefits for the selected hardware. It includes the profiling results of the code which show the most time-consuming functions and the additional reasoning for the developed architecture. Chapter 6 firstly presents the software implementation which describes the used softwares and libraries and then continues with the hardware implementation that show the capabilities of HLS and the workflow of Vitis HLS. Chapter 7 offers a full overview of the obtained results from the software and the hardware optimizations and lastly, chapter 8 discusses the entire project and its results, offering guidance for future work, such as potential improvements and useful applications of the algorithm and the design.

1.4 Research questions

- R language represents an extremely viable option for bioinformatics applications that deal with statistics and data analysis. It presents a lot of advantages such as being open-source, having numerous available packages and libraries, many plotting options and more. On the other hand it is slow due to its expressions being interpreted at runtime, instead of being compiled and translated into machine code. In addition to that, existing frameworks for High-Level Synthesis target languages such as C, C++ and C#. Based on this, the first questions that raises is:

  What is the performance potential of porting the R code to C++?
- The algorithm written in R proved to scale poorly with the number of patients and the number of omics included in the input dataset. Tests have showed that increasing these numbers affected greatly the execution time of the algorithm, making it inefficient for large datasets. Due to this issue, the second research question is formulated:

  How does the C++ code version of the algorithm scale with the number of patients and the number of omics?
- Having the algorithm written in C++ allows for a design of a hardware capable of speeding up the algorithm even more by using High-Level Synthesis. At some point the software optimizations will reach a certain saturation where the algorithm will no longer be able to have significant improvements in terms of execution time. One viable option to overcome this is the use of dedicated hardware accelerators which allow the execution in parallel of the code and the efficient usage of the available resources. This project explores the potential of FPGA as an accelerator technology. As a result the next research question is constructed:

  What impact does the hardware acceleration via an FPGA have on improving the execution time of the clustering algorithm?

2. Background

This chapter will present an overview of the hierarchical clustering and data fusion algorithm and its functionality and performance. It will start with describing each major step from the method and it will indicate an example of a final outcome of the method. Furthermore, this chapter will additionally offer a short example with a dataset containing only 5 patients and 2 omics, with states being illustrated after each merge of clusters.

2.1 HC-fused algorithm

The method of Bastian Pfeifer and Michael G. Schimek [1] allows the grouping of cancer patients into relevant clusters based on characteristics such as proteins and genes. This clustering may aid into discovering a new perspective on how the cancer evolves for these patients and how they should receive a better treatment. One of the main challenges of this kind of methods is to integrate as many different data types, usually from the TCGA (The Cancer Genome Atlas) and use them as input to these algorithms in order to get a better insight. However the usage of several omics types comes with the cost: an intensive computation. This requirement naturally has to be overcome as through the means of integrative clustering, so by clustering patients using data from different biological areas, some more complex disease subtypes may be discovered.

The algorithm called *HC-fused* by Pfeifer and Schimek [1] was written in R – an open source programming language, similar to Matlab and Python, generally used in statistical modelling and analysis and capable of providing several packages, plotting options and data wrangling facilities. *HC-fused* is one of the algorithms that works with different types of data, each type contributing to the final outcome of the method. More than that, the main advantage of this specific algorithm is that each individual view is taken into consideration in the data fusion process. This means that a greater number of views will provide a better and more exact clustering and fusion between patients.

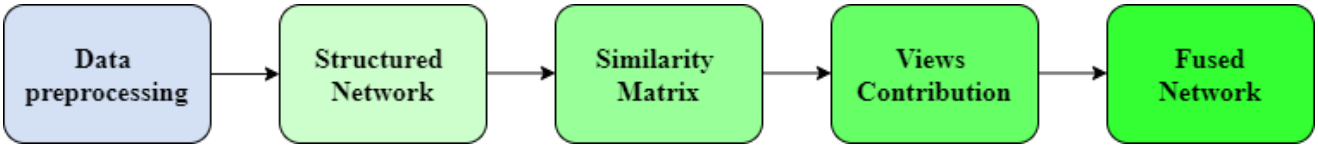

Figure 2. 1 HC-fused algorithm divisions

The first part of the algorithm consists of the **Data preprocessing**. Here data is normalized and imputed in order to add missing values across all patients. Patients and features with a percentage of more than 20% missing data are being removed at this stage. The remaining ones are being imputed using the k-Nearest Neighbor algorithm. Here data from TCGA [2] is being read from files and converted via a series of functions from Pfeifer and Schimek [1] which are included in the acceleration plan of this project. The result of this sequence of functions is the Structured Network which holds all the necessary information needed in clustering and fusing the data.

The **Structured Network** represents the main input to the *HC-fused* algorithm that needs the acceleration. It is structured as a binary matrix with dimensions based on the number of omics and the number of patients and it is obtained by performing a series of 3 steps:

- Transform each view into a connectivity matrix using Ward's hierarchical clustering algorithm, according to Murtagh and Legendre [4].
- Infer best number of clusters using the *Silhouette Coefficient* – a method used to calculate the correctness of the performed clustering, as described by Aranganayagi and Thangavel [5]
- The resulting binary matrices are multiplied element-wise, the result being stored into a matrix that indicates the connectivity between patients.

An example of such a structured network having a number of two omics and two patients is illustrated in Figure 2.2:

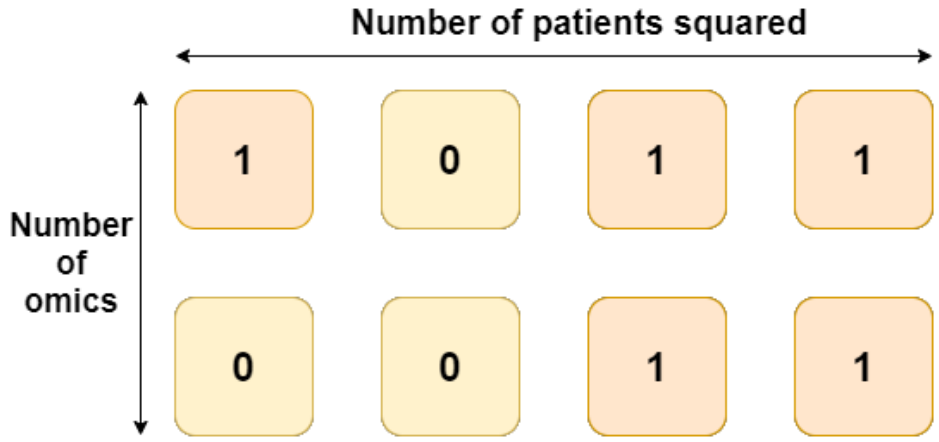

Figure 2. 2 Structured network with two omics and two patients

For generating the **Similarity matrix**, the binary matrices from the previous step are used in a bottom-up hierarchical clustering method, alongside the forming clusters of patients from this algorithm. Each iteration merges two clusters (initially these clusters have only 1 patient) based on a minimal distance calculated at this point. Afterwards this matrix is normalized. This similarity matrix can also be used in other hierarchical clustering algorithms besides the one used in this method. The document from Pfeifer and Schimek [1] also provides an insightful pseudocode for generating the **Similarity matrix**. In the pseudocode, G_i represent the network views, d_i comprise the distances between clusters for each view, S corresponds to the source matrix which indicates the contribution of each view and lastly, P signifies the fused similarity matrix. The entire pseudocode can be observed in *Figure 2.3*.

```
Given the network views G_1, G_2, ..., G_l, G_∧;
#cluster = #patients;
k = #cluster;
while k ≠ 0 do
    d_1 = dist(c_1, c_2|G_1);
    d_2 = dist(c_1, c_2|G_2);
    ⋮
    d_l = dist(c_1, c_2|G_l);
    d_∧ = dist(c_1, c_2|G_∧);
    if min(d_1, d_2, ..., d_l, d_∧) == d_∧ then
        d_min = d_∧;
        fuse(c_1, c_2|G_min, d_min);
        S_min(i, j ∈ (c_1 # c_2)) ++;
    else
        d_min = min(d_1, d_2, ..., d_l) ;
        fuse(c_1, c_2|G_min, d_min);
        S_min(i, j ∈ (c_1 # c_2)) ++;
    end
    P((i, j) ∈ C) ++;
    k − −;
end
```

Figure 2. 3 Similarity matrix pseudocode

(Figure from Pfeifer and Schimek [1]*)*

Each **View** represents an outcome state of the fusing network per method iteration and further indicates how many times a patient has been used in the fusion process. It is important to mention that a minimum of 10 iterations is recommended for the algorithm due to the random factor that appears in case of equal minimal distances. For each iteration, each view contribution is being tracked.

The **Fused Network** represents the final and main outcome of the *HC-fused* algorithm. An example of a *Fused Network* is displayed in *Figure 2.4*.

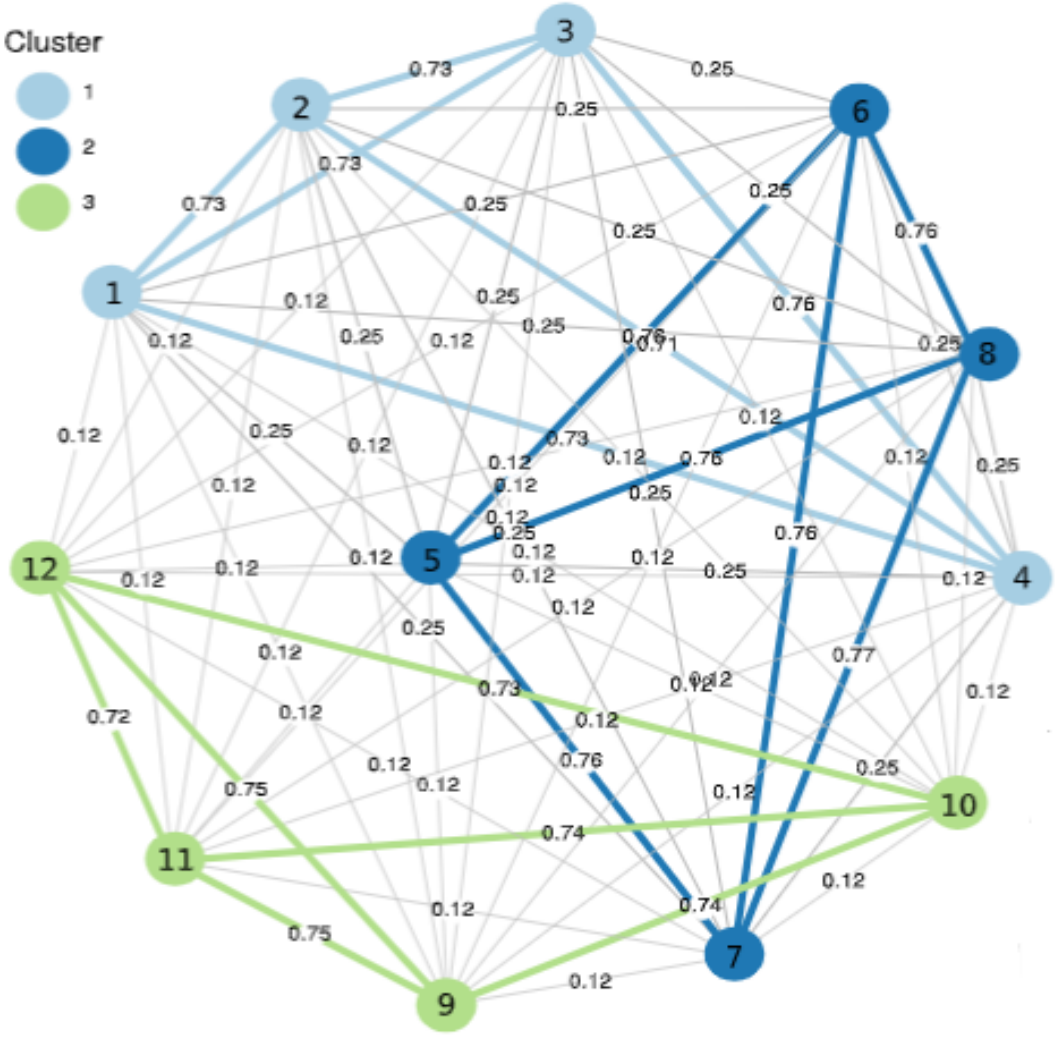

Figure 2. 4 Fused network example

(Figure from Pfeifer and Schimek [1]*)*

The above picture presents a fused network containing a total of 12 patients represented as nodes. Each color indicates a different cluster for a total number of 3 clusters in this case. The edges between nodes indicate the minimal distances between patients computed in the Similarity Matrix.

The *HC-fused* method was tested on 9 types of cancer. These are: glioblastoma multiform (GBM), kidney renal clear cell carcinoma (KIRC), colon adenocarcinoma (COAD), liver hepatocellular carcinoma (LIHC), skin cutaneous melanoma (SKCM), ovarian serous cystadenocarcinoma (OV), sarcoma (SARC), acute myeloid leukemia (AML), and breast cancer (BIC). From these types, according to Pfeifer and Schimek [1], *HC-fused* proved superior performance and results for KIRC, LIHC, SKCM, OV and SARC types. During tests, multi-omics data from TCGA was applied to the algorithm which efficiently took advantage of the contribution of each omic to the data fusion process. Results of these tests showed that *HC-fused* represents a great competitor to other *state-of-the-art* algorithms such as *SNF* by Wang et al [6]*, PINPLUS* by Nguyen et al [7] or *NEMO* by Rappoport and Shamir [8].

2.2 Method example

This subsection will present a short example of how the algorithm works. It will offer the initial configuration of the method and the states of the fusing network after each merging of clusters.

Given a number of only 5 patients and 2 types of omics for them, a network is being formed. This will act as the input for the *HC_fused* top function. Initially, there are 5 clusters, each one having only 1 patient. This initial configuration is presented in *Figure 2.5* in pseudocode.

```
1   omics = (#omics=2, #patients=5, TCGA_data);
2   network = calculate_network (omics);
3
4   Cluster1 = Patient1;
5   Cluster2 = Patient2;
6   Cluster3 = Patient3;
7   Cluster4 = Patient4;
8   Cluster5 = Patient5;
9
10  Clusters = {Cluster1,Cluster2,Cluster3,Cluster4,Cluster5};
```

Figure 2. 5 Algorithm's initial configuration example pseudocode

The top function uses the *network* as input to sequentially calculate distances between patients, identify merging clusters based on computed distances, merge clusters and finally update the fused network based on that merging – the earlier the merging of a patient, the higher its corresponding value. In the end only one cluster will remain as illustrated in *Figure 2.10.* In Stage 1, as seen in Figure 2.6, the initial structure of the clusters is shown on the left side and the fusing network is shown on the right side – initialized with 0. Afterwards, based on the similarities between Patient1 and Patient4, clusters 1 and 4 merge, the cluster 5 becoming the forth remaining cluster in Figure 2.7. This merge is recorded in the fusing network with the color green. At each stage current and previous merges are incremented in the fusing network. For a better visualization, a darker color signifies an older pair of merging patients – thus a larger number. The third stage from Figure 2.8 presents the merging of cluster 3 with the forth cluster containing only Patient5. At this point, only 3 clusters remain. In stage 4, as seen in Figure 2.9, clusters 2 and 3 merge, thus Patient3 and Patient5 will become part of cluster 2 which already included Patient2. Finally the remaining two clusters merge into only 1 cluster which contain all of the patients. Once the Fused Network is updated – which represents the output of this method – the algorithm concludes.

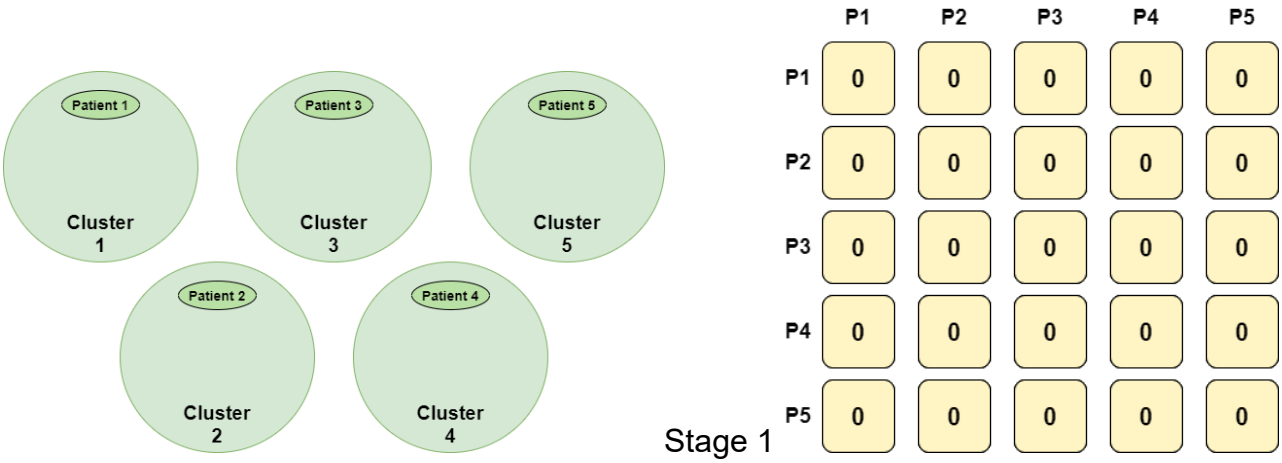

Figure 2. 6 Initial state of clusters and network

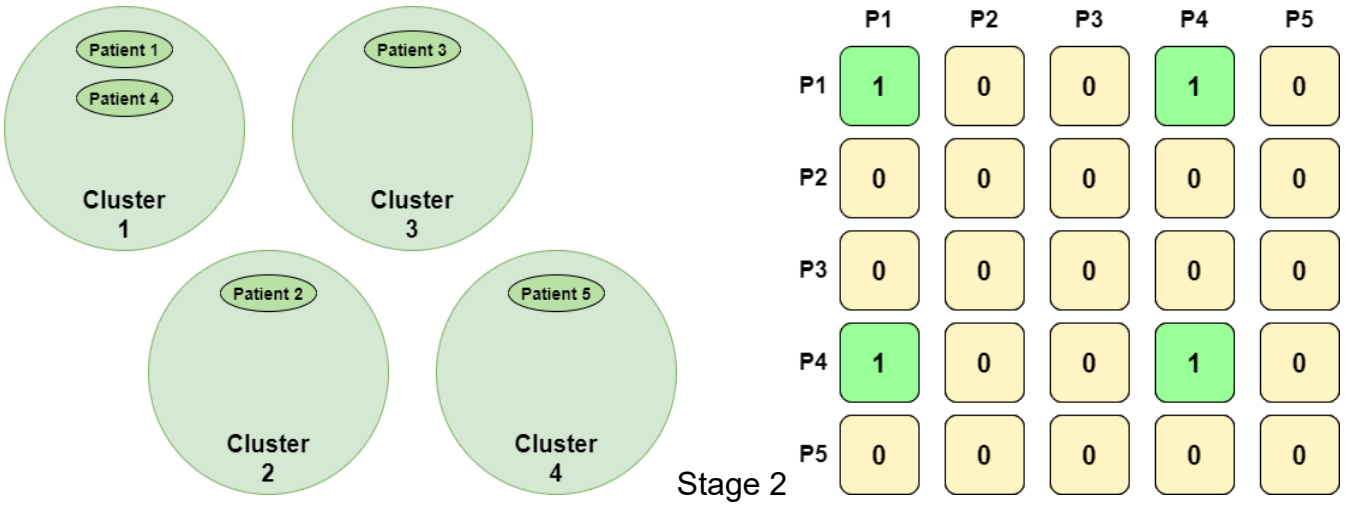

Figure 2. 7 First merge of clusters and network update

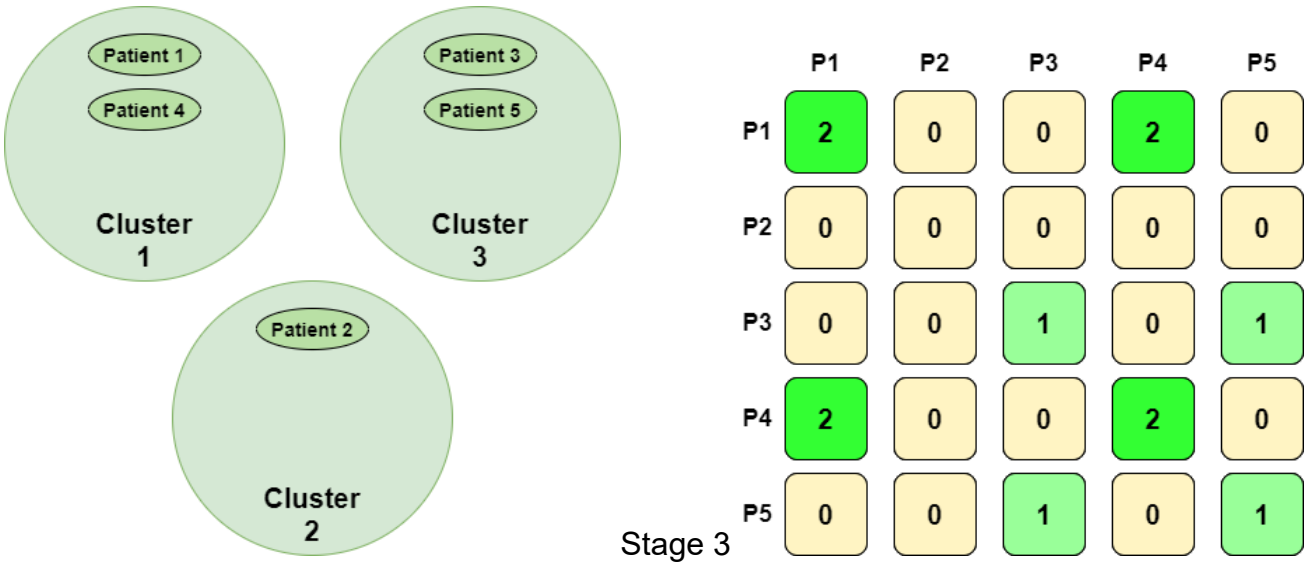

Figure 2. 8 Second merge and continuous update of fusing network

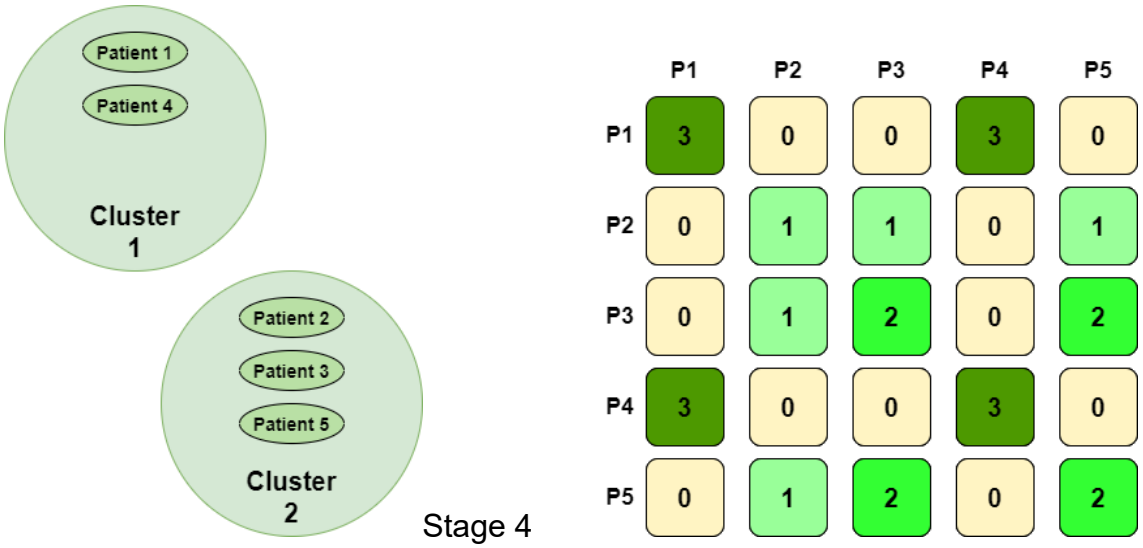

Figure 2. 9 Last merge of clusters

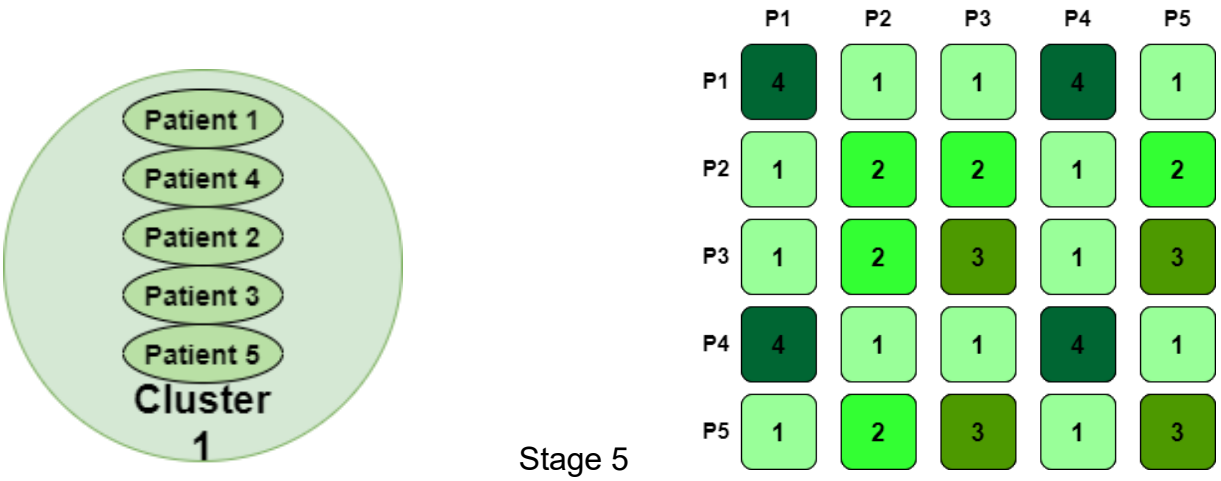

Figure 2. 10 Final state of fused network

The way the clusters merge, as illustrated in *Figure 2.7*, is a result of the computations that take place inside the *HC_fused* function. The algorithm is shortly presented in a pseudocode version in *Figure 2.11*, using the initial configuration from *Figure 2.6*.

```
HC_fused (network, #iterations){
    for (iter in 1..#iterations){
        clusters_dynamic = clusters; // Reset the clusters
        #clusters = #patients; // Reset number of clusters
        while (#clusters) > 1){
            distances = calculate_distances (network, clusters_dynamic); // Calculate similarities
            indexes = find_max_indexes (distances); // Find indexes of elements equal to max value
            pair = find_pair (indexes, network); // Find merging pair
            clusters_dynamic = merge_clusters (pair); // Merge and update clusters based on pair
            #clusters--; // Decrease number of clusters
            fused_network = update_network (clusters_dynamic); // Update the fused network
        }
    }
}
```

Figure 2. 11 Top HC-fused function pseudocode

3. Literature review

In this section several papers related to the topic of this project will be reviewed. This chapter will offer a broad overview of similar hardware implementations and their capabilities in accelerating algorithms, such as ones that perform hierarchical clustering, obtain propensity score, do match grouping minimize differences and others. By analyzing the results, new perspectives upon the expected results for this project are being formed, finally aiding in the hardware optimizations process.

In Wibowo et al. [9] the authors propose a design analysis of the K-Means clustering algorithm further implemented on FPGAs. This method is widely used in data mining – Berkhin [10], learning applications – Ahuja et al [11], object tracking – Keuper et al [12], pattern recognition – Baraldi and Blonda [13] and it is based on assigning objects from the dataset as centroids and then merging each object with the cluster with the closest centroid, either by partitioning or by hierarchical clustering. The paper states that using a symmetric multiprocessing (SMP) and an FPGA is slightly less efficient than using it with a graphical processing unit (GPU), however the FPGA allows data movements and decoupling accelerator computation and communication, which finally offers a great performance. The hardware implementation offered an acceleration of 50 times compared to the software one (where the functions were handled by the microprocessors) when processing different datasets for a server problem. The design used FPGA Xilinx Artix7 with a clock frequency of 50 MHz and was implemented in VHDL using mostly comparators for the K-means algorithm. The total memory usage of an 8-bit design was of 344672 KB, out of which 48 Slice LUTs, 48 LUT Flip Flop pairs and 40 Bonded IOBs. The design was run in about 13 seconds real-time and 12.78 second CPU time.

An efficient way of performing hierarchical agglomerative clustering (HAC) using high-end GPUs is presented in Shalom et al. [14]. The main focus here was to reduce runtime and memory bottlenecks of the algorithm through the use of partially overlapping partitions (PoP) as parallelism method and acceleration of computations. HAC algorithm is generally used in data mining applications, such as research in microarrays, sequenced genomes and bioinformatics. Initially, each object is considered a cluster and will further merge to the closest pair until only one cluster remains. The acceleration is mainly needed in calculating the distances and identifying the closest pair. The paper shows the benefits in terms of time reduction and memory complexity of transitioning the HAC implementation from the CPU to the GPU, mainly due to the GPU parallel features. The PoP idea resides in the fact that the closest pair is found for each individual cell, independent of the others. Three different implementations are presented: the traditional HAC on the GPU, the CUDA (Compute Unified Device Architecture) based PoP HAC on GPU and lastly the parallelization of the PoP computations on the GPU using CUDA. The final measurements for a data set of 15 thousand data points show that PoP GPU version is 6.6 times faster than the traditional one on the GPU and about 443 times faster than the one on CPU – due to the sequential nature of it. Memory-wise, the PoP HAC implementation on the GPU also requires far less memory than the traditional one: about 67 MB for 100,000 data points compared to 28 GB. The steps performed in the parallelization of computations are described as follows:

- Identify the tasks that can be parallelized and break down code into simpler tasks. Perform profiling to discover execution times for these tasks and the bottlenecks;
- Parallelize by breaking down tasks into executable multi-steps such as inner loops and coding the kernels to access multiple memory locations simultaneously;

- Integrate with main control program after parallelizing the tasks and check for runtime improvements. Redo first steps until the target speed gain is obtained and further evaluate and select the implementation;
- Implement and test the result comparing it with the sequential CPU version;

From Legorreta et al. [15] a novel hardware implementation for a hierarchical clustering algorithm is described. This method was mainly being used for text documents – based on text similarity - further adapted to acceleration and lastly implemented on a Field-programmable Port Extender (FPX) platform. This clustering algorithm focuses on enlarging intracluster similarity to the detriment of the intercluster one, based on a similarity metric. As in other hierarchical clustering algorithms, the clusters will iteratively be clustered. Here, the clustering via Hierarchical Partitioning, as implemented by Behrens et al. [16], is described as getting as input a set of N multidimensional points in a binary space and outputting a binary tree in which each leaf corresponds to only one point. The *Liquid Architecture Platform* by Jones et al [17] has been used to extend the FPX and allows an efficient reconfiguration of the architecture. It also provides the necessary interface in which an FPGA hardware is able to communicate with SRAM, SDRAM and high speed network interfaces. This platform alongside the *Leon* processor, containing a co-processor interface and which allows a parallel execution, obtain an increased performance in execution. Three different stages are included in this method. The first one computes the bitwise sum, the second one – the dot product and the last one returns the score for each document. For N k-dimensional vectors, where $k = 4000$, the unaccelerated method will take $12000*N$ operations, while the accelerated one will only take $0.023*4000*N + 0.31*4000 + 4*N = 99*N + 125$ operations, thus a large speedup factor.

Another similar accelerator, this time for biomedical applications is described in Page et al. [18]. It comes as a necessity for health monitoring systems which require accuracy, security and a low processing time while dealing with a large amount of data. The processing time consists mostly of feature extraction, data fusion and classification via the usage of kernels, which are mapped and executed by a manycore accelerator called Power Efficient Nano Clusters (PENC). As most biomedical applications, there is a crucial need to process in parallel the input data and often a large number of digital signal processing blocks and machine learning techniques are being used. In the paper, the capabilities of Atom and ARM are discussed alongside FPGAs, stating the need to use the latter due to their flexibility and parallel nature, the downside being the requirement of low-level logic writing and the higher leakage power. The PENC accelerator consist of processors with 6 stage pipeline, RISC instruction set for the DSPs and a Harvard memory model, having a 16-bit data path. For the Artix-7 FPGA, the implementation was performed in Verilog. The design was performed by taking into consideration area, power and latency and through the dedicated synthesis tool from Xilinx, the RTL was generated. Optimizations were performed with respect to bit resolution, parallelism and pipelining. For this implementation, power and timing results were generated from the Xilinx XPower and Xilinx Timing analyzer. The results showed that the PENC proved to have a faster development time compared to the FPGA by allowing Assembly implementations and later C ones and performing simulations on existing hardware. The PENC accelerator also showed faster results than the FPGA, Atom and NVIDIA TK1 implementations (10x, 15x, 7x faster respectively), for processing one window of Electroencephalogram (EEG) data.

In Sekhon's paper [19] the software optimizations of the *Matching* package from *R* language are described, alongside different computation variants for executing the code in either sequential or parallel way. In this article, the R functions available in the package, namely *Match*, *GenMatch* and *MatchBalance* were translated into a C++ code capable of efficiently dealing with the intense computations that take place in the algorithms. These functions consist of several matching algorithms capable of obtaining propensity score, inverse variance, group of matches able to minimize differences and much more. The results of the subject data are showed prior and post the usage of the algorithms, indicating the positive impact of the matching methods. By also using the *Simple Network of Workstations* R package, the *Genmatch* function can be parallelized on multiple CPUs or clusters of computers. This comes in really helpful due to the large number of computations needed for these datasets. The results show the benefits in terms of run time of a parallel structure, the timing indicating an acceleration of more than 3 times when using 4 CPUs instead of 1 for example. The article also indicates the observation that the execution time of the algorithm did not increase linearly with the sample size, but instead in a polynomial way. Another interesting fact is the observation that the Intel C++ compiler was not able to create a faster code, as did the GNU g++ compiler which successfully carried the task of optimizing the package.

In this project, a similar approach to Legorreta et al. [15] will be taken, the design being for an FPGA instead of for an FPX. Since the implementation will be using High-Level Synthesis, there will be no need of Verilog or VHDL as done by Page et al. [18]. The software optimizations will be performed after the R translation of the package containing the algorithm to C++, as was done by Sekhon [19], however this time by taking advantage of the capabilities of the *Rcpp* library. The hardware acceleration will be made for the computation of the Similarity Matrix and for the identification of the merging pairs of clusters. It is expected that by exploiting and exploring the parallelism available through the use of FPGAs, the algorithm will show great improvements in its execution time. To the best of the author's knowledge, this project presents the first attempt to accelerate the *HC-fused* algorithm for disease subtype discovery using a dedicated computer architecture mapped to FPGA.

4. Software design

This chapter describes entirely the process which aimed to fully and correctly translate the functionality of the R version algorithm to a series of improved and adapted C++ implementations. The methodology of this section is shortly presented below, each step being further described in the following subsections.

Methodology

- Successfully translate it to a C++ version then measure performance and verify functional correctness.
- Interface the R software with the C++ source files.
- Improve the C++ code iteratively by implementing optimizations in successive versions.
- Create an R package with all the C++ versions.
- Apply changes to the compiler settings.

4.1 Conversion to C++

This subsection will cover the first performed conversion of the algorithm from R to C++.

To better understand the algorithm and the further presented code, an overview of the method is presented. The arguments that are fed to the method are a list of binary matrices (stored in ***MAT*** variable) of size equal to the number of patients times the number of patients (***n_patients*** x ***n_patients***) and the number of iterations of the hierarchical clustering. The number of the binary matrices is given by the number of omics and is stored in a variable called ***n_elems***, while the number of patients is stored in the ***n_patients*** variable. Based on this number the variable that holds the clusters of patients is declared. Also the returned value of the top function, the ***NETWORK*** variable is of size ***n_patients*** *x* ***n_patients***. Based on the input argument matrix ***MAT*** and on the object containing the clusters, the similarity matrix, here called ***distances*** is being computed. Afterwards via the successive calls of a series of functions (including *which_is3* and *which_vec_mat functions*), the indices of the 2 clusters that are going to merge per iteration are being saved in the ***map_info_pair*** and further used in the merging and erasing of them from ***obj*** – object that contains the clusters of patients. Lastly, the ***NETWORK*** output is being updated based on this and previous merges.

Starting off with the first translation of the R code to C++, this version has the longest execution time out of all others, including the R one. This is mainly because the code was written in a way to mimic as much as possible the code style specific to R. As a result, a large overhead was obtained leaving a lot of room for further improvement in the next versions. The code consists of a total of 9 functions, including the top one.

The main benefits of this version are:

- The final result returned from the top function was verified and proved to be correct.
- The integration between R and C++ was checked.
- Further implementation no longer needed the R code as a reference.

4.2 Software optimizations

This subsection will present an overview of the successive created C++ version and each stage of optimizations performed. Every optimization stage will first be described in a more scientific manner and afterwards the technical details behind them will be assessed. The resulting execution time will be presented for each of the method's versions including the R one. The chapter will conclude with an overview of the performance for each version. Either pseudocode or C++ code is presented for the optimizations, depending on the functionality of the enhanced code.

At this stage of the project, two datasets were used to check the functionality and execution time of the code. The first and smaller one is made of 105 patients and includes two types of omics: *mRNA* and *Methy*. This dataset was applied as input to all of the presented C++ versions and to the R one. The other dataset containing 849 patients and an additional set of omics compared to the first one (*miRNA)* was only applied to the R version and to the last two best C++ versions, due to timing reasons. In order to have a timing result as close as possible to the correct one, the hardware upon which the algorithm was running was not used for other purposes during the execution.

4.2.1 Improved code organization and decreased number of computations

The next implementation showed a much better result, having an execution time for the same dataset of only 17.44 seconds for 10 iterations. This means an improvement of 16.12 times faster than the previous version and 4.5 times faster than the R version, thus the C++ integration an R already showing its benefits. Some of the major per stage improvement will be further presented using code snippets and described.

Improvements from the first C++ version to the second one are the following:

- The data structure containing the clusters of patients has been declared with the maximum potential size needed to cover all possible scenarios. In this way the memory allocation for it is deterministic. The data structure, called *obj,* contains the clusters that merge over time and has been declared as an *n_patients* x *n_patients* matrix (*n_patients* being the number of patients from the input dataset) instead of an *n_patients* x 1 matrix that grew over time. This declaration also prepares the code for the hardware design, as the maximum size is already allocated and known from the beginning. This implementation can be observed in *Figure 4.1.*

```
1   obj = matrix (#patients x 1);
2   for (i in 1..#patients){
3       obj[i][0] = i;
4   }
```

```
1   obj = matrix (#patients x #patients);
2   for (i in 1..#patients){
3       obj[i][0] = i;
4   }
```

Figure 4. 1 obj declaration with maximum allocation for worst-case scenario

Before (top) and after (bottom)

- The actual number of patients from each cluster is identified using an additional vector variable called *obj_sizes*. This additional data structure comes a support to the one containing all the clusters of patients and it identifies the actual number of patients from each cluster. This is kept for all the following versions of the C++ code, as in *Figure 4.2.*

```
obj_sizes_initial = vector (#patients);
for(i in 1..#patients){
    obj_sizes_initial[i] = 1;
}
```

Figure 4. 2 obj_sizes declaration and initialization

- Based on these previous improvements, the way the clusters data was updating has been adapted.
- The data structure called *obj_sizes* which holds the valid number of patients from each cluster has also been adapted to the current implementation. Both this and previous upgrade can be seen in *Figure 4.3.*

```
obj_dyn[merging_cluster1] += obj_dyn[merging_cluster2]
erase(obj_dyn[merging_cluster2])
```

```
for(i in 0..obj_sizes[merging_cluster2]){
    obj_dyn[merging_cluster1][obj_sizes[merging_cluster1]+i] = obj_dyn[merging_cluster2][i];
}
obj_sizes[merging_cluster1] += obj_sizes[merging_cluster2];

erase(obj_dyn[merging_cluster2]);
erase(obj_sizes[merging_cluster2]);
```

Figure 4. 3 obj and obj_sizes merging and erasing

before (top) and after (bottom)

- In the function that computes the similarity matrix – the distances – an inner function called *mean_matrix_elem_selection* was substituted with a much faster *getmean* function. Both functions are used to compute a mean value of elements from a given selected matrix and can be seen in *Figure 4.4.* Only two for loops are used instead of four, fact that provides an increased efficiency in the total computation time for the similarity matrix.

```
mean_matrix_elem_selection(network, lines, columns){
    for(i in 1..#omics){
        for(j in 1..#patients*#patients){
            for(li in 0..size(lines)){
                if(i!=lines[li]){
                    continue;
                }else{
                    for(co in 0..size(columns)){
                        if(j!=columns[co]){
                            continue;
                        }else{
                            result += network[i][j];
                        }
                    }
                }
            }
        }
    }
    result = result/(size(lines)*size(columns));
    return res;
}
```

```
getmean (i,j, network, obj, obj_sizes){
    for(k in 0..obj_sizes[i]){
        for(l in 0..obj_sizes[j]){
            accum += network[ obj[i][k] ] [ obj[j][l] ];
        }
    }
    accum = accum / (obj_sizes[i]*obj_sizes[j]);
    return accum;
}
```

Figure 4. 4 mean_matrix_elem_selection (top) and improved getmean function with reduced number of loops and conditions (bottom)

4.2.2 Removed redundancy and converting 3D data to 2D

The next two versions of C++ code presented in this subsection were executed in 16.14 seconds and 5.25 seconds, respectively for 10 iterations, the first one being just a slight improvement compared to the previous one – only 1.08 times faster and compared to the R version, 4.8 times faster. The major change that occurred in this version is the swapping of a 3D vector with a 2D one in the process of selecting the merging clusters. The simple change is illustrated in *Figure 4.5*:

```
MAP_matrix_2 = 3D_vector (dim1 x dim2 x dim3)
for(i in 0..dim1){
    for(j in 0..dim2){
        for(k in 0..dim3){
            MAP_matrix_2[i][j][k] = pairs[k][j];
        }
    }
}
```

```
MAP_matrix_2 = matrix (dim1 x 2)
for(j in 0..dim1){
    for(k in 0..2){
        MAP_matrix_2[j][k] = pairs[k][j];
    }
}
```

Figure 4. 5 3D (top) to 2D (bottom) vector declaration and assignment improvement by using less memory and less loops

The next version on the other hand provoked a much significant improvement, running 10 iterations in only 5.25 seconds, thus being 3.07 faster than the aforementioned one and having an overall execution time of 15 times faster than the original one.

The following improvements have been made to this version compared to the previous one:

- Removed *combn2* function and used 2 *for* loops instead for creating a matrix with all possible combinations between numbers based on a certain input object length. The result of the two *for* loops is assigned to the *pairs* variable, as in *Figure 4.6*

```
combn2 (obj_length){
    //Shortcut to combinations of obj_length taken by 2
    column_number = obj_length * (obj_length-1) / 2;
    line0 = 1;
    line1 = 2;
    combinations = matrix (2 x column_number)
    for (j in 0..column_number){
        combinations[0][j] = line0;
        combinations[1][j] = line1;
        if (line0 + 1 == line1){
            line0 = 1;
            line1++;
        }else{
            line0++;
        }
    }
}
```

```
//Shortcut to combinations of obj_length taken by 2
column_number = obj_length*(obj_length-1)/2;
pairs = matrix (2 x column_number)

column_index=0;
for(i in 1..obj_length){
    for(j in i+1..obj_length+1){
        pairs[0][col_index] = i;
        pairs[i][col_index] = j;
        col_index++;
    }
}
```

Figure 4. 6 Pairs improvement instead of combn2 function

before (top) and after (bottom)

- Eliminated the necessity of using *MAP_matrix_2* by directly assigning the results from the two new *for* loops. Now the *map_info_pairs* will be directly assigned values from the *pairs* variable, as shown in *Figure 4.7.*

```
pairs = combn2(obj_length);
MAP_matrix_2 = matrix (dim1 x 2);
for(j in 0..dim1){
    for(k in 0..2){
        MAP_matrix_2[j][k] = pairs[k][j];
    }
}

map_info_pair[0] = MAP_matrix_2[id_min][0];
map_info_pair[1] = MAP_matrix_2[id_min][1];
```

```
map_info_pair[0] = pairs[0][id_min];
map_info_pair[1] = pairs[1][id_min];
```

Figure 4. 7 Updating map_info_pairs improvement

before (top) and after (bottom)

4.2.3 Improved data declarations, network update and functions

Although this version does not have a significant impact on the execution time of the algorithm, it provided a lot of beneficial enhancement and room for further optimizations. The execution time for 10 iterations was registered to be of about 4.65 seconds, 1.13 times faster than the previous version and 17 times faster than the original one.

The major upgrades applied to this version are presented below:

- Improved the way *matAND* variable is created in the top function. This variable stores the bitwise logic *and* operation for the *MAT* input matrix. Both implementations are shown in *Figure 4.8*

```
matAND = matrix (#patients x #patients);
for(i in 0..#patients){
    for(j in 0..#patients){
        matAND[i][j] = 1;
    }
}

for(i in 0..#patients){
    for(j in 0..#patients){
        for(elem in 0..#omics){
            //Perform bitwise logic and
            matAND[i][j] = matAND[i][j] && network[elem][i][j];
        }
    }
}
```

```
matAND = matrix (#patients x #patients);
for(i in 0..#patients){
    for(j in 0..#patients){
        matAND[i][j] = network[0];
    }
}

for(i in 0..#patients){
    for(j in 0..#patients){
        for(elem in 1..#omics){
            //Perform bitwise logic and
            matAND[i][j] = matAND[i][j] && network[elem][i][j];
        }
    }
}
```

Figure 4. 8 matAND declaration improvement

before (top) and after (bottom)

- The update of the output of the algorithm, the *NETWORK* matrix has been improved the way *NETWORK* is getting updated at the end of the method. Functionality has been kept while using less code, as seen in *Figure 4.9.* Only 3 *for* loops are now being used instead of 5.

```
for(obj_index in 0..#patients){
    object = obj_dyn[obj_index];// Assign a cluster
    if (obj_sizes[obj_index]) > 1){ // More than one patient
        for(i in 0..#patients){
            for(j in 0..#patients){
                for(li in 0..obj_sizes[obj_index]){
                    if(i!=object[li]){
                        continue;
                    }else{
                        for(co in 0..obj_sizes[obj_index]){
                            if(j!=object[co]){
                                continue;
                            }else{
                                NETWORK[i][j]++;
                            }
                        }
                    }
                }
            }
        }
    }
}
```

```
for(obj_index in 0..#patients){
    object = obj_dyn[obj_index];// Assign a cluster
    if (obj_sizes[obj_index]) > 1){ // More than one patient
        for(i in 0..#patients){
            for(j in 0..#patients){
                NETWORK[ object[i] ][ object[j] ]++;
            }
        }
    }
}
```

Figure 4. 9 Network update improvement with less loops and conditions before (top) and after (bottom)

- Created a function *get_ij* that returns the *map_info_pair* required in selecting the clusters that will merge. *Figure 4.10* illustrates the functionality of the *get_ij* function, written in C++.

```
get_ij (index, obj_size){
    col_elems = obj_size;
    nr_elems = obj_size;
    sum_col_elems = 0;
    res = vector {-1, -1};

    if(index < obj_size){ // first column
        res[0] = 0;
        res[1] = index;
        return res;
    }

    for (int i=1; i<obj_size;i++){
        col_elems--;
        nr_eles += col_elems;
        sum_col_elems += col_elems;
        if (index < nr_elems){
            res[0] = i;
            res[1] = index - sum_col_elems;
            break;
        }
    }
    return res;
}
```

```
map_info_pair = get_ij(id_min/3, obj_dyn.size());
```

Figure 4. 10 get_ij function implementation and call

4.2.4 Function merge and converting 2D data to 1D

This stage also greatly contributed to the efficiency of the method, making the algorithm execute 10 iterations of the small dataset in only 1.3 seconds. In other words, compared to the initial R one this is already 60 times faster, while also being about 3.57 faster than the previous optimized version.

The main improvements performed at this stage are the following:

- Combined two of the functions (*which_vec* and *which_mat* – shown in *Figure 4.11*) that aided in computing the merging clusters into one single function called *which_vec_mat,* which can be seen in *Figure 4.12*.

```
which_vec (distances){
    max_element = max(distances);

    for(element in distances){
        if (element == max_element){
            mylist.push(index);
        }
    }
    count = size(mylist);
    result = vector (count);

    for(element in mylist){
        result = mylist.pop_front();
    }
```

```
which_mat (distances){
    max_element = max(distances);

    for(element in distances){
        if (element == max_element){
            mylist_lines.push(line);
            mylist_column.push(column)
        }
    }
    count = size(mylist_lines);
    result = matrix (count x 2);

    for(element in mylist){
        result[][0] = mylist_lines.pop_front();
        result[][1] = mylist_column.pop_front();
    }
```

Figure 4. 11 which_vec and which_mat functions, used for extracting the indexes of merging clusters

```
which_vec_mat (distances){
    max_element = max(distances);

    for(element in distances){
        if (element == max_element){
            mylist.push(index)
            mylist_lines.push(line);
            mylist_column.push(column)
        }
    }
    count = size(mylist);
    result_vector = vector (count);
    result_matrix = matrix (count x 2);

    for(element in mylist){
        result_vector[] = mylist.pop_front();
        result_matrix[][0] = mylist_lines.pop_front();
        result_matrix[][1] = mylist_column.pop_front();
    }

    result = {result_vector, result_matrix};
    return result;
}
```

Figure 4. 12 which_vec_mat function which extracts both the 1D and 2D indexes of merging clusters

- Created a new *struct* containing a one-dimensional and a two-dimensional vectors, needed for the return phase of the new function. Having this additional data structure was helpful in being able to call only one function while also sharing some computations an eventually decreasing the execution time. Also the *struct*, called *ids_vec_mat* was wrapped in order for R to know how to handle it. The code is indicated in *Figure 4.13.*

```
#include <RcppCommon.h>

typedef struct{
  vector<int> ids;
  vector<vector<int>> ids2;
} ids_vec_mat;

namespace Rcpp {
 template <>
 SEXP wrap(const ids_vec_mat& x);
}

#include <Rcpp.h>
using namespace Rcpp;

namespace Rcpp {
  template <>
  SEXP wrap(const ids_vec_mat& x) {
    Rcpp::NumericVector ids;
    Rcpp::NumericVector ids2;
    return Rcpp::wrap(Rcpp::List::create(Rcpp::Named("ids") = Rcpp::wrap(x.ids),
                                         Rcpp::Named("ids2") = Rcpp::wrap(x.ids2)));
  };
}
```

Figure 4. 13 ids_vec_mat struct declaration and wrapping

- Data types were converted from *double* to *int* were possible, offering an overall reduced memory allocation size.
- Removed the *to_matrix* function (shown in *Figure 4.14)* and maintained all elements of the input argument matrix *MAT* as vectors instead of matrices. The code was adapted for this change, meaning that the functions now worked with 1D vectors instead of 2D ones.

```
to_matrix (myvector){
    mymatrix (sqrt(myvector) x sqrt(myvector));
    for(elem in myvector){
        mymatrix[][] = myvector[];
    }
    return mymatrix;
}
```

Figure 4. 14 Removed to_matrix function

- Removed the *break* from the previous-stage created function *get_ij*, by creating a *bool* variable indicating that the wanted result has been found. A disadvantage is that the loop iterates through all its elements now permanently. The advantage is that the number of iterations for the *for* loop is always known. The C++ implementation of the upgraded function is shown in *Figure 4.15.*

```
get_ij (index, obj_size){
    col_elems = obj_size;
    nr_elems = obj_size;
    sum_col_elems = 0;
    bool found = false;
    res = vector {-1, -1};

    if(index < obj_size){ // first column
        res[0] = 0;
        res[1] = index;
        return res;
    }

    for (int i=1; i<obj_size;i++){
        col_elems--;
        nr_eles += col_elems;
        sum_col_elems += col_elems;
        if (index < nr_elems && found == false){
            res[0] = i;
            res[1] = index - sum_col_elems;
            found = true;
        }
    }
    return res;
```

Figure 4. 15 Improved get_ij function for selecting the two merging clusters

4.2.5 Removing dispensable code

The results of this stage were satisfying enough to declare that the software optimizations can be concluded. However another stage was further required to adapt the code to the HLS requirements, details being described in the section 4.2.6.

This version of code was able to run 10 iterations of the small dataset of 105 patients and two omics in only 0.61 seconds on average. Compared to the previous version this one became only 2.13 times faster. On the other hand, compared to the initial R version this is 128 times faster. Due to this increased performance, another dataset was tested using this version. The new set consisted of data from 849 patients and included 3 types of omics: mRNA, Methy and miRNA. The initial R version was firstly tested with this dataset, being able to run only 1 iteration in about 11 hours. In contrast to this, this stage of optimizations made the current version run 10 iterations in only about 11 minutes, thus accelerating the algorithm by a factor of 604 times. This result is very promising, showing that the code is able to scale greatly with the number of patients and omics.

The notable improvements that were made in this stage are further presented:

- All lists were swapped with vectors, performing the same functionality. Functions have been adapted to this change.
- *which_vec_mat* function has been improved, using a total of 2 *for* loops instead of 5. It identifies the 1D and 2D indexes from the similarity matrix that are equal to the maximum element from it. The pseudocode implementation is seen in *Figure 4.16.*

```
1   which_vec_mat (distances){
2       max_element = distances[0];
3       for(element in distances){
4           if(element > max_element){
5               reset(result_vector, result_matrix);
6               result_vector.push(index);
7               result_matrix.push(line,column);
8           }else if(element == max_element){
9               result_vector.push(index);
10              result_matrix.push(line,column);
11          }
12      }
13  }
```

Figure 4. 16 Improved which_vec_mat function for extracting indexes of merging clusters

4.2.6 Memory preallocation and pointer-based access

This stage of optimizations provided great enhancement to the software part of the project, offering a satisfying result in which the execution time of the code improved even more. Compared to the initial R code which ran 10 iterations in about 78 seconds, for the smaller dataset, this version is able to run in only 0.1 seconds, being 784 times faster. Relatively to the previous stage this is 6.1 seconds faster. As was done with the previous version, the large dataset was also applied to this one. It ran 10 iterations in only 1.95 minutes, being 3384 times faster than the initial one.

Below, the software optimizations that were applied at this stage are indicated:

- All vectors were removed from the source code. Instead pointers to allocated memory were created and passed as arguments. Memory space has been allocated using *malloc*, as shown in the example from *Figure 4.17.*

```
int col_nr = n_patients*(n_patients-1)/2;
double *distances = (double*)malloc(sizeof(double)*(n_elems+1)*col_nr);
for(int i=0;i<(n_elems+1)*col_nr;i++){
  distances[i] = 0;
}
```

Figure 4. 17 Pointers declaration using malloc example

- Removed the ids_struct due to the usage of pointers. The corresponding function which returned the created data structure now returns *void*, the relevant data being assigned value with reference.
- Adapted the entire functionality to working with pointers. The *which_is3* function which return the lines indexes of elements equal to *dist_size* is implemented as illustrated in *Figure 4.18.*

```
1  which_is3 (ids_2D, distance_size){
2      result = vector;
3      for (i in 0..size(ids_2D)){
4          if( line(ids_2D[i]) == distance_size){
5              result.push(i);
6          }
7      }
8      return result;
9  }
```

```
1  which_is3 (ids_2D, ids_valid_size, distance_size, is3_result, is3_valid_size){
2      is3_counter = 0;
3      for (i in 0..2*ids_valid_size){
4          if( line(ids_2D[i]) == distance_size){
5              is3_result[is3_counter] = i/2;
6              is3_counter++;
7          }
8      }
9      is3_valid_size = is3_counter;
10 }
```

Figure 4. 18 Adapted functionality of which_is3 function for selecting merging clusters indexes before (top) and after (bottom)

- Swapped *while* loops with *for* loops. The number of iterations is now known every time.

This version, identified in this report as *Cpp v8,* showed the best results in terms of timing and complexity and was further used in the project. Some main performed optimizations include:

- The removal of redundant code.
- the decreased number of functions and data transfers
- the conversion of multidimensional to one dimensional data
- the preallocation of memory space and usage of pointers

These optimizations among others have all been gathered in this final software version of the hierarchical clustering algorithm written in C++.

In Chapter 5 the process through which this C++ code was used to design an accelerator hardware architecture is going to be presented. Furthermore, this version of the algorithm was additionally used in tests which used diverse datasets consisting of different numbers of patients and omics. These tests naturally took advantage of the increased efficiency in terms of execution time of the fastest implemented C++ method.

5. Accelerator design

This chapter focuses on the architecture of the hardware implemented using High-Level Synthesis and mapped onto an FPGA. It will firstly offer an introduction to High-Level Synthesis, then the reasoning for accelerating only certain parts of the algorithm and finally it will present the developed architectures and their parameters.

5.1 Introduction to High-Level Synthesis

Logic synthesis represents an important process from the electronic circuit design cycle. Through a synthesis tool, an RTL design is being translated to an implementation consisting of logic gates. Generally these RTLs are being described using a Hardware Description Language such as Verilog or VHDL, which describe at an abstract level a targeted behavior.

High-Level Synthesis allows the synthesis of a design at a more abstract level. Coussy et al [20] indicate that even from the beginning, HLS tools were able to deal with relevant design options and parameters such as timing estimations, interfaces and partitioning, communication, synthesis and lastly co-simulation. The functional specification which is written in a High-Level Language involves consuming the input data all at once, performing calculations and further outputting the data simultaneously. This data is usually stored in structures of either floating point or integer types, the user not having to specify the bit-accurate sizes. Additionally, the tools are able to transform the untimed design into a timed one, taking into account the communication interfaces to generate an efficient hardware architecture.

From Coussy et al [20] the following specific HLS tasks have been extracted:

Figure 5. 1 High-Level Synthesis capabilities

In figure 5.1 The HLS capabilities are being enumerated and will be further detailed here. The *specification compilation* is a step in which the high-level language is being translated into a formal representation. At this point usually optimizations are being performed upon the code. Next is the allocating of hardware resources, step in which the necessary hardware such as functional units or storage elements are being reserved for the current architecture. The number depends on the constraints that have been imposed by the user. These elements are being selected from an RTL library capable also of providing information about area, timing and power which is further used in synthesis results. The scheduling of operations represents a process in which each computation is being scheduled accordingly to clock cycles. This involves reading the sources and bringing it to a functional unit, perform the computations and finally send the result to the destination. Depending on the operation involved, the operation may be scheduled in one or more clock cycles. Furthermore, the binding of operations to functional units is done in order to efficiently exploit the functional units' potential of executing an operation. Efficiency is gained due to the selection chosen by the binding algorithm in the case that several functional units are capable of executing a certain task. The next step exposed is the binding of variables to storage elements, representing the assigning of each variable to a viable container. In addition to that, through HLS, transfers are being bound to buses with an increased efficiency in communicating, due to the necessity of assigning each transfer between component to a bus or a multiplexer. Finally, the generation of RTL design represents the conclusive design produced through the HLS based on decisions made in the previous tasks.

5.2 Profiling

In order to better identify the algorithm's tasks which needed the most the hardware accelerating, a profiling process has been made to have a better understanding upon the execution time of these parts. This allows for a greater focus on designing hardware for the most time consuming inner functions or code blocks.

The profiling was performed upon the software version of the code and using the *Chrono* library, more specifically the *steady clock* from this library. The time was measured for all the functions starting from the call of that function and ending with the return from it, after each iteration the resulting time being accumulated. Once the algorithm finished its work, the conclusive times were printed in order to see their values.

The top function calls a total of 5 functions per iteration then updates the patients' clusters and the final output network. The profiling results showed that two of these functions occupy the majority of time, namely *HC_fused_calc_distances()* and *which_vec_mat()*. The first one is used to compute the Similarity Matrix based on the input Structured Network and the forming clusters. The latter one is used in identifying clusters that can merge from the Similarity Matrix – however only two of these clusters will merge per iteration.

Function name	105 patients and two omics Execution time percentage	849 patients and three omics Execution time percentage
HC_fused_calc_distances()	55%	63%
which_vec_mat()	35%	31%

Table 1 Main profiling results

Further measurements have been made on the *HC_fused_calc_distances()* function. Based on the number of omics and the number of patients, the inner function *getmean()* is called several times

and it was found out that it occupies about half of its the execution time, thus around 28-31% of the total execution time. As a result, the *getmean()* function as a standalone also represents a high priority in the hardware acceleration process.

Table 1 shows that the two functions occupy 90% of the execution time for the smaller dataset and 94% for the larger one. These results clearly indicate that the two functions are of most importance in the accelerating process of the algorithm as improving their execution time will have the most effect on the overall method execution time.

5.3 Architecture

Based on the profiling results, the first two priorities for creating RTL using the HLS tool were the functions which computed the Similarity Matrix and the one that helps in the initial discovery of the clusters that are going to merge in the current iteration. In subsections 5.3.1 and 5.3.2 the architectures of the two solutions that were found to accelerate the algorithm are presented.

For each of the architectures design space exploration will be performed in order to find the optimal solutions. The constraints that will decide this optimal solutions will be the hardware resources available on the target device, the memory bandwidth which is considered to be of 10-12 GB/s, an initiation interval of 1 – meaning that data can be feed every clock cycle to the designed hardware and finally, of course, the overall execution time of the designed hardware. For the purpose of this project, the power consumption of the designs was not assessed.

5.3.1 Accelerator architecture for the computations of the Similarity Matrix

The calculation of the Similarity Matrix represents the top priority in developing an efficient architecture for the hierarchical clustering algorithm, based on the profiling results from section 5.2. Results have showed that the execution time of this function which computes the Similarity Matrix also increases with the number of patients and omics included in the datasets. As a result, there are several reasons why this method needs hardware acceleration.

In order to obtain the first synthesis results and an initial architecture for this method, the numbers of loop iterations had to be fixed to a constant value representing the maximum potential value. This change affected almost all of the loops inside the method. The consequence was the extremely increased execution time of the whole method and thus the whole algorithm. In order to avoid some of the introduced overhead created by this constraint, hardware was generated for the *getmean* function, method which was called repeatedly from the function Similarity Matrix calculation function. It was measured that this function was taking about 50% of outer one, so around 30% of the total execution time.

The software version was measured to execute this function in about 146 ns on average (an average was needed since the number of iterations for the loops and also the sizes of the variables were not constant). Based on this, the hardware design should provide a faster result to be viable for usage. However, even after generating hardware for the inner-most function, the initial execution time was of about 79 ms, thus being more than 500 000 times slower than the software version, making it really inefficient for future design improvements. Furthermore, measurements have been made and it was found out that for about 96% of the time, the outer loop only iterates once, compared to the hardware version which iterated every time for a number equal to the total number of patients. In other words, many additionally computations were performed and their result were not useful for the functionality of the algorithm. A decision was made on this matter, and a new inner function was created

named *getmean_inner* which is called from the *getmean* one. Since the latter is no longer considered part of the hardware, its number of iterations was reset to a variable one, thus decreasing the overall execution time.

The architecture which was created for this subsection is detailed in the block diagram illustrated in Figure 5. 2. In order to improve the execution time of the method, the architecture will allow the execution in parallel of the necessary operations. For this reason, a parallelism parameter *P* is being introduced, indicating the level of parallelism of the architecture. For each level of parallelism additional hardware is being allocated for computing a part of the method at the same time with other hardware resources. For each iteration each block of resources will be getting 2 patients at a time, it will process the data and then select a memory location from the input network to add up to the local accumulator. After the loop is finished all of the parallel accumulators are being synchronized and summed in a final accumulator which is then stored back to the memory.

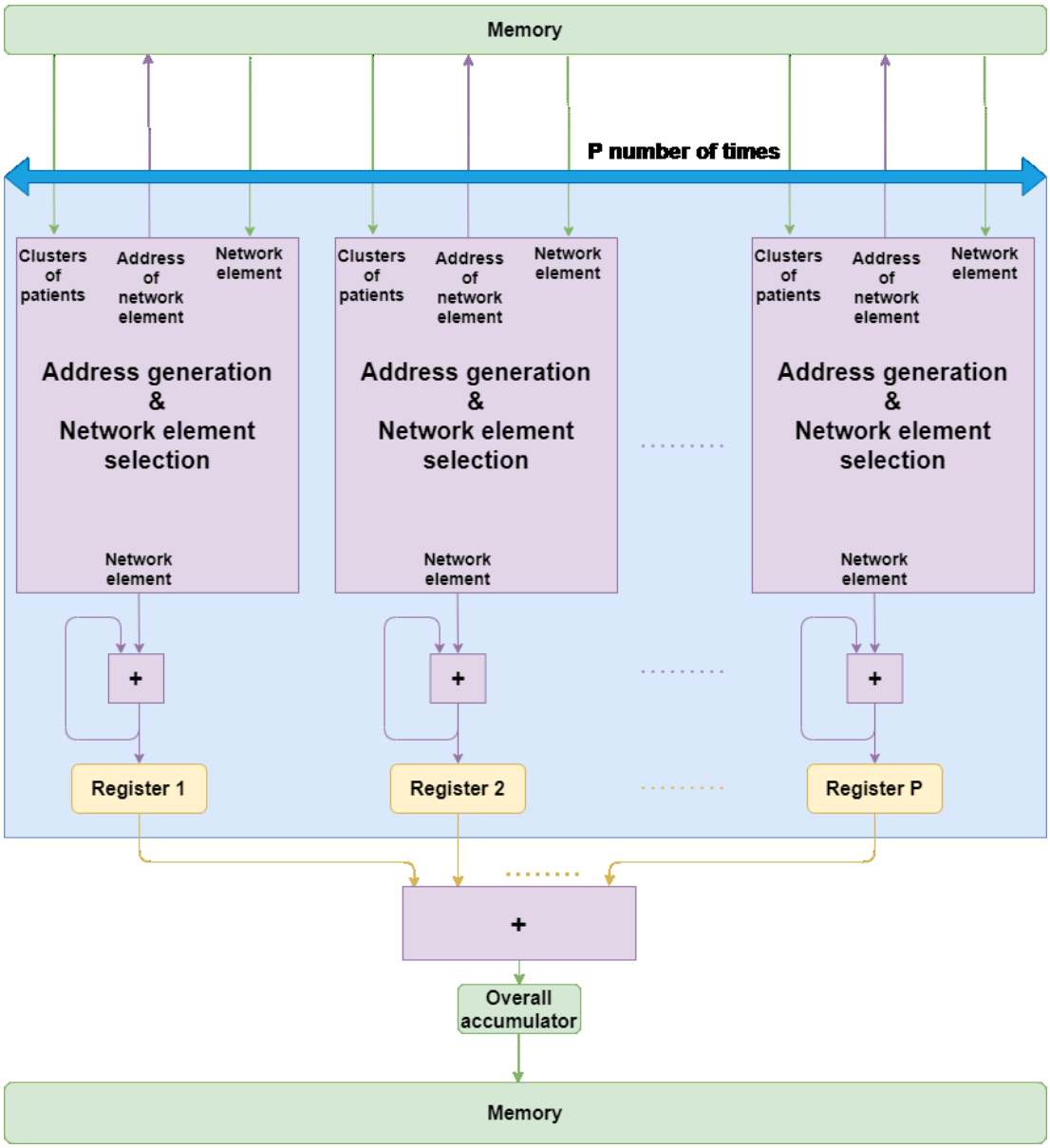

Figure 5. 2 Architecture for computing the Similarity Matrix

5.3.2 Accelerator architecture for computing the merging clusters indexes

Based on the results from the profiling section, this function occupies about a third of the total execution time of the algorithm. Additional measurements were made on the software of this part and it was found out that on average this function runs once in about 60 microseconds. An average upon these runs was necessary because of the variable number of iterations that are present inside of one of the two *for* loops. This variable number of iteration is based on the number of available clusters from the current iteration from which this function is called. The first change that affected the architecture of this hardware was to swap this variable number with a constant one, more exactly the maximum potential number that may arise during these computations. After applying this change and also some directives for indicating the *trip count*, the first synthesis results became available. The timing related parameters are presented in Table 2, while the resources usage is presented in Table 3.

Estimated period	Initiation Interval	Total interval	Latency [cycles]	Pipelined	Bandwidth [GB/s]
6.721 ns	1	32766	32765	Yes	1.48

Table 2 Initial timing results for the merging clusters index finding function

In this initial version of the code, input can be fed each clock cycle. The total number of cycles and the latency are mainly given by the number of iterations for the two loops, indicated via the *trip count* directive, these being equal to 5460 and 3. Additionally, both of the loops have been pipelined successfully during synthesis. The bandwidth has been calculated based on the memory accesses that are needed during runtime and the latency of the execution. During each iteration there are 4 memory accesses for reading and 3 memory accesses for writing, these 3 memory spaces being 3 out of the 4 memory locations from which data is being read. In total per iteration 160 bits are being transferred through the interface per one iteration.

BRAM	DSP	FF	LUT	URAM
0	2	217	836	0

Table 3 Initial resources results for the merging clusters index finding function

The block diagram of the initial fully sequential architecture is showed in Figure 5.3. In this diagram the method part which computes the maximum number is fed only the Similarity Matrix from the memory. Based on it, it further sends out the maximum element found in the matrix towards the second part of the function which extracts the one-dimensional and two-dimensional indexes. This part is also reliant on the Similarity Matrix, but also on the computed indexes from the previous iterations, which also are being obtained from the memory. The result of this part consists of the update of the elements' indexes, the function ending with them being stored back into the memory.

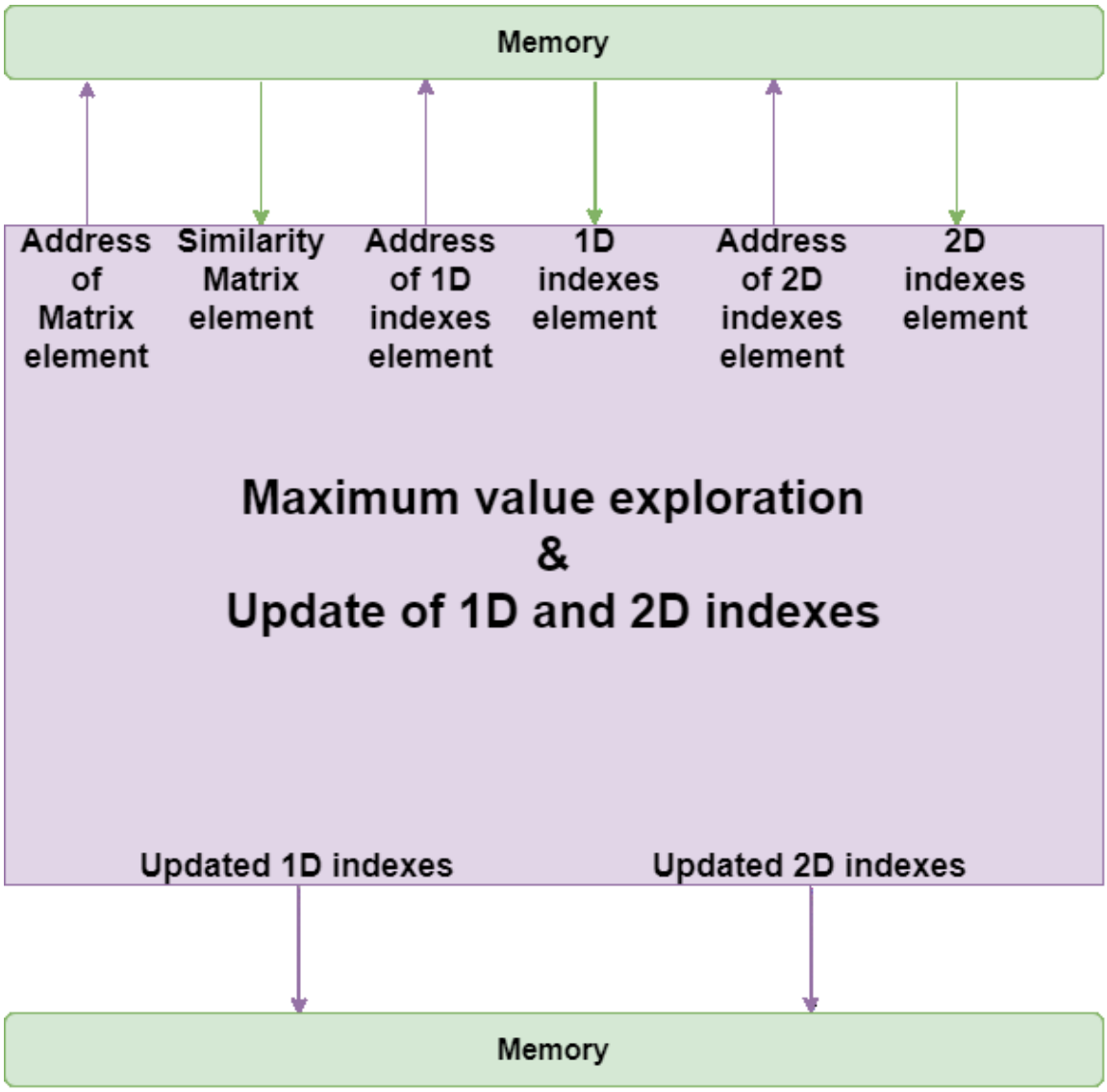

Figure 5. 3 Block diagram of initial architecture of the indexes extraction method

In order to speed up the execution using the available hardware, parallelism should be applied to the function so that data can be processed at the same time by different hardware elements. Since the computation of the maximum value from the Similarity Matrix and the extraction of indexes could not execute in parallel, the function was further divided into two parts: one that computes the maximum value from the Similarity Matrix and one that uses this value to extract the indexes of elements which match it. The division is naturally needed to get the correct indexes based on the correct maximum found in the Similarity Matrix. As a result two design space explorations were made in this section, one for each part necessary in finding the indexes of the merging clusters. The architecture of the first part will be presented in section 5.3.2.1, while the other one will be described in section 5.3.2.2.

5.3.2.1 Accelerator architecture for extracting the maximum value from the Similarity Matrix

For the search of the maximum value for the Similarity Matrix, a separate function has been created in order to exploit as much as possible the potential parallelism and available hardware, while maintaining a memory bandwidth of around 10-12 GB/s for the required memory transfers. This function involves going through the Similarity Matrix and simply comparing elements to find the maximum value.

The block diagram for this architecture is further presented in Figure 5. 4. The architecture of this method was made capable of executing in parallel by dividing into equal chunks the Similarity Matrix and finding *local* maximum values which are ultimately compared with each other to find the overall maximum value of the matrix. Each part of the matrix is read and compared, after which the method outputs 3 maximum values, 1 for each line of the Similarity Matrix. This is done for *P* number of times, *P* representing the parallelism parameter of the architecture.

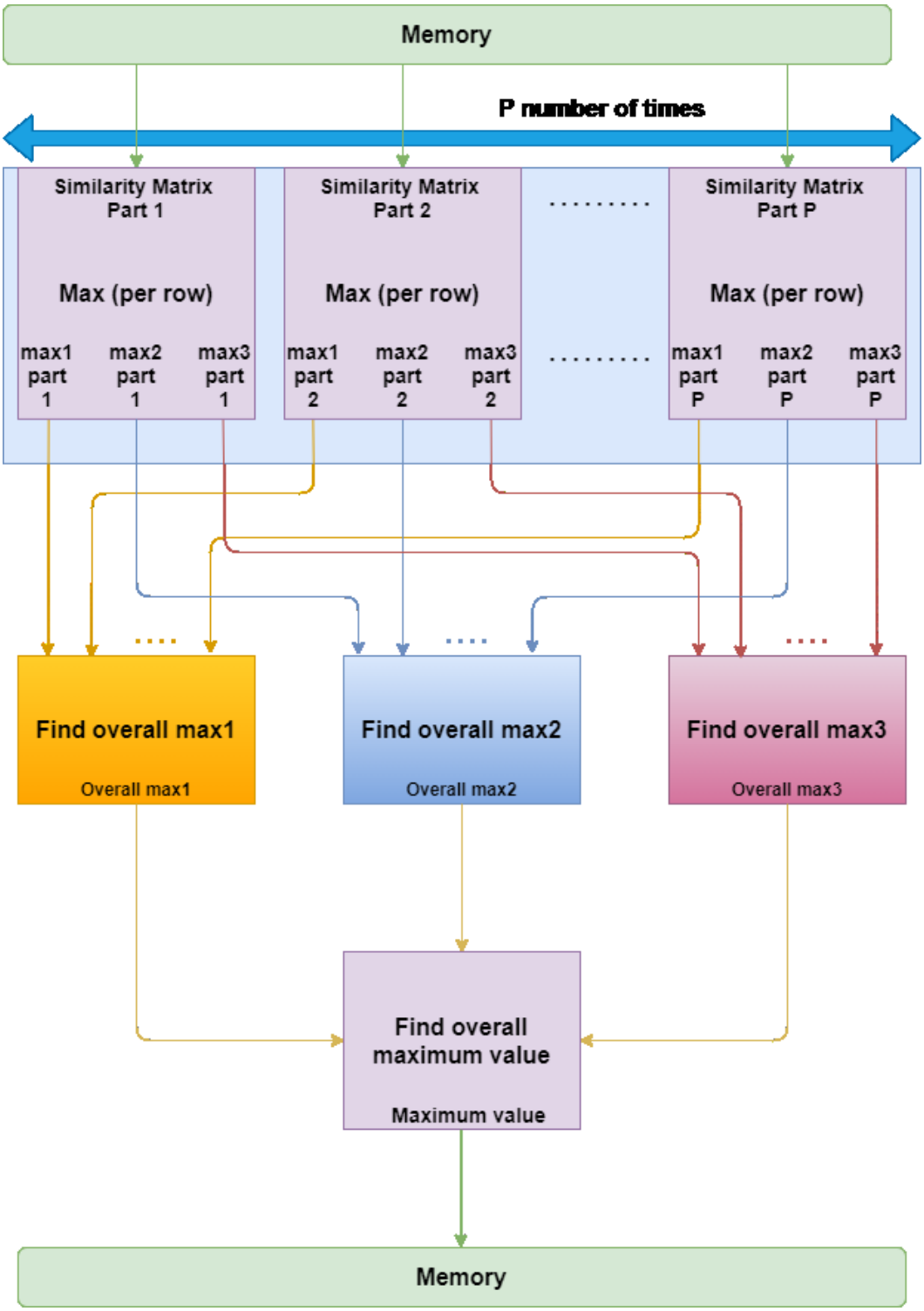

Figure 5. 4 Parallel architecture for maximum value search of the Similarity Matrix

5.3.2.2 Accelerating the extraction of indexes from the Similarity Matrix

This subsection describes the architecture of the designed hardware capable of extracting the 1D and 2D indexes of elements from the Similarity Matrix that are equal to the previously computed maximum value from the design showed in subsection 5.3.2.1. This method basically iterates through the Similarity Matrix and compares each element with the previously found maximum value. In case the values match, then it stores the one dimensional index into a variable and the two dimensional index into another one. These indexes are further used in selecting the clusters that are going to merge in the current iteration.

The block diagram of this architecture is presented in Figure 5. 5. The parallelism parameter *P* indicates the amount of copies needed for memory locations of necessary input data and for actual blocks that perform the comparison and computations needed for updating the indexes. After each block the same memory locations which held the indexes and were read are now being assigned the updated values. As a result of this parallelism, separate chunks of the Similarity Matrix are being checked at the same time by the computational blocks, thing which decreases the overall execution time at the expense of more hardware resources. The functionality of this architecture depends on the prior computation of the overall maximum from the Similarity Matrix.

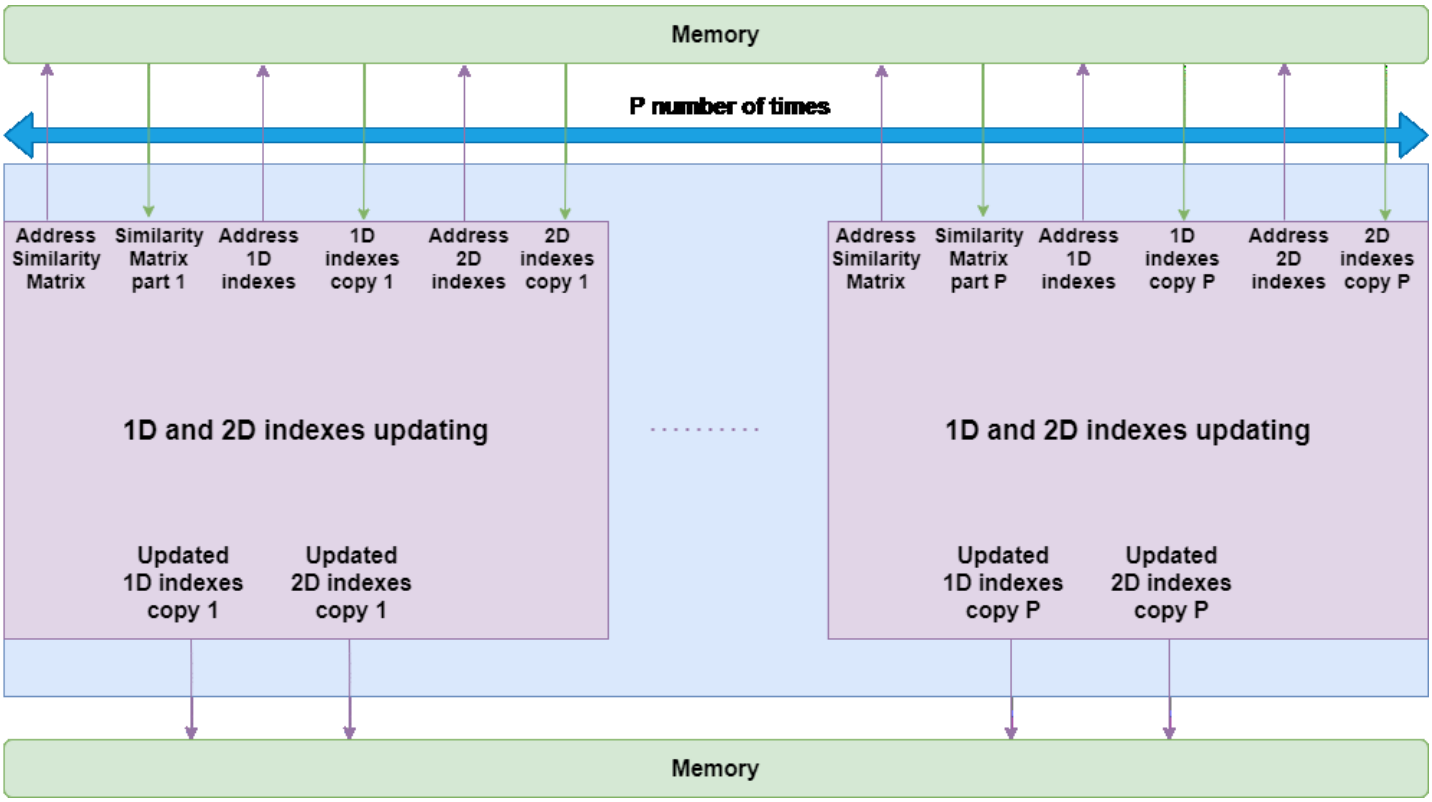

Figure 5. 5 Architecture of merging clusters indexes finder method

6. Implementation

This chapter focuses on the implementation of both software algorithm and hardware design. It will begin with the performed implementation of the software part, indicating the used software programs and libraries, including their interface with the user and their functionality. Afterwards, the hardware implementation will be discussed, starting with the HLS adaptations of the code and the Vitis HLS software which was used for developing the hardware design. Finally, the required directives used to implement the chosen hardware architecture will be assessed.

6.1 Software implementation

6.1.1 R and Rstudio

The algorithm described by Pfeifer and Schimek [1] was written in *R* programming language and can be found on GitHub (pievos101/HC-fused) Pfeifer. According to Venables et al.[21] R represents an integrated environment which offers:

- great options for handling data;
- several calculation possibilities for vector and matrix operators;
- statistic modeling and analysis;
- many graphical display options;
- loads of packages and libraries with easy integration;
- well performed data wrangling;

R can be seen as an extension of the *S* language, created by Rick Becker, John Champers and Allan Wilks at *Bell Laboratories*. In general documentation of the *S* language also applies to *R*. Besides being already an extension of *S*, *R* is rapidly improving itself through its large community which have extended it through the creation of *packages*. Any user can upload a new package to CRAN if the submission passes the *CRAN Repository Policy* requirements. The majority of files required to use R can be found on the official website (at https://CRAN.R-project.org). At this website precompiled binaries can be downloaded to install R on either Windows, macOS, Linux, Debian, Fedora/Redhat or Ubuntu. Also, older versions of R alongside R packages can be downloaded from CRAN.

In this project, R was successfully downloaded, installed and used on Windows and Linux. There were not any notable differences in the execution time for R on the two different operating systems.

Although *R* can be used directly in a command prompt, *Rstudio* was also installed in order to run *R* in a friendlier environment. *Rstudio* is open-source and it is made available at http://www.rstudio.com/. It has been chosen and used as main R working environment during this project. The interface consists of:

- console/terminal/jobs – used as input commands and output results and shown in *Figure 6.1.*

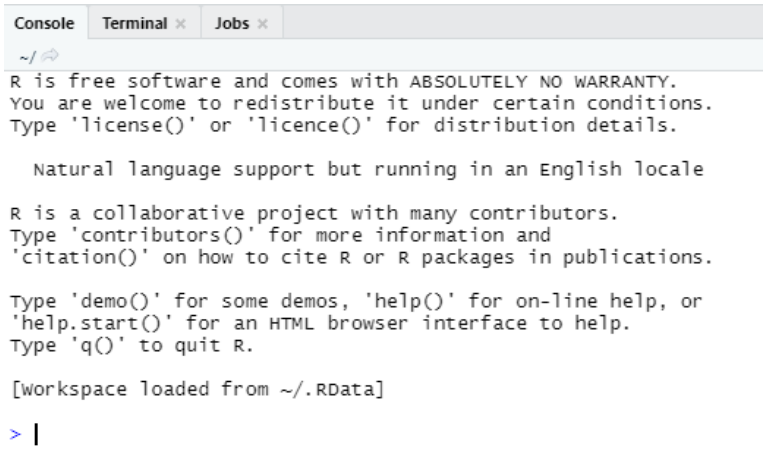

Figure 6. 1 Example of Rstudio console perspective

- environment – overview of data, values and functions;
- history – presents the history of run commands;
- connections – allows the connection to separate data;
- tutorial – offers learning opportunities using the *learnr* package prior installed;

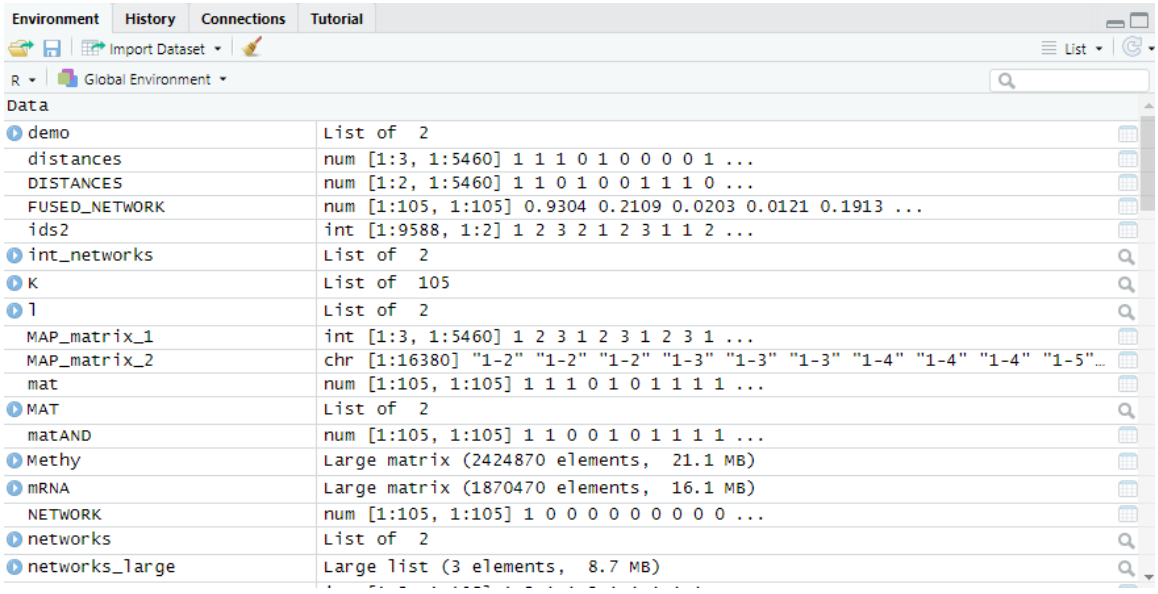

Figure 6. 2 Aspect of the Rstudio Environment

- Source overview – displays the Rscripts, source files and data and allows the user to run and source code. An example is shown in *Figure 6.3*;

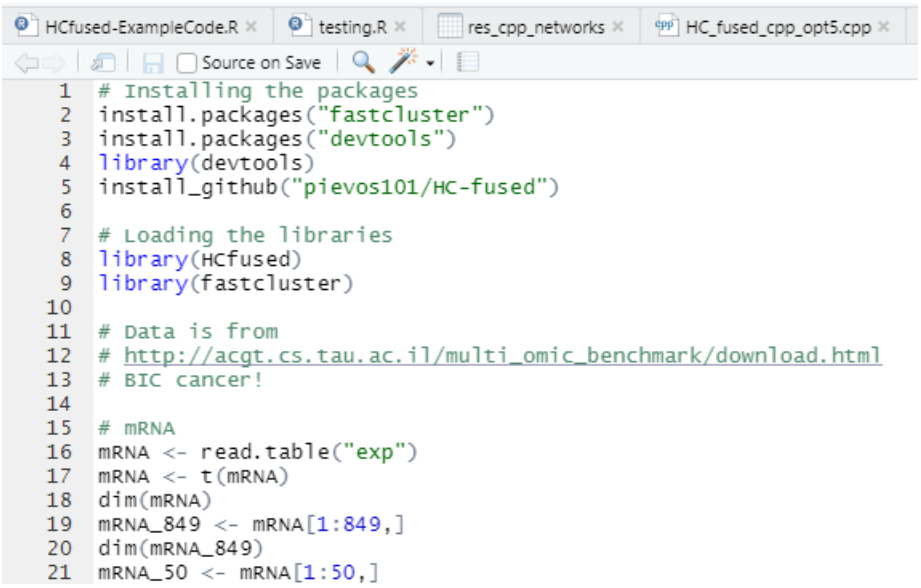

Figure 6. 3 Example of an R script in Rstudio

(Figure from Pfeifer [1]*)*

- Files – displays the folders from the working directory;
- Plots – displays the available plots;
- Packages – indicates installed or ready-to-be-installed packages. Examples of such packages are shown in *Figure 6.4*;
- Help – used to understand R related topics;
- Viewer – used to access content from local web;

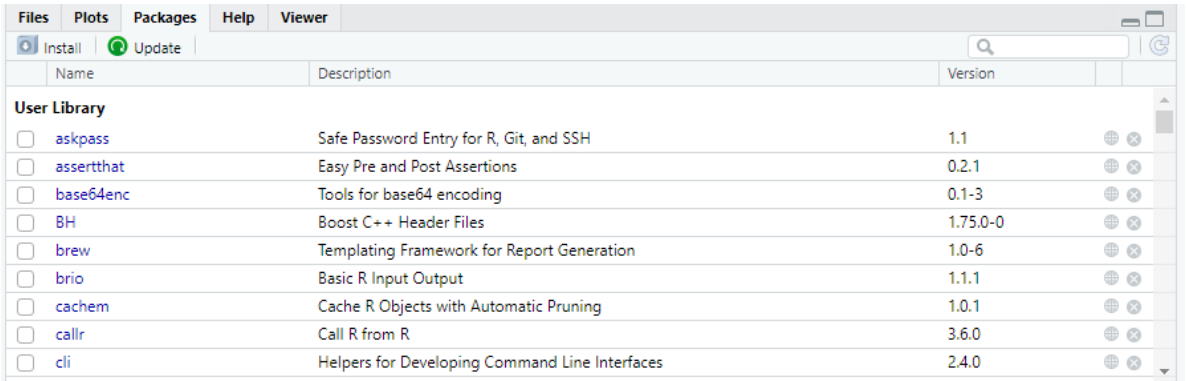

Figure 6. 4 Example of available R packages from Rstudio

A working version of the R algorithm was written in an R script and loaded on the Rstudio platform. Some of the steps required in fetching and preparing the data from the TCGA are further presented.

Firstly, the necessary packages and libraries were installed and loaded on R. *Devtools* represents a general package used in installing other packages such as *Rcpp* as presented in the next section of this chapter. Once the packages are installed, the *library()* function performs the loading of them, from this point the user being able to use the functionality of the libraries. These instructions can be observed in *Figure 6.5.*

```r
# Installing the packages
install.packages("fastcluster")
install.packages("devtools")
library(devtools)
install_github("pievos101/HC-fused")

# Loading the libraries
library(HCfused)
library(fastcluster)
```

Figure 6. 5 Walkthrough for how to install and load packages in R

Next, the data from the TCGA, which was prior downloaded and saved in the same folder as the working directory from R, is being read and saved in the Rstudio Environment. Afterwards it is being fed as an argument to the *HC_fused_calc_network()* function which will not be included in the acceleration process. This function returns a binary matrix – a structured network which will act as an input to the hierarchical clustering algorithm discussed in this project. The steps are written in *Figure 6.6.*

```r
# mRNA
mRNA <- read.table("BREAST_Gene_Expression.txt")
mRNA <- t(mRNA)
dim(mRNA)

# Methy
Methy <- read.table("BREAST_Methy_Expression.txt")
Methy <- t(Methy)
dim(Methy)

# Get the network (binary) structured data (n.patients x n.patients)
omics        <- list(mRNA, Methy)
networks     <- HC_fused_calc_NETWORK(omics)
```

Figure 6. 6 Reading the datasets and obtaining the binary network

The actual functionality of the algorithm written in R will be assessed in the following sections, where a C++ translation of it will also be available.

Lastly, an example of a call to the top function of the method is shown, altogether with one of the ways that the timing of the function was computed during this project. The result of the function was saved in the *res* variable and then saved as a *.csv* file to be further compared to the other versions of the code. It is important to mention that the code was made deterministic from the beginning of the project, by removing the random factor from the method. As a result, a certain output is always expected and easily checked with other versions. A call example is displayed in *Figure 6.7.*

```r
Nruns <- 10
start_time <- Sys.time()
res   <- HC_fused_new(networks_listed, n.iter = Nruns)
end_time <- Sys.time()

end_time - start_time
```

Figure 6. 7 Calling top function and recording the time

6.1.2 Rcpp library

One of the disadvantages of *R* is the execution time of the code, which is also longer than similar programming languages that are capable to perform the same statistical analysis, plotting and operations such as Matlab and Python. In order to overcome this, R had to be combined with another programming language which was able to speed up the execution process.

C or C++ natively offer the improvements that the algorithm needed in order to decrease the execution time of the method, while also scaling accordingly with the number of patients and types of data. The main improvement comes from the compilers which transform the C/C++ code into machine code, process which greatly improves the execution time of a code.

Amongst others, C is a language in which R was initially created. Naturally, C code can already be created inside R without the need to extend it with external packages and libraries. However, a great downside to this, is the necessity to also create wrappers in order to invoke certain data types and functions, besides the actual C code. These wrappers basically tell R how to deal with the code written in C. This capability of integrating C inside R is usually used to improve execution time of certain parts of R code that are running slow, such as *for* loops.

One specific library, *Rcpp*, deals elegantly with the interface between R environment and C++. Similar to C, C++ is able to greatly accelerate the computations that R struggles with. In addition, *Rcpp* solves the integration problem between the two programming languages by automatically creating wrappers for the existing data types and for the user-created functions. However, any additional data type, such as *structs* and *classes*, require manual wrapping to R. In some of the versions that will be presented in the next subsections of this chapter, a *struct* was created. In order to allow the integration with C++, it was wrapped as in the following snip of code from *Figure 6.8*:

```
namespace Rcpp {
 template <>
 SEXP wrap(const ids_vec_mat& x);
}

#include <Rcpp.h>
using namespace Rcpp;

namespace Rcpp {
  template <>
  SEXP wrap(const ids_vec_mat& x) {
    Rcpp::NumericVector ids;
    Rcpp::NumericVector ids2;
    return Rcpp::wrap(Rcpp::List::create(Rcpp::Named("ids") = Rcpp::wrap(x.ids),
                                         Rcpp::Named("ids2") = Rcpp::wrap(x.ids2)));
  };
}
```

Figure 6. 8 Example of wrapping a newly created struct using the Rcpp namespace

Firstly, prior to using *Rcpp* capabilities, it needs to be downloaded and installed from the *CRAN* website. This was performed for Windows and Linux operating systems in this project and both installations were successful. Afterwards the following header files and namespace respectively can be used in the C++ source file: *#include <Rcpp.h>, #include <RcppCommon.h>, using namespace Rcpp*. The library also needs to be loaded inside the R script that calls the C++ functions. This is done by simply running the following command in R: *library(Rcpp)*.

Once the *Rcpp* library is loaded, there are two possible ways to call C++ functions inside R. The first method is to directly write the code as an argument to the *cppFunction()* directly in the R script that needs it. The second method is to create a separate *.cpp* file and source it using *sourceCpp()* function. Due to the relatively large number of functions that should be called, the second option was chosen for this project. Both variants are illustrated in *Figure 6.9* and *Figure 6.10*:

```
cppFunction(
  'int get_sample(NumericVector vec,bool randomize, int vec_valid_size){
   if(randomize == false){
     return vec[0];
   }else{
     return vec[(rand()%vec_valid_size)];
   }
  }')
```

Figure 6. 9 Example of using cppFunction() for implementing a function

```
sourceCpp("HC_fused_cpp_opt6.cpp")
res_cpp_v6 = matrix(unlist(HC_fused_cpp_opt6(networks_large, 10)),nrow=849,byrow = TRUE)
```

Figure 6. 10 Example of using sourceCpp for sourcing a C++ file for further use

Another necessary aspect is the requirement of indicating R which C++ functions need to be exported. A function that is not exported should only be called internally inside the C++ source and cannot be further called from an R script. Above the function that needs to be exported, the following code line must be written: "// [[Rcpp::export]]". An example is shown in *Figure 6.11*:

```
// [[Rcpp::export]]
vector<vector<int>> HC_fused_cpp_opt6(vector<vector<int>> MAT,int n_iter)
{...}
```

Figure 6. 11 Example of exporting a function to R

Through the *Rcpp* library, the user has also the possibility to change the C++ compiler options. Such a thing a performed by creating an R package. For this project, the *Rcpp.package.skeleton* library was used to create such a package. The structure of the package includes among other files a *DESCRIPTION* file used to indicate details about the usage and creation of the package, an R folder in which all relevant R scripts can be found and a source file containing the C++ source files and an optional *Makevars* file. This latter one is responsible for allowing the user to change the compiler settings, such as C++ version and optimization flags.

6.2 Hardware implementation

This section describes the process which aimed to develop a hardware capable of further improving the execution time of the *HC-fused* algorithm. It will present relevant hardware related implementation decision which were taken at certain points during the workflow of designing the hardware accelerators. It will firstly cover the HLS adaptations that took place at the beginning of the hardware design phase, then it will shortly describe the HLS tool which was used to generate the RTLs. Afterwards the design implementation of the 3 IPs will be discussed.

6.2.1 HLS adaptations

The majority of High-Level Synthesis coding constraints were extracted from Xilinx documentation [22] and [23]. The most efficient version of software code was additionally adapted to be integrated inside the HLS software. Firstly the code was divided into a source code and a testbench. The source code is made out of all of the implemented functions, starting with the top one and going to the inner-most ones. The *vector* library and all of the *R* and *Rcpp* ones were removed from the source, as they are not supported. Also the source code file should not contain any allocation of variable size, so nothing based on the input network in this algorithm's case. As a result, every memory space was allocated in the testbench and through the use of *malloc* and pointers from the partially adapted code coming from the software optimization stage, the data is being transferred as arguments to the top function. Additionally, the top function of the algorithm is being called from the testbench and the returning result is also being printed there. For all of the architectures, only one universal testbench was used, the difference between them being only the select top function of the architecture. Other unsupported elements in Vitis HLS include:

- No system calls inside the source file. Printing is done from the testbench only in this project.
- No dynamic memory usage in the source file. This has been handled by allocating everything in the testbench, thus the required resources are prior specified.
- Limited pointer usage and no function pointers. Vitis only allows pointer casting only between native C++ types.
- Recursive functions are not supported since they are not synthesizable. In this project *while* loops were converted into *for* loops.
- Standard Template Libraries which contain recursion or dynamic memory allocations are not supported.

High-level synthesis can be viewed as an efficient alternative to HDLs such as Verilog and VHDL, however taking into account the restrictions above. Some of the advantages of HLS were extracted from [22]:

- Developing and validating algorithms at C-level, having an abstract level of hardware implementation.
- Using C-simulation to check design, thus being more quickly than validating the traditional RTL design.
- Using directives (pragmas) to control the synthesis and create performant implementations.
- Ability to create several solutions based on the source code and the pragmas and find optimal solution based on the design space exploration.
- Ability to quickly recompile source in order to target different hardware's.

The workflow of HLS software's such as Vitis is presented in *Figure 6.12*. It illustrates how the HLS software is fed the algorithm altogether with the viable C++ libraries and user-defined pragmas and it produces multiple solutions which include scheduling, RTL designs and resources utilization.

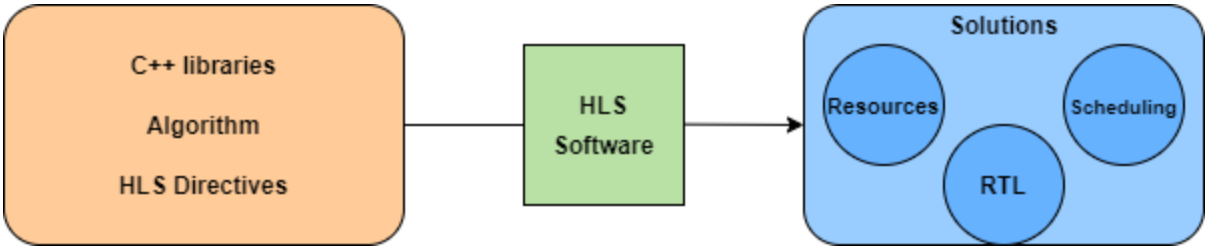

Figure 6. 12 HLS workflow

Besides the above workflow, according to Xilinx [22], an HLS software also includes the following 3 stages:

- **Scheduling** – clock cycles are assigned an operation based on dependencies, clock frequency, operation execution time, available resources and user defined directives.
- **Binding** – scheduled operations are assigned hardware resources.
- **Control logic extraction** – generates a finite state machine that sequences operations based on the schedule.

6.2.2 Vitis HLS

Given the document from Xilinx [22], Vitis HLS represents a novel software which allows through the use of its high-level synthesis tool, the generation of RTL based on C, C++ or OpenCL source code. It deals with most of the changes required to implement and optimize the source code while also having the overall goal of achieving a high throughput.

The 2020.2 software version was used for this project. After creating an HLS project and uploading the source files and testbench file, the usual sequence of processes is the following. Perform C simulation and verify that the source code acts as expected after applying the HLS constraints. If the results are correct then HLS should be run in order to generate the RTL files and synthesis reports. These output files have to be analyzed and the following parameters assessed:

- Latency – amount of clock cycles needed to output all values.
- Initiation Interval (II) – amount of clock cycles needed to initiate a new set of input data and restart the computation process.
- Loop iteration latency – amount of clock cycles each loop iteration requires.
- Pipelined – indicates if the function or loop are pipelined in the design.
- Slack – timing slack for the function or loop. A negative slack signifies that the design cannot work correctly.
- Tripcount – indicates a loop's number of iterations.
- Resource Utilization – indicates the hardware consumption: number of BRAM, DSP, LUT and FF implemented.

Generally many iterations of C-Simulation and HLS synthesis are needed to increase the performance of the solutions. Changes may occur either in the C++ source file or testbench or in the available directives that are applied to the code, such as loop unrolling or memory interfaces.

6.2.3 Implementing the Similarity Matrix calculation design

Initially, a hardware design was to be made to the outer function – the one which was calculating the distances between patients and which was calling the inner function to obtain a mean value based on the selected patients. In order to get relevant synthesis results, the number of iterations for each of the 12 loops involved in the function had to be fixed to a constant highest potential value instead of a variable one. Unfortunately, the results were extremely poor in terms of execution time and as a result it was decided to generate hardware for the inner function which now was going to be called the exact necessary amount of time. After proceeding with the same approach, the results were still not good enough to be able to accelerate the design compared to the software version. The final solution found to this problem was to implement an additional function inside the previously inner-most one to have only 1 loop altered with a maximum number of iterations possible instead of 2.

After setting up the optimal top function for this method in the synthesis settings, memory interfaces were selected using the *HLS INTERFACE* pragma which allows the user to choose from a series interface protocols with memories, such as FIFOs, AXI4 or RAMs. For this implementation the RAM memories were selected due to the increased performance that was observed compared to the others. The directive for such an interface is:

```
#pragma HLS INTERFACE ap_memory port=network
```

Afterwards, the target was to decrease the execution time of the design as much as possible, while fulfilling the constraints presented in the Architecture design section 5.3. Consequently, different directives were tried for the *loops* which contained the majority of computations, for example unrolling or pipelining them.

One of the lasts and most efficient directive that was applied is the *DATAFLOW* one which allowed the execution in parallel. As a requirement however, all of the declarations had to be made before the pragma and all of the computations after it. Another additional necessity was the creation of copies for variables used inside those computational blocks – this being the main reason why incrementing the parallel level also increased the amount of hardware resources. Lastly, the computations were divided between the additionally created loops in order to share the workload and obtain a faster execution.

6.2.4 Implementing the merging clusters indexes extraction designs

For this method, two different designs were implemented. This was done so as to exploit at maximum the parallelism potential and to fulfill the memory bandwidth requirement. These two parts could not work in parallel altogether since the latter one depends on the results of the first one, thus a parallel execution would cause an incorrect output if not done in a sequential manner.

The first design has the scope of going through the Similarity Matrix, find the maximum value out of it and store in inside the memory. Furthermore, the second one has to use this value and compare it to all of the elements from the Similarity Matrix. If a match is found, then the results are stored using pointers inside the memory.

The interface with the memory was set to a RAM one for both designs, proving the best results compared to others. Concerning the hardware optimizations, the loops were pipelined and parallelized using the *PIPELINE* and *DATAFLOW* pragmas, respectively. Similar to the previous design, copies of variables were needed so as to achieve execution parallelism between the separated loops which were splitting the workload.

7. Performance evaluation

In this section the overall performance improvement of the *HC-fused* hierarchical clustering algorithm is presented. It will first describe the experimental setup of the project. Afterwards it will discuss the software results that were obtained through the aforementioned designs and implementations and it will continue with the hardware ones. Finally, the efficiencies of the software implementation and the HLS accelerator design are being combined to obtain the potential total acceleration of the method. Since HLS designs were only implemented for the smaller dataset, only this acceleration will be assessed.

7.1 Experimental setup

For the measurements performed during this project only two datasets were used. The data was first collected from the TCGA [2] and then adapted to be further used in this algorithm, as done by Pfeifer in [1]. The first dataset, the smaller one, consisted of 105 patients and two types of omics, namely mRNA and Methy. The larger one contained 849 patients and three types of omics: mRNA, Methy and miRNA.

These datasets were applied to the hierarchical clustering and data fusion algorithm. The software performance was recorded for both the R and the C++ versions. These versions were run on an x64 based Intel Core i7-4720HQ CPU with a frequency of 2.60 GHz. Furthermore, the targeted device for the hardware implementation was the Zynq Ultrascale+ ZCU102 evaluation board. This board is characterized by a quad-core Arm Cortex-A53, dual-core Cortex-R5F real-time processors, and a Mali-400 MP2 graphics processing unit.

7.2 Software performance

This subsection will create an overview of the entire software design process, indicating the most relevant results obtained, the fulfilled tasks during this process and it will shortly revise the required softwares or libraries that were needed in order to perform the software implementation.

The R version of the hierarchical clustering and data fusion algorithm was successfully converted to C++. Using the *Rcpp* library, the C++ source files were integrated with the R software, datasets being able to be fed to the algorithm from a different environment than C++. All of the C++ versions were checked for the correct results and their timing was measured.

An R package was created, containing all of the C++ versions, R usage examples, tested datasets and more. Afterwards the package was loaded into R and successfully showed the correct functionality. An additional *Makevars* file was created in the source file of the package, allowing different compiler options to be chosen. The differences could be seen when loading up the package. The package was tested directly in the command prompt console but also in the *Rstudio*. Additionally, its functionality was tested on *Windows OS* and *Linux*, both successful.

All of the results from the software optimization stages have been summarized in Table 4. The first 5 versions of code were only fed the smaller dataset of 105 patients and two omics. The R version together with the last two versions of C++ code – which showed the best overall results in execution time – were also fed as input the larger dataset consisting of 849 patients and three omics. From each

version to the other, a short description of the improvements applied is shown. Also the execution time of all versions is indicated and compared to the initial one. Additionally, to have a better view of each stage's improvement, the forth column of the table shows the relative acceleration compared to the previous version of the algorithm. These results show a continuous improvement for the C++ versions for the execution time of the method.

Version	Language	Improvements	Performance improvement per stage	105 patients mRNA, Methy time	105 patients Acceleration relative to R version	849 patients mRNA, Methy, miRNA time	849 patients Acceleration relative to R version
Initial R	R	-	1x	78.45 sec	1 x	10.98 hours for 1 iteration	1 x
Cpp v1	C++	R to C++	0.28 x	281.14 sec	0.28 x	-	-
Cpp v2	C++	Less and faster functions	16.12 x	17.44 sec	4.5 x	-	-
Cpp v3	C++	3D and 2D variables, Less *for* loops	1.08 x	16.14 sec	4.8 x	-	-
Cpp v4	C++	*For* loops instead of slow function	3.07 x	5.25 sec	15 x	-	-
Cpp v5	C++	Improved *clusters* merge and *network* update	1.13 x	4.65 sec	17 x	-	-
Cpp v6	C++	Merged functions, 2D to 1D variables	3.57 x	1.3 sec	60 x	-	-
Cpp v7	C++	Lists to vectors Less *for* loops	2.13 x	0.61 sec	128 x	10.91 minutes for 10 iterations	604 x
Cpp v8	C++	Adapt to HLS Use of *malloc* and pointers	6.1 x	**0.1 sec**	**784 x**	**1.95 minutes for 10 iterations**	**3384 x**

Table 4 Software results overview

7.3 Hardware results

The HLS design phase has produced three separate IP cores. The first one is able to accelerate a part of the function which computes the Similarity Matrix in the algorithm, namely the *getmean_inner* function, being able to run 1 execution in 107 nanoseconds and having a nominal acceleration ratio of 1.36.

For this architecture, the parallelism parameter *P* was set to values ranging from 1 to 6 in order to increase performance. During the performance measurements the timing, required memory bandwidth and resources consumptions were observed to correctly identify the optimal design point. In Table 5 the timing related values are indicated. Since the software was executing on average in about 146 ns, each design point capable of running in less than that represents an improvement for the scope of this project.

Parallelism parameter P	Estimated period [ns]	Initiation Interval	Total interval	Latency [cycles]	Pipelined	Bandwidth [GB/s]	Time [ns]
P = 1	2.007	1	108	108	Yes	5.81	216
P = 2	2.854	1	56	56	Yes	7.88	159
P = 3	2.569	1	38	38	Yes	12.90	98
P = 4	3.585	1	30	30	Yes	11.71	107
P = 5	3.300	1	24	24	Yes	15.90	79.2
P = 6	3.585	1	21	21	Yes	16	75.2

Table 5 Timing of designs for computing the Similarity Matrix. The optimal design point is marked with the color green, while the exceeding bandwidths are marked with the color red

The results from Table 5 indicate that the optimal solution is represented by the design point with the parallelism parameter *P* equal to 4, offering an execution time of 107 us. This value, compared to the average software execution is only 1.36 times faster. The main disadvantages that did not allow for a faster execution time were the constraint of having a constant number of iteration, equal to the nominal possible value (constraint which caused more functional-unnecessary computations) and the bandwidth which limited the number of transfers with the memory during one second.

Parallelism parameter P	BRAM	DSP	FF	LUT	URAM
P = 1	0	0	595	546	0
P = 2	0	0	1078	1080	0
P = 3	0	0	1482	1526	0
P = 4	0	0	1884	1983	0
P = 5	0	0	2287	2428	0
P = 6	0	0	2691	2888	0

Table 6 Hardware utilization for computing the Similarity Matrix. The optimal design point is marked with the color green

The resources utilization for all of the six design points are described in the Table 6 and illustrated in the bar chart from Figure 7. 1. For this architecture the resources utilization did not represent a constraint in choosing the optimal solution as the percentage of utilized resources was around 1% of the total hardware available. This is mainly due to the relatively small designs and large number of iterations that are required for this method.

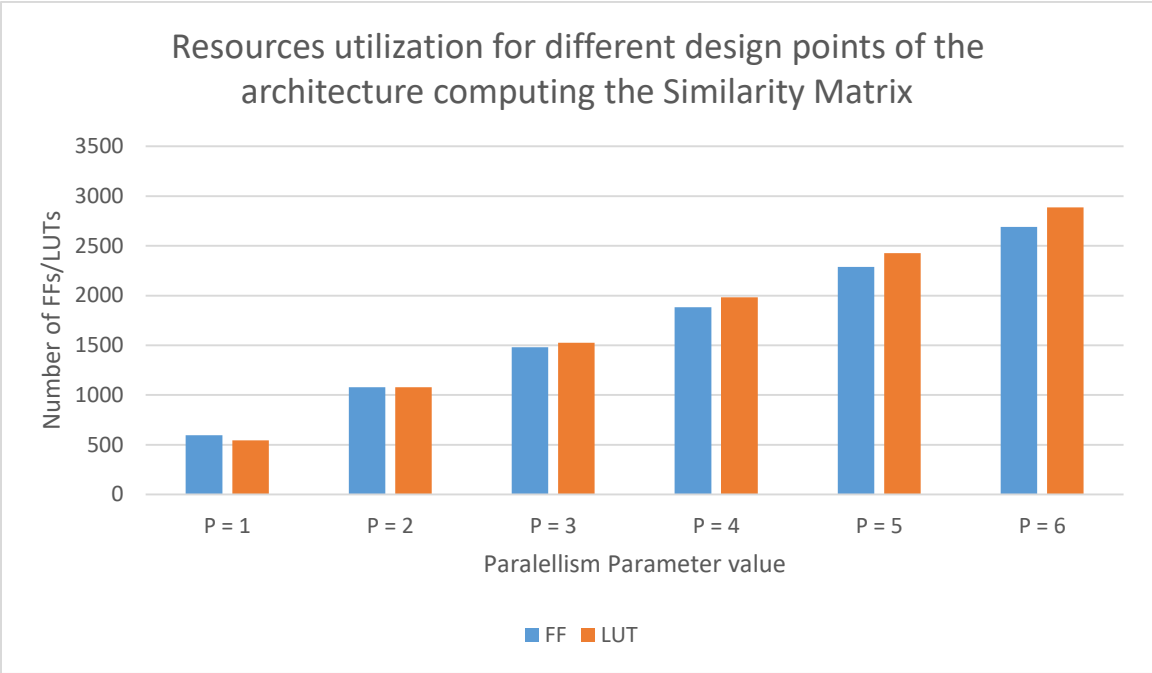

Figure 7. 1 Resources utilization of the architecture computing the Similarity Matrix

The second and third accelerators deal with the indexes extraction from the Similarity Matrix, the *which_vec_mat* function. Both of the designs execute in a total of 40 nanoseconds, being able to improve execution time of the method 1.5 times.

The first out of this two deals with the computing of the maximum value from the Similarity Matrix. The design space exploration was made for the parallelism parameter P to find out which design point is optimal for executing the operations while maintaining a frequency below the selected memory bandwidth of 12 GB/s. Below, in Table 7 the timing results of this subsection are being presented. The parallelism parameter P does not seem to affect greatly the estimated working frequency of the hardware, which is of about 158 MHz. Additionally the architecture allows for data to be feed every cycle to the input. An improvement is observed in the total interval and latency of the hardware, as increasing *P* steadily causes these factors to decrease and thus making the computations faster. However the cost of this speed is the additional hardware that it is needed to perform such operations in parallel. In the case of this function, an additional port of 64 bits is added up for each incrementing of the parallelism parameter *P*. This alongside the decreasing latency of the design points directly affect the required Bandwidth of the memory which is supposed to support these transfers. It was found that a value of *P* equal to 4 causes a bandwidth of about 15.82 GB/s, value which exceeds the imposed constraints of the memory. Consequently, this makes the design point with the parallelism parameter equal to 3 the optimal one.

Parallelism parameter P	Estimated period [ns]	Initiation Interval	Total interval	Latency [cycles]	Pipelined	Bandwidth [GB/s]	Time [us]
P = 1	6.056	1	5464	5463	Yes	3.96	33
P = 2	6.343	1	2738	2733	Yes	7.58	16
P = 3	6.343	1	1827	1823	Yes	11.31	11
P = 4	6.343	1	1373	1368	Yes	15	8

Table 7 Timing results of accelerated function finding the maximum value from Similarity Matrix. The optimal design point is marked with the color green, while the exceeding bandwidth is marked with the color red

Table 8 describes the hardware utilization of the design. Naturally, increasing the parallelism parameter also increases the amount of utilized hardware. Even though the amount of flip-flops and look-up tables has increased, the total usage of resources is extremely low due to the relatively small design. Additionally, no block RAMs, DSPs or URAMs were needed for this architecture.

In Figure 7. 2 the resources utilization of the 4 design points are illustrated in a bar chart.

Parallelism parameter P	BRAM	DSP	FF	LUT	URAM
P = 1	0	0	975	1277	0
P = 2	0	0	1849	2621	0
P = 3	0	0	2777	3956	0
P = 4	0	0	3710	5245	0

Table 8 Resources of the accelerated function finding the maximum value from Similarity Matrix. The optimal design point is marked with the color green

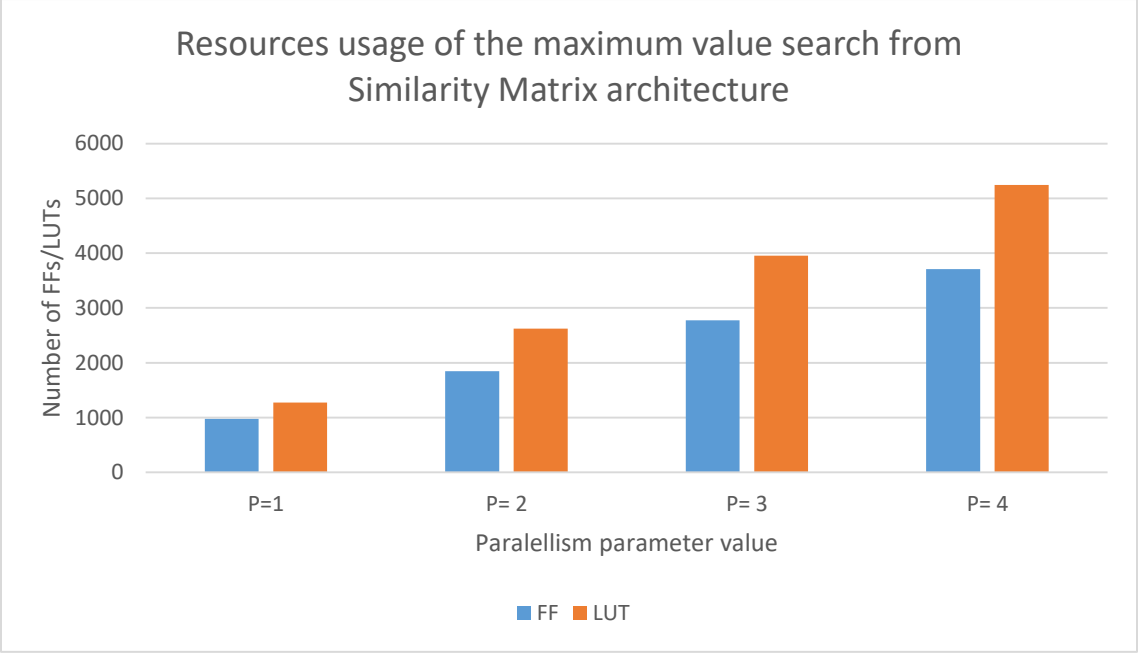

Figure 7. 2 Resources usage of the maximum value search architecture

The last accelerator is used to obtain the indexes of the potential merging clusters. The design space exploration was made for this architecture, having the same constraints as the previous two architectures. The results for each design point are presented in Table 9. The software version was executing this method in about 60 us on average. Since in the hardware design process this method was divided into two separate blocks (one finding the maximum and one finding the indexes) their times have to add up and still be less than the 60 us in order to be qualified as an acceleration. Similar to the other two architectures, a critical downside was the requirement to set a constant number of iterations for one of the loops of the method. In other words, instead of only going through the relevant part of the Similarity Matrix and perform those computations, the hardware design is required to always go through all of it and compute operations which are unnecessary, thus decreasing its time efficiency. Nevertheless, the optimal design point, the one with a parallelism parameter of value 4, shows good performance in terms of timing, being able to execute in only 29 us while also maintaining the memory bandwidth requirements.

Parallelism parameter P	Estimated period [ns]	Initiation Interval	Total interval	Latency [cycles]	Pipelined	Bandwidth [GB/s]	Time [us]
P = 1	7.024	1	16383	16383	Yes	2.84	115
P = 2	7.085	1	8193	8194	Yes	5.64	58
P = 3	7.088	1	5463	5464	Yes	8.45	39
P = 4	7.088	1	4098	4099	Yes	11.27	29
P = 5	7.088	1	3279	3280	Yes	14.09	23

Table 9 Timing results of the indexes extraction design points. The optimal design point is marked with the color green, while the exceeding bandwidth are marked with the color red

The resources utilization of these design points are showed in Table 10 and illustrated in the bar chart from Figure 7. 3. As in the other two architectures, the number of flip flops and look-up tables increase concurrently with the increment of the parallelism parameter. On the other hand, the design points from this architecture are the only ones who also involve digital signal processors, these being in number equal to the parallelism parameter.

Parallelism parameter P	BRAM	DSP	FF	LUT	URAM
P = 1	0	1	503	852	0
P = 2	0	2	993	1586	0
P = 3	0	3	1483	2303	0
P = 4	0	4	1970	3022	0
P = 5	0	5	2461	3747	0

Table 10 Resources utilization of the indexes extraction design points. The optimal design point is marked with the color green

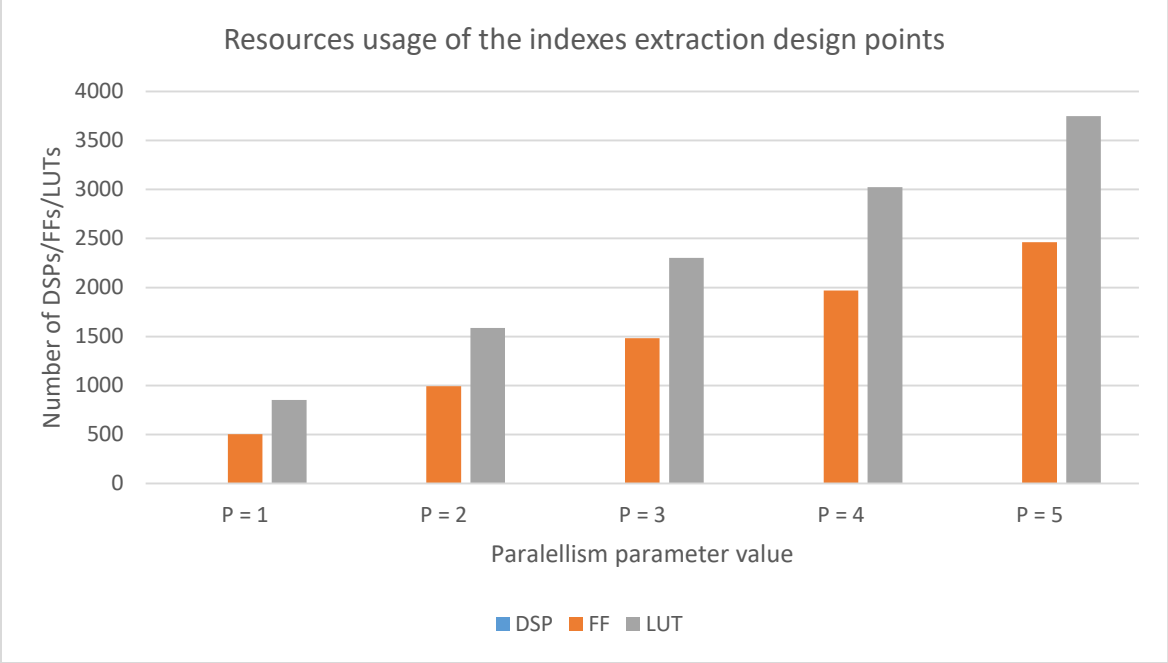

Figure 7. 3 Bar chart for resources utilization of the indexes extraction design points

By combining the results from the profiling process and the ones from the accelerators designs the overall execution time of the top function *HC-fused* will decrease. The method for computing the Similarity Matrix will run 10 iterations of the algorithm in 17.3 milliseconds instead of 23.6 milliseconds, while the one responsible for extracting the indexes equal to the maximum value from the Similarity Matrix will run 10 iterations in 24.6 milliseconds instead of 37 milliseconds.

By summing up these differences, a total of 18.7 milliseconds are saved for 10 iterations which were recorded to run in the software version in about 100 milliseconds. This difference is directly proportional with the number of iterations which the algorithm runs. As a result, the hardware acceleration will also scale with the number of iterations that the algorithms runs.

7.4 Overall timing improvement

After the software implementations, the fastest software version written in C++ was able to run 10 iterations of the dataset containing 105 patients and two types of omics in 0.1 seconds, being 784 times faster than the original version.

By summing up all of the execution times now, the execution time is of only 81.3 milliseconds, thus being 965 times faster than the initial algorithm's execution time of 78.45 seconds.

Given the fact that the software optimizations improved greatly the scaling of the method with the number of patients and number of omics, the overall execution time for larger datasets should show better results in terms of efficiency by also combining the acceleration. The newly designed hardware accelerators provide the main advantage of parallelism of computation and the main disadvantage of performing several additional computations which are not further used in the algorithm's functionality.

8. Conclusions and future work

In this report, a series of optimizations were proposed, having the scope of increasing the execution time efficiency of a performant hierarchical clustering and data fusion algorithm. In order to reduce the execution time of the algorithm, the code was first ported to a C++ version which was gradually improved using different software optimizations. With the aim of designing hardware accelerators to further improve the speed of the method, the C++ code was adapted to be used in generating RTL via High-Level Synthesis. The hardware designs were made compatible with the dataset of 105 patients and 2 omics, the resulting execution time being almost 1000 times faster than the original one.

8.1 Addressing the research questions

Initially, the algorithm was implemented in R language, a programming language with the disadvantage of interpreting the code at runtime, causing it to execute slowly. To overcome this, it was decided to implement the code in C++ instead, in order to make us of its compilers and faster execution. Based on this proposed solution, the following research questions was formulated:

What is the performance potential of porting the R code to C++?

The software conversion from R to C++ proved to be extremely beneficial to the overall execution time of the algorithm. Due to the C++ compiler and the successive stages of optimizations performed upon the code, it was possible to run the algorithm in far less time than in the previous state. For the two datasets, acceleration ratios of 784 and 3384 were obtained when applying the small datasets and the large dataset, respectively, to the hierarchical clustering algorithm.

For the purpose of achieving great efficiency for the future tests and applications of the C++ implementation of the algorithm, the method's scalability had to be taken care of. Ultimately, the algorithm will be involved with large datasets and its execution time should be relatively efficiently appropriate in terms of execution time, to the data size. Consequently, the ensuing research question was made:

How does the C++ code version of the algorithm scale with the number of patients and the number of omics?

Based solely on the execution results of the available two datasets, the scaling with the number of patients and the number of omics seems to no longer represent an impediment to the utilization of the algorithm. Measurements of the initial R version show the execution time inefficiency when applying large datasets to the method. However, now using the C++ version, tests of the algorithm were easily obtained for the same large dataset.

At some point, the software optimizations will no longer be able to greatly affect the execution time of the algorithm. As a consequence, a hardware architecture capable of executing the code in a faster manner using various hardware techniques was made. Based on this, the next research question was formed:

> **What impact does the hardware acceleration via an FPGA have on improving the execution time of the clustering algorithm?**

The hardware optimizations proved to be only slightly beneficial to the goal of decreasing the execution time. A total of 3 architectures were created, 1 for the computing of the Similarity Matrix and 2 for generating the indexes of merging clusters. Using the optimal designs of these architectures, the execution time of these computations were decreased by 1.36 times and 1.50 times, respectively. The relatively small accelerations were a consequence of the limitation imposed by the memory bandwidth, as the nature of the algorithm involved lots of memory accesses and not too many computations. These architectures were chosen to be created based on the profiling process which showed which inner functions were executing in the most time. The profiling results also showed that the percentages of execution time for these functions increased with the number of patients and omics. As a result, the scaling efficiency is also improved by using the designed accelerators.

8.2 Future work and improvements

Regarding future work perspectives, a better understanding of the scalability potential of the improved algorithm may be obtained by applying more diverse datasets, containing increased numbers of patients and data omics. Additionally, to take advantage of the hardware acceleration capabilities, a fixed number of patients can be found in order to make the accelerators universal to all future applied datasets. It is important that this number of patients is always considered enough to provide an accurate and performant clustering result. Furthermore, additional investigation should be made upon how to increase the acceleration of the algorithm using hardware techniques. One great disadvantage in terms of speed was given by additional computations caused by the fixed number of iterations of the loops, a necessary step for implementing the design. Moreover, the bandwidth requirements could be fulfilled with more parallelism involved by assigning data to reduced-size data types which are capable of performing the same functionality, while decreasing the transfers' sizes with the memory.

Since the original algorithm obtained the greatest results for kidney renal clear cell carcinoma, liver hepatocellular carcinoma, skin cutaneous melanoma, ovarian serous cystadenocarcinoma and sarcoma, it comes naturally that the implementations resulted from this project will also be applied to the same causes. This is due to the unaffected functionality of the algorithm, which produces the same output but in a much shorter execution time – a result of the software and hardware optimizations performed in this project.

References

[1] B. Pfeifer and M. G. Schimek, “A hierarchical clustering and data fusion approach for disease subtype discovery,” *J. Biomed. Inform.*, vol. 113, 2021, doi: 10.1016/j.jbi.2020.103636.

[2] K. Tomczak, P. Czerwińska, and M. Wiznerowicz, “The Cancer Genome Atlas (TCGA): an immeasurable source of knowledge.,” *Contemp. Oncol. (Poznan, Poland)*, vol. 19, no. 1A, pp. A68-77, 2015, doi: 10.5114/wo.2014.47136.

[3] F. Nie, S. Shi, and X. Li, “Auto-weighted multi-view co-clustering via fast matrix factorization,” *Pattern Recognit.*, vol. 102, 2020, doi: 10.1016/j.patcog.2020.107207.

[4] F. Murtagh and P. Legendre, “Ward’s Hierarchical Agglomerative Clustering Method: Which Algorithms Implement Ward’s Criterion?,” *J. Classif.*, vol. 31, no. 3, pp. 274–295, 2014, doi: 10.1007/s00357-014-9161-z.

[5] S. Aranganayagi and K. Thangavel, “Clustering Categorical Data Using Silhouette Coefficient as a Relocating Measure,” in *International Conference on Computational Intelligence and Multimedia Applications (ICCIMA 2007)*, 2007, vol. 2, pp. 13–17, doi: 10.1109/ICCIMA.2007.328.

[6] B. Wang *et al.*, “Similarity network fusion for aggregating data types on a genomic scale,” *Nat. Methods*, vol. 11, 333–33, 2014, [Online]. Available: https://www.nature.com/articles/nmeth.2810.

[7] T. Nguyen, R. Tagett, D. Diaz, and S. Draghici, “A novel approach for data integration and disease subtyping,” *Genome Res.*, 2017, [Online]. Available: https://genome.cshlp.org/content/27/12/2025.

[8] N. Rappoport and R. Shamir, “NEMO: cancer subtyping by integration of partial multi-omic data,” *Bioinformatics*, vol. 35, no. 18, 15, pp. 3348–3356, 2019, [Online]. Available: https://doi.org/10.1093/bioinformatics/btz058.

[9] F. W. Wibowo, Sudarmawan, and M. Sulistiyono, “Hardware Plaftorm Design Analysis of K-Means Clustering Algorithm Implementation,” *Int. J. Eng. Technol.*, vol. 7, No. 4.4, 2018, [Online]. Available: https://www.sciencepubco.com/index.php/ijet/article/view/24082.

[10] P. Berkhin, “A Survey of Clustering Data Mining Techniques,” *Kogan J., Nicholas C., Teboulle M. Group. Multidimens. Data. Springer - Verlag Berlin Heidelb.*, vol. Grouping M, pp. 25–71, 2006, [Online]. Available: https://doi.org/10.1007/3-540-28349-8_2.

[11] R. Ahuja, A. Chug, S. Gupta, P. Ahuja, and S. Kohli, “Classification and Clustering Algorithms of Machine Learning with their Applications,” *Yang XS., He XS. Nature-Inspired Comput. Data Min. Mach. Learn. Stud. Comput. Intell. Springer, Cham*, vol. 855, 2020, [Online]. Available: https://doi.org/10.1007/978-3-030-28553-1_11.

[12] M. Keuper, S. Tang, B. Andres, T. Brox, and B. Schiele, “Motion Segmentation & Multiple Object Tracking by Correlation Co-Clustering,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 42, no. 1, 2020, [Online]. Available: 10.1109/TPAMI.2018.2876253.

[13] A. Baraldi and P. Blonda, “A survey of fuzzy clustering algorithms for pattern recognition,” *IEEE Trans. Syst. Man, Cybern. Part B*, vol. 29, no. 6, 1999, [Online]. Available: 10.1109/3477.809033.

[14] A. S. S.A. and M. Dash, “Efficient Partitioning Based Hierarchical Agglomerative Clustering

Using Graphics Accelerators With Cuda," *Int. J. Artif. Intell. Appl.*, vol. 4, pp. 13–33, 2013, doi: 10.5121/ijaia.2013.4202.

[15] D. Legorreta, M. Looks, and S. Padmanabhan, "A Hardware Implementation of Hierarchical Clustering for Text Documents." [Online]. Available: https://www.isi.edu/~youngcho/cse560m/finalpres/hwclust.pdf.

[16] D. Behrens, K. Harbich, and E. Barke, "Hierarchical partitioning," in *Proceedings of International Conference on Computer Aided Design*, 1996, pp. 470–477, doi: 10.1109/ICCAD.1996.569862.

[17] P. Jones *et al.*, "Liquid Architecture," *18th Int. Parallel Distrib. Process. Symp.*, 2004, [Online]. Available: 10.1109/IPDPS.2004.1303228.

[18] A. Page, N. Attaran, C. Shea, H. Homayoun, and T. Mohsenin, "Low-Power Manycore Accelerator for Personalized Biomedical Applications," *GLSVLSI '16, May 18-20*, 2016, [Online]. Available: https://doi.org/10.1145/2902961.2902986.

[19] J. S. Sekhon, "Multivariate and Propensity Score Matching Software with Automated Balance Optimization: The Matching package for R," *J. Stat. Softw.*, [Online]. Available: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=1009044.

[20] P. Coussy, D. Gajski, M. Meredith, and A. Takach, "An Introduction to High-Level Synthesis," *Des. Test Comput. IEEE*, vol. 26, pp. 8–17, Sep. 2009, doi: 10.1109/MDT.2009.69.

[21] W. N. Venables, D. M. Smith, and R Core Team, "An Introduction to R," version 4.1.0, 2021. [Online]. Available: https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf.

[22] Xilinx, *Vitis High-Level Synthesis User Guide UG1399 (v2020.2)*, March 22. 2021.

[23] Xilinx, "Vitis Unified Software Development Platform 2020.2 Documentation," 2020. https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/vitis_hls_coding_styles.html.

Appendix 1

In this Appendix, the fastest software version of the algorithm written in C++ is being presented. This version was further used in the developing of the accelerator hardware architecture, but also for additional tests with various datasets of patients and omics, due to its increased efficiency in time.It consists of a total of 7 functions which are called several times during the run of the method. For each of the functions, its functionality and arguments are being described.

1. ***HC_fused_cpp (MAT, n_iter)***

    - This is the top function of the algorithm, which is given the input for the algorithm and which returns the fused network. It directly calls the function that computes the similarity matrix (*HC_fused_calc_distances_cpp)* and the functions which identify and select the clusters that will merge (*which_vec_mat, which_is3, get_sample* and *get_ij*).
    - *MAT* – argument which represents the input structured network to the top function of the algorithm. It is stored as a binary matrix with the first dimension given by the number of omics from the dataset and the second one given by the number of patients included in the dataset. It is passed as an argument via a pointer.
    - *n_iter* – argument which represents the number of iterations that the algorithm will run before outputting the final fused network. It is recommended for better results that this number to be chosen around 100.

2. ***HC_fused_calc_distances_cpp (obj, MAT, matAND, obj_sizes, distances, mat_distances, n_elems, n_patients, n_clusters, col_nr)***

    - This function is used to compute the Similarity Matrix between patients and represents the first important stage of the algorithm. It makes use of the input binary network and the current clusters and its result is further passed to compute the new merging clusters.
    - *Obj* – argument representing the dynamic clusters of patients. It is passed as a pointer to the memory locations containing the updating clusters.
    - *MAT* – argument identical to the input binary matrix that is fed to the top function of the algorithm. This data is only used inside this function, being further passed to the *getmean* function to generate the distances between patients.
    - *matAND* - argument passed as a pointer, containing the bitwise logic *and* operation of the elements from *MAT*. This data aids in creating a more stable result for the similarities matrix.
    - *obj_sizes* – argument passed as a pointer which indicates the number of patients in each cluster.
    - *Distances* – argument passed as a pointer representing the Similarity Matrix between patients. Elements for this allocation are of type *double*. This is the main output of this function which will be further used in calculating the merging clusters.
    - *mat_distances* – argument passed as a pointer, also of type *double,* used to store the intermediate results from the *getmean* function.
    - *n_elems* – argument indicating the number of omics from the input network.

- *n_patients* – argument which represents the number of patients from the dataset.
- *n_clusters* – argument representing the dynamic number of clusters from the current iteration
- *col_nr* – argument of integer type, representing the shortcut result of combinations of *n_patients* taken by 2. It is used to go through the columns of the Similarity Matrix

3. *getmean (i, j, MAT, elem, obj, obj_sizes, n_patients)*

- The *getmean* is called several times inside the *HC_fused_calc_distances_cpp* function and is able to calculate an average value from a matrix formed by two vectors representing elements from lines and from columns respectively.
- *i* – Integer argument used to indicate the selected line used together with *obj* and *obj_sizes*.
- *j* – Integer argument used to indicate the selected line used together with *obj* and *obj_sizes*.
- *MAT* – argument passed as pointer, representing the input binary matrix.
- *elem* – integer argument used to select data for a specific omic from the input binary matrix.
- *obj* – argument passed as pointer, containing the dynamic clusters of patients.
- *obj_sizes* – argument passed as pointer, containing the actual sizes of clusters.
- *n_patients* – integer argument indicating the number of patients from the dataset.

4. *which_vec_mat (distances, res_vec, res_mat, ids_valid_size, n_clusters, n_elems, col_nr)*

- This function is able to return the 1D and 2D indexes of elements from the Similarity Matrix equal to the maximum element from it. It uses the result stored in *distances* and updates the *res_vec* and *res_mat* with the correct indexes.
- *distances* – argument passed as pointer and gone through to extract the elements with the maximum values.
- *res_vec* – argument passed as pointer, receiving the one-dimensional indexes of elements equal to the maximum one from the Similarity Matrix.
- *res_mat* – argument passed as pointer, receiving the two-dimensional indexes of elements equal to the maximum one from the Similarity Matrix.
- *ids_valid_size* – integer argument which gets updated based on the number of elements found that fulfil the equal to maximum condition.
- *n_clusters* – integer argument indicating the dynamic number of clusters.
- *n_elems* – integer argument representing the number of omics from the dataset.
- *col_nr* – integer argument used to go through the column of the Similarity Matrix.

5. ***which_is3 (res_mat, ids_valid_size, n_elems, is3, is3_valid_size)***

    - This function returns the line numbers of elements from *res_mat* which are equal to the number of rows from the Similarity Matrix. In other words it searches for elements equal to the maximum value from the Similarity Matrix and additionally looks for the ones that were computed based on the *matAND* matrix containing the results from the bitwise logic *and* operation of the input network.
    - *res_mat* – argument passed as pointer, which contains the 2D indexes of elements equal to the maximum value from the Similarity Matrix.
    - *ids_valid_size* – integer argument indicating the valid size that should be used when going through *res_mat*.
    - *n_elems* – integer argument showing the number of elements.
    - *is3* – argument representing the resulting vector of values that fulfil the aforementioned conditions.
    - *is3_valid_size* – integer argument that gets updated with the correct size of the *is3* vector.

6. ***get_sample (vec, randomize, vec_valid_size)***

    - This function return one single element from a given vector.
    - *vec* – argument passed as a pointer, indicating a vector of elements – either *is3* or *res_vec.*
    - *randomize* – bool argument which selects either a deterministic or a non-deterministic selection of the element. During the testing, the deterministic option was chosen in order to check the functionality of the algorithm.
    - *vec_valid_size* – integer argument used to know the number of elements to go through.

7. ***get_ij (id_min, obj_size, map_info_pair)***
    - This last function is used to finally identify the pair of clusters that will merge in the current iteration.
    - *id_min* – integer argument returned from the *get_sample* function, which is further used to find the pair of clusters that will merge, based on a series of computations.
    - *obj_size* – integer argument representing the dynamic number of available clusters for merging.
    - *map_info_pair* – argument passed as pointer that will be updated with the indexes of the merging clusters.