Master Degree in Electronic Engineering

Proximity-based resource sharing in high level synthesis for FPGAs

Roberta Priolo

Supervisors
prof. Luciano Lavagno
prof. Jordi Cortadella

Politecnico di Torino
December 2019

Abstract

Resource sharing is a well known and commonly used method employed during the design of a circuit in order to reduce its area. Usually, this happens to the expense of the delay of the involved path that, if corresponding to the critical path, may affect the minimum clock period.

The goal of this thesis is to prove that applying a smart resource sharing to the biggest units of the circuit, it is possible to achieve better results in terms of both area and clock period.

The smart resource sharing solution that has been employed in this project relies on proximity. The algorithm is based on the belief that a cluster of units placed closer to each other should be replaced by one shared unit, while units that are located further apart should use different resources. This process leads to shorter connection wires that result in less area occupancy and, hopefully, shorter delays.

Acknowledgements

Fist of all, I would like to express my deep gratitude to Professor Luciano Lavagno and Professor Jordi Cortadella, my research supervisors.
Their patience and enthusiasm in guiding me through the months of research have been inspiring. They have always been more than supportive, available for debate and willing to answer my questions.

I am eternally grateful to my parents and my sister Irene for their immense support. You have always encouraged me to take up challenges and not to back off for fear of failure. The path I chose was demanding and I wouldn't have had the courage to take it without you.

Furthermore, I would like to thank the rest of my big family for having always believed in me, even when I didn't.

I would also like to thank my new family in Turin: Francesca and Arianna who shared with me every single day, laughter and tears; Giacomo, Paolo and Giorgio who welcomed me in their home; and Loris, Giovanni and Daniele with whom I started and finished this chapter of my life.

Moreover, I am so thankful to all my lifelong friends from Catania who made me feel their love even from afar, in particular to my best friends Angela, Elisabetta, Claudia, Lucrezia, Ruggero, Valeria and Chiara. I know I can always count on you.

Finally, I want to thank Manfredi for standing by my side all these years and for being my role model and my home. These last years of uni have been really tough, I can't count how many times I felt like giving up, but you have always kept me going and helped me in so many ways I'll never thank you enough for. I am so happy to be sharing this achievement with you.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Electronic systems are becoming increasingly complex and they require ever more power and area occupation. Now that it is no longer possible to rely on Moore's law, it is necessary to employ new methods in order to reduce the number of resources required for a chip. This can result in a smaller flat area of the circuit or in reduced resource usage in an FPGA, and therefore it may be possible to use a smaller one.

One of the most effective methods to tackle this major challenge is *resource sharing*. It consists in reducing the number of instances of one type of component and arranging the requests to use the remaining units.
Reducing the number of components may result in a longer latency if some of them were scheduled to be used at the same time. It is of the utmost importance that the delay increase does not affect the critical path, for it would cause a longer clock period and therefore an inferior clock frequency.
In order to avoid a significant increase in the path delay, it is crucial to plan the arrangement of the remaining units with the aim of reducing the impact on timing.

The purpose of this thesis is to find a clever algorithm with which to set up the groups of operations that must be allocated to each remaining unit after the implementation of the resource sharing. In this case, the components taken into account are the biggest units present in the circuit such as multipliers or multiply and accumulate units (MACs), that require a great amount of resources allocation in an FPGA.
Decreasing the number of the largest units can highlight the performance

gap, in terms of area and timing gradient, between different sharing arrangements.

The smart resource sharing algorithm taken into account in this report relies on the proximity method: the hypothesis is that a cluster of units placed closer to each other should be replaced by one shared unit, while units that are located further apart should use different resources.
This results in an even greater improvement in area reduction since this method leads to shorter connections. In fact, the wires connecting elements that used to interact with the original units have been replaced with only one element to be shared.
Using the smart sharing algorithm the new shared unit will be placed in proximity of all the elements that need to reach it.

As well as resource usage, the proximity method may be able to limit the path delay increase inevitably caused by the employment of the sharing. Indeed, the employment of shorter wire connections translates into quicker communications and shorter delays.

Lastly, three examples are reported in order to highlight the effectiveness of proximity-based resource sharing.
The first example consists in a direct form FIR filter. The Finite Impulse Response (FIR) filter's architecture is an optimum model to test the smart resource sharing since it is simple and some of the best sharing configurations are already known. The sharing is applied on a modified architecture obtained grouping each couple of Multiplier and Adder as a MAC unit.
The filter architecture is described as a loop but, in order to separate its different components, it is necessary to unroll the loop. As a result, each MAC is seen by the compiler as an independent unit and their instances can be detected in the dfg and shared.

The second example is a study of a knp_mod code. Kahn process networks (KPNs) are models of computation that describe sequential processes connected between each other by FIFO channels.
Its cpp code is full of chained loops with numerous multiplications and additions, therefore it was useful to isolate multiply and accumulate units (MACs)

and apply the sharing to these units. Implementing a proximity-based resource sharing instead of a random sharing could lead to significant improvements in terms of Area Usage of the resulting chip.
The main loop includes two kinds of MAC operations: the first one is a multiplication and addition of integer variables, while the second involves float variables. Therefore, two mac functions are created: *imac* and *fmac*.

Finally, the third example consists in an LDL, a Cholesky decomposition used to deconstruct a matrix into a product of matrices. It is an ideal study code since it describes a complex and realistic design though chained loops with numerous operations that could be optimized through sharing.
The most worthwhile units to isolate and share are multiply and subtract blocks, called *multsub*.
As well as for the previous example code, the unrolling of the main loop is necessary to see each *multsub* as a separate functional unit and, subsequently, to apply the sharing.

Chapter 2

High Level Synthesis Design

The aim of the following section is to introduce the reader to the world of High level Synthesis (HLS) and to illustrate the main characteristics of the employed HLS tool, Vivado HLS.
Furthermore, a brief presentation on FPGAs is given, since the method exposed in this work is expected to be implemented on this kind of devices.

2.1 High Level Synthesis (HLS)

An electronic system is a collection of electronic components working together in order to fulfill a common purpose. Their behaviors are interdependent from one another and therefore each component cannot be designed singularly but it is necessary to take into account the relationship and communications with the other elements.
A system usually is not made up by a single kind of components but it combines elements from different fields: it can include software as well as mechanical, electric or electronic hardware.

The design of the hardware of electronic systems has become more and more complex every year. At the beginning the circuits were small and the number of transistors was limited, therefore designers were capable of describing each transistor and their connection inside the chip.

Nowadays, the manufacturing of transistor increased exponentially and it is no longer possible to work on single transistors while designing a whole electronic system that contains billions of them.

Therefore, designers are relying on increasingly sophisticated tools that are capable of helping them in the design phase. Rather than projecting every single transistor, it is possible to specify only a general circuit and let the electronic design automation (EDA) tool create a physical layout out of the given instruction.
This allowed to work on new levels of abstraction and to design bigger circuits more efficiently.

Since the hardware complexity has continued to increase, new and more abstract hardware programming languages have become necessary. Register Transfer Level (RTL) is the first level of abstraction after the Gate Level, it consists in the describing the circuit registers and operations as blocks whose internal structure will be automatically created by an EDA tool.
The tool automatically translates the RTL instructions into specifications for another tool that is able to implement the digital circuit. These instructions can be used to create a new electronic device or can be used to program a field-programmable gate array (FPGA).

High Level Synthesis is a further level of abstraction that designers use to focus on wider architectural issues and global behaviors of the electronic system rather than managing the structure of every register and connection. At first, only hardware description languages like Verilog were supported by HLS tools able to generate detailed RTL architectures, but later also C/C++ started to be widely used.
An algorithmic HLS tool is supposed to handle numerous RTL matters automatically such as: exploiting concurrency; implementing interfaces to the rest of the system; limiting the critical path delay to the desired one through the insertion of registers; or achieving the most efficient implementation by mapping operations into the best logic configuration.

All of these decisions are made by the HLS synthesis tool automatically based on user directives and design constraints. However, there are several

HLS tools that implement these operations through distinct methods reaching different levels of effectiveness. The HLS tool that has been used in this work is Xilinx Vivado HLS.

2.2 Vivado HLS

Vivado High-Level Synthesis is a Xilinx HLS tool that starting from C, C++ or System C specifications is able to translate the HLS instructions directly into a file synthesizable on Xilinx programmable devices without having to manually describe the RTL.

2.2.1 Inputs for the synthesis

Vivado HLS requires an input code written in C, C++ or SystemC. The input function must be tested through an exhaustive test bench code that receives the results generated by the code and compares them with the correct ones. If the outputs correspond the test is passed and Vivado HLS can proceed with the synthesis.
The user must declare a target FPGA device on which the synthesis will be carried out, as well as a maximum clock period bound: the tool will arrange the logic and insert registers were necessary to stay below the given clock margin.

In order to better customize the hardware design to the user purpose, it is possible to insert some directives guiding the implementation of the circuit by Vivado. These directives are called **#pragmas** and through them, the designer can give hints to the tool on how to generate the most efficient hardware layout.
Pragmas can be added directly in the source code kernel and include multiple optimization types such as:

- **pragma HLS allocation:** this pragma indicates a limit to the number of resources allocated inside the kernel. It can be placed inside a specific loop or function to limit the Register Transfer Level (RTL) hardware components instances employed for the synthesis of the specific code section. For example, it is possible to limit the number of multiplier, adder or function instances implemented in a loop of the code.

Some common operations such as additions, multiplications, array reads and writes can be automatically detected and limited through an allocation pragma, while more complex operands must be manually enclosed in a function and later its instances can be limited.

- **pragma HLS latency:** this pragma declares a minimum and/or maximum latency value for the function, loop or region to complete its execution.

- **pragma HLS inline:** this pragma forces the tool to consider the function where it is declared as a part of the main code, from where the function is called, and no longer as a separate entity.
  Often, inlining a function leads to improvements in terms of area reduction of the circuit since it allows the sharing and optimization of operands within the function with the rest of the code. On the other hand, an inlined function cannot be shared and this can increase the required area.
  The pragma inline can also be used with the suffix "OFF" to specify that the function must NOT be inlined into any calling function. It is mandatory to insert this command if the function must be considered a separate entity in order to share it in the RTL implementation, otherwise, it could be automatically inlined by the tool.

- **pragma HLS pipeline:** this pragma sets the initiation interval (II) of a function or loop to an inferior value than the total number of clock cycles needed to complete its execution. This allows the concurrent execution of operations and therefore pipelining.
  A pipelined function or loop can start a new execution every N clock cycles, where N is the II of the loop or function set by the user. The default II of the pragma is 1, which means that a new process is started every clock cycle.

- **pragma HLS unroll:** this pragma unrolls loops in order to generate multiple and separated statements from the original assortment of operations. The unroll pragma remodels loops by reproducing multiples copies of the original loop body, this allows to manage the execution of the operation more freely.
  It is possible to force the loop body operations to be executed in parallel or implement them as independent components in the register transfer level (RTL) design. In fact, rolled loops are synthesized with logic for

only one iteration and it is executed for each repetition of the loop.
The pragma allows the loop to be fully or partially unrolled, in this case, the user must specify an unrolling factor N, to create N copies of the loop body.

2.2.2 Vivado HLS limits

As seen previously Vivado HLS provides tools to perform a closer control on the RTL and enhance the expressiveness of the language employed through pragmas. Nevertheless, it limits the designer possibilities in some other aspects since the input code cannot support some features, such as:

- Dynamic memory allocation (operations like malloc(), free(), new)(), and delete());
- Recursive function calls;
- System calls (like abort(), exit(), printf()) are ignored by the synthesis;
- Limited use of some libraries and pointers;
- All interfaces must always be accurately defined;

2.2.3 Vivado HLS outputs

Vivado HLS is a powerful tool capable of exploiting the synthesis of an HLS design into an RTL layout and subsequently the place & route.
The Place and Route (PAR) is a complex process that consists of the determination of each hardware component location on the board and the relative connections between them. The PAR must be carried out on a specific target device on which the resources will be mapped.

The main output that Vivado HLS, or any other HLS tool, provides is an RTL hardware design obtained after the compilation of the input function and its simulation through an exhaustive testbench. It is a VHDL or Verilog synthesizable code.
Furthermore, the tool automatically generates some RTL simulations based on the testbench.

Starting from the RTL design file the synthesis can be carried out on a target FPGA through a command where the desired clock period is specified.

The Verilog/VHDL design is translated into a netlist of the FPGA logical elements and the connections between them.
Lastly, Vivado HLS provides some assessment on path delays and resource usage. These files are obtained from the synthesis before the place and route process, therefore they are only estimates.

Lastly, the netlist can be used by the tool to perform the Place & Route of the design on the desired FPGA device. The PAR determines the resource used on the FPGA, their exact location and their connections. The information on the FPGA configuration is enclosed in a bit-stream that can be loaded on a physical FPGA of the target type to program it as intended in the original HLS design.

2.3 FPGA

A field-programmable gate array (FPGA) is an integrated circuit characterized by the possibility to be configured by the user after the manufacturing. It is an incredibly versatile tool for engineers that have been using it as a first step in the design of specialized circuits before the hard-wired production for large-scale distribution.
The FPGA can be customized to the user needs loading a bit-stream containing the design configuration, written in a hardware description language such as VHDL or Verilog.
Afterwards, it can be reprogrammed to implement different logic design allowing to work on hardware with the same flexibility of software computing.

FPGAs are composed of an array of programmable logic blocks and memory elements connected between each other through programmable interconnections that allow different wirings configurations. The simple logic blocks can be assembled to implement logic gates or more complex combinatorial functions.
Most FPGAs also include memory elements: simple flip-flops or more complete blocks of memory such as block RAMs (BRAMs).

The flip-flops (FF) are the basic memory elements in the FPGA and they

are typically implemented as a lookup table (LUT), a memory where the inputs are the address signals and the outputs are stored in the memory entries.
A more complex and powerful programmable logic element, called a configurable logic block (CLB) or Slice, can be obtained assembling FFs, LUTs, multiplexers and possibly other specialized functions.
A slice may also incorporate a complex unit like a full adder, this is an operation of "hardening" of the FPGA since that resource cannot be used as programmable logic but only as a full adder, obtaining better performances at the expense of flexibility.

A programmable interconnect network is used to link the different Slices. Their ports are connected to a routing channel that holds a set of configuration bits that can be switched to connect or disconnect the slice to the interconnections. Routing channels are linked to switchboxes which are a set of switches enabling the connection paths to change routing channel.

As more sophisticated FPGAs started to appear, they have begun to incorporate more complex units as "hard" resources. An example is the DSP48, a complex and efficient component specialized in arithmetic operations like multiplication, addition and multiply and accumulate, that implements bigger operations in a much more efficient way than programmable logic.
Another hardened resource commonly used in FPGAs is the block RAM (BRAM). BRAMs are configurable random access memory units that can be customized by the user since they support different memory layouts and interfaces.
These complex blocks highlight the problem of hardening in FPGAs. Even though they are programmable for some aspects, they are not nearly as flexible as typical FPGA logic blocks. On the other hand, "hard" blocks can lead to great improvement in the FPGA efficiency and performance at the expense of flexibility.

Chapter 3

Smart Resource Sharing

The aim of this chapter is to present the proximity-based resource sharing employed in this study.
Firstly, a brief introduction about the benefits introduced by a classic resource sharing is inserted.
Moreover, the theory that leads to the smart resource sharing based on proximity hypothesis are outlined.
Lastly, all the necessary steps to implement the proximity-based resource sharing algorithm are described in detail in the following sections.

3.1 Advantages of Resource Sharing

Resource Sharing is a common technique used by designers to reduce the number of resources used in a circuit and consequently its area.
It consists in replacing 2 or more identical components with just one of them. The consequently "shared" item must be used by the compiler for the same number and kind of operations as it was doing with the previous configuration. If two or more operations had to be necessarily performed at the same time, the sharing of only one resource can increase the latency and therefore the clock period.

Nevertheless, not always resource sharing affects the clock frequency: this only happens if it impacts the critical path delay making it longer than it was with the previous hardware configuration. Since the clock period must be longer than the critical path delay this will force a lower working frequency.

Consequently, not all of the resource sharing configurations are equally effective. The best sharing combinations are the ones that lead to the smallest area occupation on the FPGA, this includes the smallest number of multiplexers, shortest routing and, of course, maximum reduction of allocated resources.

The goal of the smart resource sharing exposed in this thesis is to find the best sharing solution in order to exploit its main advantage in terms of area savings and at the same time reduce the newly generated paths.
The idea that allows the smart resource sharing to do so is the awareness of the importance of proximity.
Replacing closest elements instead of random or furthest one with only one of them results in shorter connections. Shorter wiring is not only fundamental to obtain a smaller area demand but also to achieve quicker communication delays that influence the timing of the circuit.

3.2 Proximity-based Resource Sharing

The Smart Resource Sharing Algorithm presented in this thesis aims to reduce the area consumption of a circuit and coherently the critical path delay through the sharing of resources exploiting *proximity*.

3.2.1 Definition of proximity and expected advantages

Proximity indicates the spacial closeness of elements in a planar surface that can be either the virtual space of the DFG graph or the physical board where the electronic components are installed.
The position of the instances of a selected type of component is the data under study, from which the distance between them can be computed. A group of components of the same typology that are placed closer to each other in the planar space, compared to the rest of them, is called a cluster.

The optimal proximity clusters are found by an iterative algorithm called K-Means that generates groups of items based on their closeness. The process consists in identifying randomly a target number of centroids and redefining iteratively their position until the algorithm converges. Afterwards, K-Means associates each element to the closest centroid through the calculation of the mean.

The solution is found through a heuristic algorithm, therefore it is not guaranteed to be the optimal one.

The developed smart resource sharing aims to find the optimum sharing configuration that leads to the smallest FPGA area occupation. This is expected to be achieved through the study of the proximity of the components: closest elements should share the same unit, while furthest elements should use separate ones.
Applying the proximity resource sharing to an electronic design is anticipated to result in the shortest possible routing and the smallest number of employed multiplexers. These factors have a strong impact on FPGA resources allocation, area usage and path delays.

Whereas, forcing a "bad" resource sharing configuration, such as letting one unit being shared by components that are placed far apart in the chip, is expected to cause higher muxing and longer wiring. This happens because further areas of the circuit must be connected through longer routings that take up a significant amount of additional area. In addition to larger FPGA resources allocation and area usage, these factors cause longer path delays, due to longer wires, that may result in a lower clock frequency.

In order to obtain the maximum effectiveness of the smart resource sharing employed, it is necessary to detect which are the biggest and most time-consuming units in a given code, such as Digital Signal Processors (DSPs) or Multiplier-Accumulators (MACs). The sharing of big components and their consequent number reduction leads to a greater impact on the reduction of resource usage and, accordingly, area occupation.
The chosen elements must be isolated from the rest of the code so that they can be used as physical components whose number must be reduced.

The Proximity-based Resource Sharing consists in the assumption that while applying resource sharing on a kind of component: a cluster of units placed close to each other should be replaced by a shared one, while units that are located further apart should use different resources.

3.2.2 Smart sharing effectiveness validation process

The process followed in order to prove the effectiveness of the proximity-based resource sharing consists of multiple steps described in the sections below.

Detect units to which apply resource sharing

Given a realistic code to be synthesized, the most demanding operations in terms of area and energy consumption must be detected.
These units are often the ones that require a bigger number of DSPs allocated in the FPGA layout and those that are most likely to affect the maximum path delay. An example of a big operation unit is a Multiply and Accumulate (MAC) which requires a large amount of area and time especially due to the presence of a multiplier.

If the DSPs units are in a loop, it can be necessary to manually unroll the loop in order to obtain visibly separated instances of the units under study.

Prevent the inlining of the chosen function

Each operational unit must be enclosed in an external function that is called from the core of the code. Every function must contain a #pragma that prevents its "*inlining*", that is to say inhibiting the compiler from inserting the function code at the address of each function call.

The *inlining* is usually useful in order to save the overhead of a function call, but this will prevent the distinction of the different units that must be shared. Therefore, this process allows to highlight the presence of the big units individually inside the circuit and, more importantly, in the Data Flow Graph (DFG) generated after the synthesis. The DFG represents graphically all the operations that must be synthesized in the chip, connecting each data object to the right operating unit displaying sequential or concurrent executions.

Synthesis of the code before resource sharing

In order to apply a smart resource sharing, it is necessary to know the position in which the big units under study would be placed after the synthesis of the original code.

Therefore, the code, where the units have been isolated as described in the previous step, is imported in the hls tool Vivado HLS. The tool is able to carry out the synthesis on a chosen FPGA through a previously written script that defines the FPGA and some basic parameters, such as a target clock period.
After the synthesis, the DFG is generated and saved in a DOT file.

Visual rendering of the DFG through Graphviz

The DFG is described in a DOT file generated by Vivado HLS after the synthesis of the circuit.
The DOT language is a graph description language that defines a graph but it is not sufficient to visually render it, therefore it is necessary to use a package called Graphviz (Graph Visualization Software) that provides a set of tools to process a DOT file.
Some of these tools have been used to visually render the DOT file, each of them has different characteristics that affect the resulting graphical representation:

- **dot**: a tool that produces layered drawings of directed graphs. It places the nodes in layers sorted in a hierarchical way.
- **sfdp**: a tool that visually renders undirected graphs into a non-layered structure.
- **circo**: a tool that, from the original DOT file, produces drawings with a circular graph layout.
- **patchwork**: a tool that visually renders the original DOT file depicting a graph as an implementation of squarified tree-maps.

Find the optimum proximity solution through clustering

After the graphical rendering of the Data Flow Graph the nodes indicating the chosen operation units must be detected in the graph.
A Python code has been created to extract their coordinates from the DOT file opened through one of the Graphviz processing tools.
A chosen number of clusters is generated using K-Means, an algorithm based on proximity. The clusters may be composed of an uneven number of elements, for this reason, the EqualGroupsKMeans library is adopted to obtain balanced clusters, when possible.

According to the proximity theory, the function calls related to elements belonging to the same cluster are the ones that must share the same functional unit.
In Figure 3.1 it is shown an example of the K-Means algorithm employed on 6 items to be divided into 3 groups. The elements belonging to the same cluster are identified as dots of the same light colour, while the darker spots indicate the location of the centroids. The centroid is the point that may be considered the center of the plane figure created by the cluster's units.

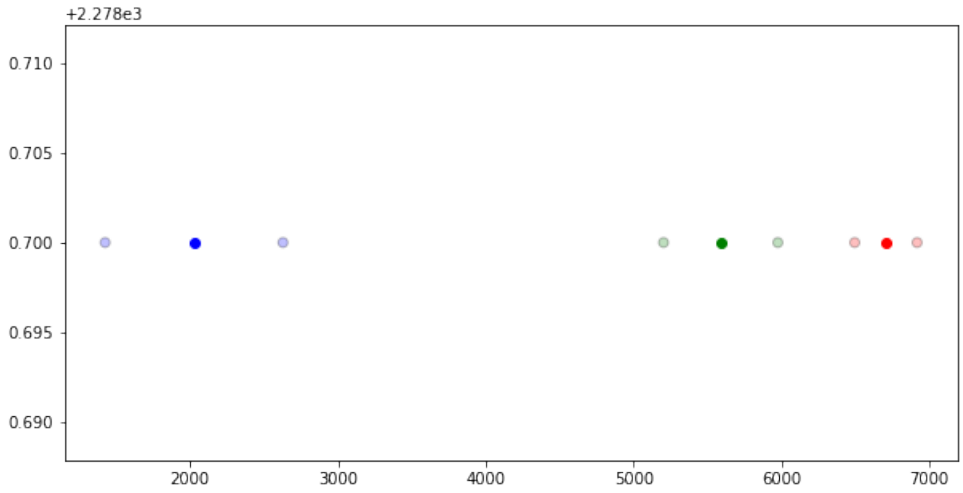


Figure 3.1. Ex: clustering algorithm application - n=6, k=3

Generation of new code files

Once the optimum locality solution is found through the clustering algorithm, a new .cpp file can be generated on the basis of the original one.
For some examples such as the FIR filter, it is necessary to modify also the header file since the structure and the units are described there. This is carried out employing a similar process to the one used for the cpp file.

A python code has been written to open the original code file and to read it in order to detect the name of the units to be shared (MAC). The name of each MAC instance is replaced with a new name representing the cluster it has been mapped in. For example, the function "mac1" belonging to the second cluster is changed into "mac_new2".

Subsequently, in order to simulate the sharing of one unit for a chosen group of MACs, the number of instances of the new functions must be limited to 1. This restriction is applied to each new function through a #pragma command: the compiler can only have 1 instance for each cluster function as if it was a physical resource being used from different lines of the code.

Once the optimum solution according to the locality method has been generated, it is useful to compute some random cluster solutions in order to assess its goodness.
The easiest way to do so is shuffling the vector "labels": it is an array whose positions correspond to the original units and the content of each cell to the clusters to whom that unit belongs.
Afterwards, the .cpp or the header file can be printed as explained before.

Synthesis and Place & Route

The next step is to import the new codes in Vivado and to carry out the synthesis and place & route. This process requires a script for each solution to be synthesized and some hours to exploit the placement and routing of every project.
At the end of all of the simulations, Vivado returns different files from which it is possible to get information on the key features of the resulting circuit such as:

- The resource usage required by each process and therefore the chip area occupation.
  This information can be extracted from the *"_utilization_routed.rpt"* report file that contains the Utilization Design Information. The report is obtained after the place and rout and shows how many items are needed in order to synthesize a project. They are classified into 10 categories:
  - Slice Logic
  - Slice Logic Distribution
  - Memory
  - DSP
  - IO and GT Specific
  - Clocking

  - Specific Feature
  - Primitives
  - Black Boxes
  - Instantiated Netlists

  These quantities are also expressed as a percentage of the used resources over the available ones.
  The value used to assess the goodness of a solution is the maximum percentage of utilization of one kind of resource. This quantity is fundamental in this study because the research is based on a resource sharing method for FPGA application mainly. Therefore, since the FPGAs contain a finite number of resources, this becomes a decisive parameter.

- The minimum and maximum latency, measured in number of clock cycles. This information is included in the *"_csynth"* file, a report file generated by Vivado after the synthesis.

- The maximum path delay, that corresponds to the critical path delay of the circuit. This parameter is included in the *"_timing_paths_routed"* file, a report file generated by Vivado after the place and route where the delays of the 10 longest paths are illustrated.
  This value puts an upper bound to the frequency of the designed circuit.

Computation of the Clustering Cost

Another parameter that can be used to assess the goodness of a solution is the *Clustering Cost*: the sum of distances of each unit to its centroid.

The Clustering cost is computed through a Python code from the coordinates of the graph drawing for each balanced combination.

Results comparison and behaviour analysis

Another Python code has been created to compare the computed values and to graphically plot the results.

All the report files *"_timing_paths_routed"* from all the different solutions are read and the path timings are extracted. They include 10 paths each, for every solution the maximum delay and the root mean square of the delays is computed and printed in the output file *"_random_delay.txt"*.

The first one is the optimal solution found through the clustering method and it is analyzed individually, while the following 10 values are related to the 10 random combinations.

Afterwards, the post-implementation resource usage is evaluated from the *"_utilization_routed.rpt"* file.
The report files from the different solutions are read to extract the maximum percentage of FPGA resources, such as Slices, DSPs, Flip Flops, BRAMs or Shift Register, employed for the system place and route.

The graphs that have been plotted are of different kinds:

- **Delay-Area diagram**: In the y-axis, the total execution time of a solution is plotted as the multiplication of the maximum path delay and the numbers of clock cycles. There could be two versions of this graph since Vivado provides a minimum and a maximum latency value, therefore it is possible to use both and see if this leads to some changes in the diagram.
  In the x-axis is plotted the Resource Usage maximum percentage, this parameter gives an impression of how much the chip area increases or decreases changing the sharing arrangements.

  The resulting graph includes several points, one for each resource sharing arrangement, marked with the name of the respective solution or with a number if related to a random solution.

- **Delay-Clustering Cost diagram**: The total execution time of each solution is plotted in the y-axis as the multiplication of the maximum path delay and the numbers of clock cycles.
  In the x-axis is plotted the Clustering Cost, the sum of distances of each unit to its centroid. This parameter quantifies the goodness of the clustering solution found: the closer the items are to their clustering centroid the smallest is the clustering cost.
  The resulting graph is composed of several dots, one for each resource sharing arrangement, marked with their solution name or with a number if they are related to a random solution.

  From these data, it is possible to extract the linear regression in order to model the evolution of the results as a line that shows an increasing

or decreasing trend along with the increasing of the clustering cost.

- **Area-Clustering Cost diagram**: The Resource Usage maximum percentage of each solution is plotted in the y-axis. This parameter indicates how much the chip area increases or decreases changing the sharing arrangements.
  In the x-axis, the Clustering Cost is plotted. It is the sum of distances of each unit to its centroid.
  The resulting graph is composed of one dot for each resource sharing arrangement, and they are marked with the respective solution name or with a number if related to a random solution.

  The linear regression can be extracted from the obtained dots. It is useful in order to model the evolution of the results as a line that can show an increasing or decreasing trend along with the increasing of the clustering cost.

From these graphs, it is possible to draw some conclusions on the smart resource sharing theory that was assumed from the outset. The smart sharing based on proximity. The assumption is that, While applying resource sharing, replacing closer units instead that further units with a single shared resource may results in a shorter path delay, due to shorter interconnections, and smaller resource usage.

K-Means Clustering Algorithm

The clustering algorithm that has been employed in order to decide which items would be replaced by the same shared resource is the K-Means clustering [1].
A cluster is a group of items that share a certain characteristic: in this case proximity.
The aim of K-Means is to partition an original collection of units spatially placed in 1, 2 or more dimensions into a given number of clusters.
Each cluster is built around a centroid, a point in space that represents the center of the cluster. The used algorithm starts from these points to build the cluster through an iterative technique.

Starting from the whole collection of points in space, the K-Means algorithm identifies randomly a target number of centroids. Subsequently applying an

iterative computation, the position of the centroids is redefined until the algorithm converges: the position of the centroids remains the same after the iteration. The solution is found through a heuristic algorithm, therefore it is not guaranteed to be the optimal one.
Afterwards, K-Means creates clusters associating each element to the closest centroid through the calculation of the *mean* and the least squared Euclidean distance.

Clusters may, therefore, be unbalanced since the number of elements belonging to each of them is decided individually by locality. Nevertheless, for some applications, it might be necessary to work with clusters that are made up of the same number of units. Therefore, it is possible to force the assembling of balanced clusters through another Python library: *EqualGroupsKMeans*

Equal Groups K-Means Clustering is a variation of the K-Means algorithm that generates balanced clusters: groups of items of the same size.
It uses the same logic employed by K-Means but computes the maximum number of items per cluster at the beginning and distributes the elements according to this bound.

Chapter 4

Simulations and results

In this chapter three examples are reported in order to test and to prove the effectiveness of the proximity-based resource sharing.

The first example code describes a direct form FIR filter. The smart sharing is applied on a modified architecture of the filter obtained grouping each couple of multiplier and adder as a MAC unit.
Since the filter architecture is described as a loop, an unrolling procedure is mandatory to work on the single fir cells.

The second example is a study of a Kahn process network code, full of chained loops with numerous multiplications and additions. Combining them into MAC units and applying the smart sharing on these components leads to significant improvements in terms of area saving.
The main loop includes two kinds of MAC operations, one using integer variables and one float variables: *imac* and *fmac*.

The third example is an LDL code, a Cholesky decomposition used to deconstruct a matrix. It is a complex and realistic design with chained loops with numerous operations. The smart sharing is implemented on multiply and subtract blocks, called *multsub*. As well as for the FIR code, the unrolling of the main loop is necessary to access each *multsub* unit.

4.1 FIR filter

Finite Impulse Response (FIR) filters are a widely known digital signal processing (DSP) application. It is a filter characterized by a finite duration

impulse response.
It is an optimum model for hardware implementation as well as academic applications since it has a highly optimized and simple architecture.

The FIR filter has been picked for this study because the best sharing

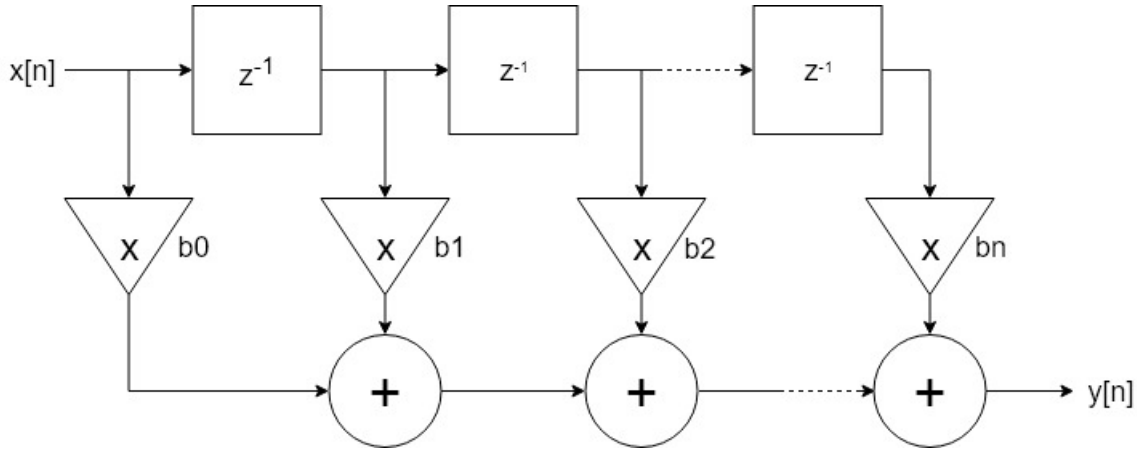


Figure 4.1. Direct form FIR filter of order n

configurations for its direct form are already known. This allows to demonstrate that the congestion oriented resource sharing algorithm leads indeed to area saving and delay reduction.

Modifying its architecture grouping each couple of Multiplier and Adder as a whole MAC unit, the well known best sharing combinations for a FIR filter in direct form are:

- **In order optimal solution** Dividing the ordered group of MACs in k groups simply isolating the first cluster (MAC1, MAC2, ..., MACn/k) and so on collecting k items per group until the last one (MAC(n/k)+1, MAC(n/k)+2, ..., MACn). Each set of components must share one MAC unit.

  This sharing solution is the simplest and yet most effective one. Forcing ordered groups of cells to share the same components, the muxing and routing are minimized due to the closeness of the cells between each other and, therefore, the proximity of the elements that communicate with them.

- **Alternate optimal solution** Dividing the ordered group of MACs in k groups picking the elements alternately from the ordered architecture of the filter. For example, in the case of k=2, the clusters will be [MAC1, MAC3, ..., MACn-1] and [MAC2, MAC4, ..., MACn]. Each set of components must share only one MAC unit.

  This second sharing solution is another configuration that guarantees an optimal trade-off between area saving and path delay reduction. Distributing the cells alternately in the sharing groups that are to be replaced by the same component, minimizes delays while reducing area requirement. This process allows to use different components from different parts of the circuit in consecutive clock cycles and, therefore, minimizes the waiting time necessary to use the same resource again.

A simple DOT representation of a direct form discrete-time FIR filter of order 8 has been created following the architecture shown in Figure 4.1. Its structure includes 8 Multipliers and 7 Adders, for a total of 8 cells.

The Graphviz tool *dot* has been used to open the file and generate the multi-layered structure depicted in Figure 4.2. Since *dot* produces layered drawings form directed graphs, the nodes are placed in layers sorted in a hierarchical way.
Consequently, the Adders are represented as nodes placed on a diagonal, while the Multipliers are distributed in parallel to them. It appears clear that the layers of this structure have been generated following the stages of the scheduled operation Data Flow Diagram, as shown in Figure 4.2.

It was worthwhile applying the K-Means clustering algorithm for k=4 to the Multipliers, for they are the biggest and most time-demanding components of the FIR filter architecture.
The obtained clusters are not balanced [0-1-2, 3-4, 5-6, 7] as portrayed in Figure 4.3.

Therefore, it was necessary to apply the EqualGroupsKMeans algorithm that resulted in a non-expected cluster combination [0-1, 2-4, 3-5, 6-7], Figure 4.4.

These solutions do not match the anticipated optimal sharing combination for a direct form FIR filter, hence the FIR architecture needs to be

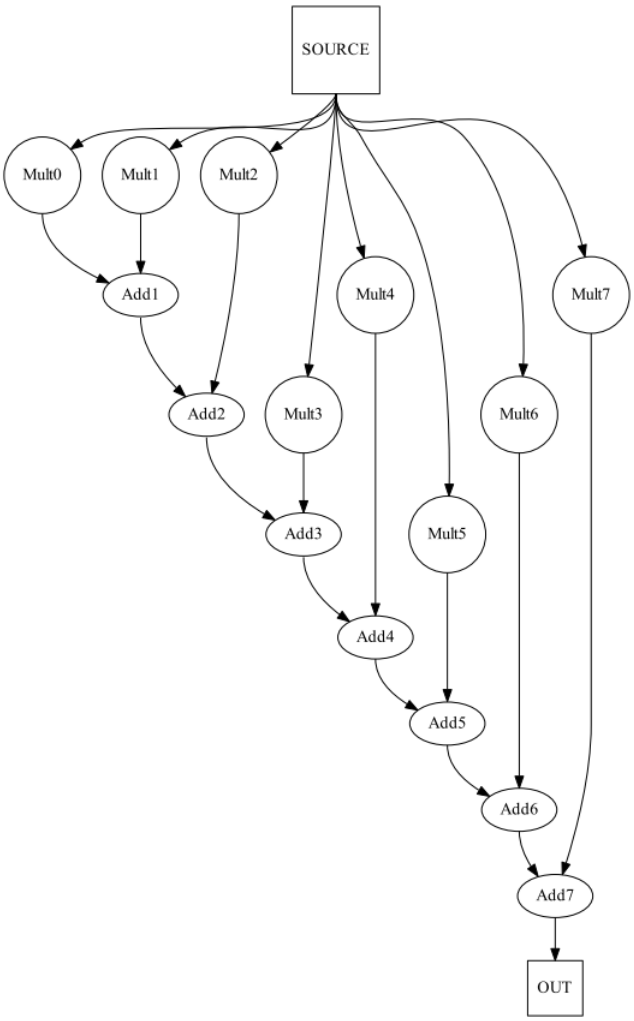


Figure 4.2. 8th order FIR visually rendered through *dot*

modified as follows.

A second DOT representation of a direct form discrete-time FIR filter of order 8 has been created following the architecture shown in Figure 4.5. This time its structure is depicted grouping each couple of Multiplier and Adder as a whole MAC unit, for a total of 8 MACs. The first Multiplier is replaced with a MAC unit as well, even though it is not coupled with an Adder. The area overhead is indeed minimal, for the number of MACs will be dramatically reduced through the Smart Resource Sharing.

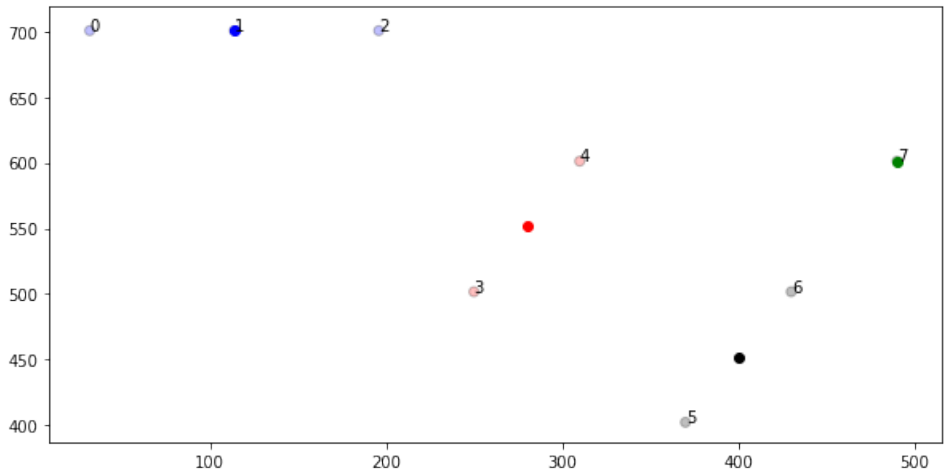


Figure 4.3. FIR filter Multipliers unbalanced K Means Clustering

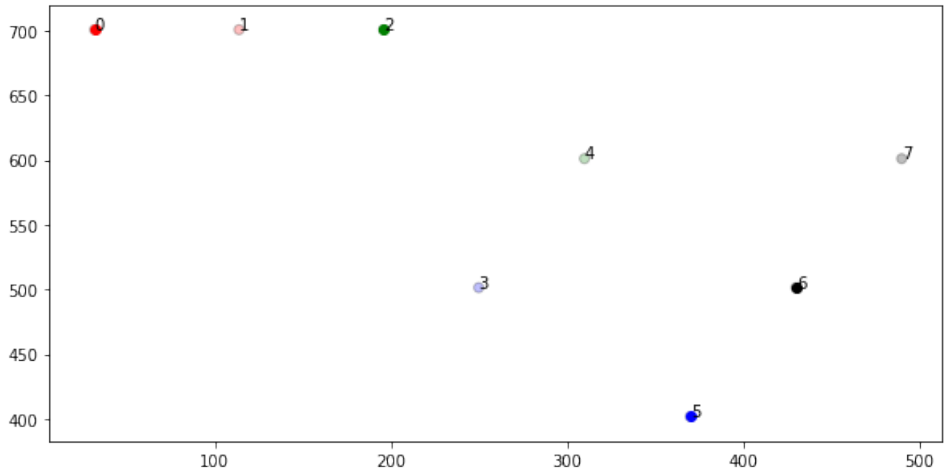


Figure 4.4. FIR filter Multipliers balanced EqualGroupsKMeans Clustering

The *dot* Graphviz tool has been used to open the file and the generated image, Figure 4.6, shows a graph whose nodes are disposed in a hierarchical structure. Since *dot* produces layered drawings form directed graphs, MACs units are represented as nodes placed on a diagonal.

Applying the K-Means clustering algorithm for k=4, the clusters obtained are [0-1, 2-3, 4-5, 6-7], as displayed in Figure 4.7. This configuration corresponds to one of the optimal sharing solutions taken into account at the

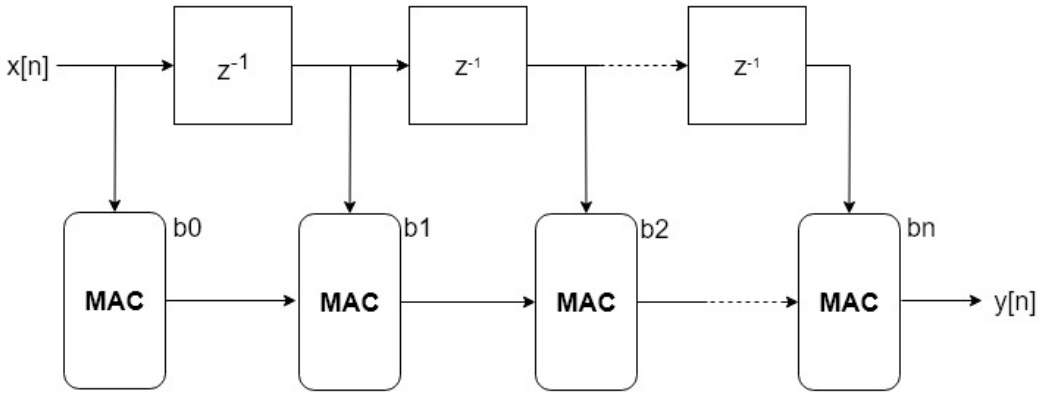


Figure 4.5. Direct form FIR filter whose operators are replaced with MACs

beginning of this chapter.

Some other Graphviz tools have been employed to highlight the difference of each rendering method.

From Figure 4.8 it is possible to appreciate how the same DFG is rendered using the *sfdp* Graphviz tool. The generated structure does not resemble a tree, since *sfdp* renders undirected graphs into a non layered structure.

In Figure 4.9 it is pictured the DFG of the 8th order FIR composed by MACs rendered through *circo* Graphviz tool. The generated structure resembles a circle, since *circo* produces drawings in a circular graph layout.

In Figure 4.10 it is possible to appreciate how the same DFG is rendered using the *patchwork* Graphviz tool. The tool generates a squared structure where each component is a block inside of it, in fact *patchwork* renders graphs into an implementation of squarified tree-maps.

4.1.1 FIR filter cpp code

In order to carry out the simulation on a FIR architecture, a cpp code of a simple FIR filter has been taken from the Vivado examples, A.1 A.2.

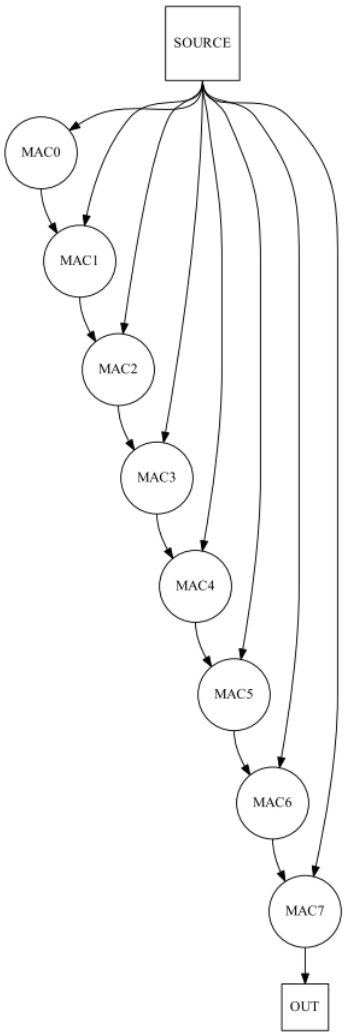


Figure 4.6. 8th order FIR with only MAC units rendered through *dot*

The code includes a header file where the FIR filter architecture is described as a loop limited by N-1, the order of the filter and number of cells. Each couple of operators described in the filter architecture, Multipliers and Adders, must be replaced with a MAC unit, previously defined as an external function.

In order to separate the different components of the filter, it is necessary to unroll the loop. In this example, the order of the filter is set to 84 (N=85), while the order of unrolling is 6. These numbers have been chosen because

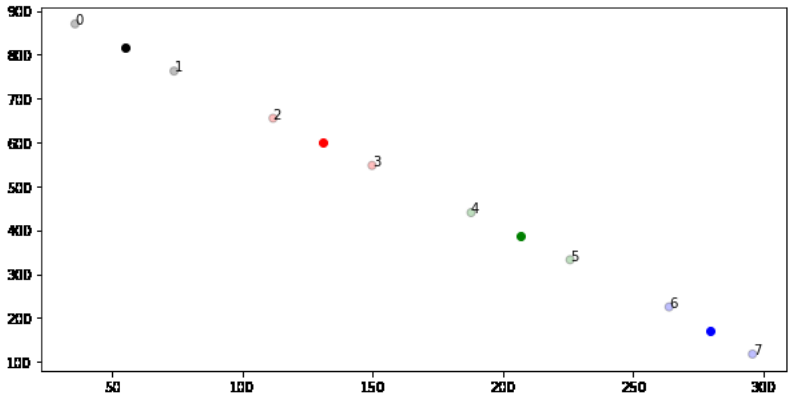


Figure 4.7. FIR filter MAC units K Means Clustering

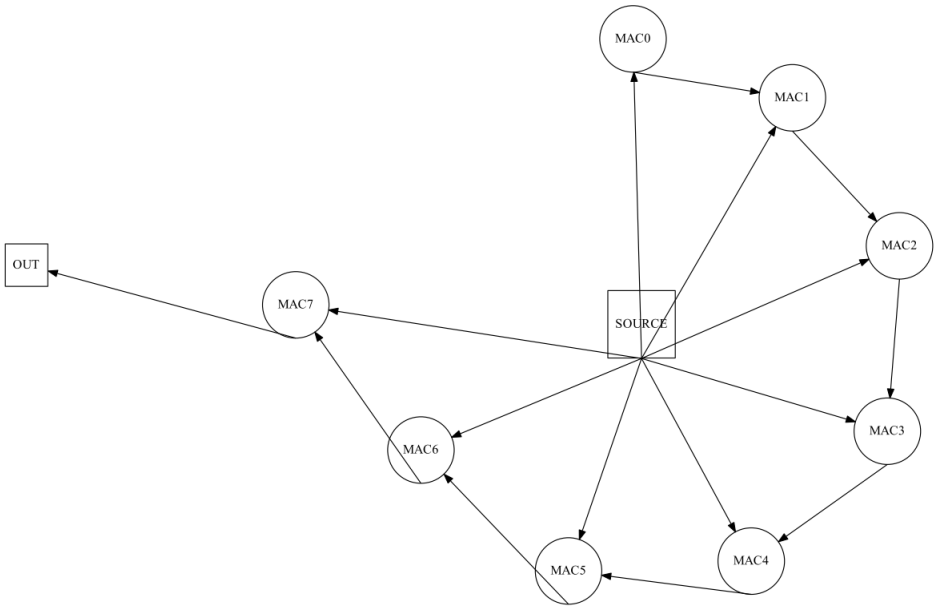


Figure 4.8. 8th MAC FIR visually rendered through *sfdp*

6 is a divisor of 84 so that the whole filter can be described in exactly 14 executions of the loop.

The 6 MAC units, now explicitly described in the header file, must be enclosed in different external functions called mac0, mac1, mac2, mac3, mac4

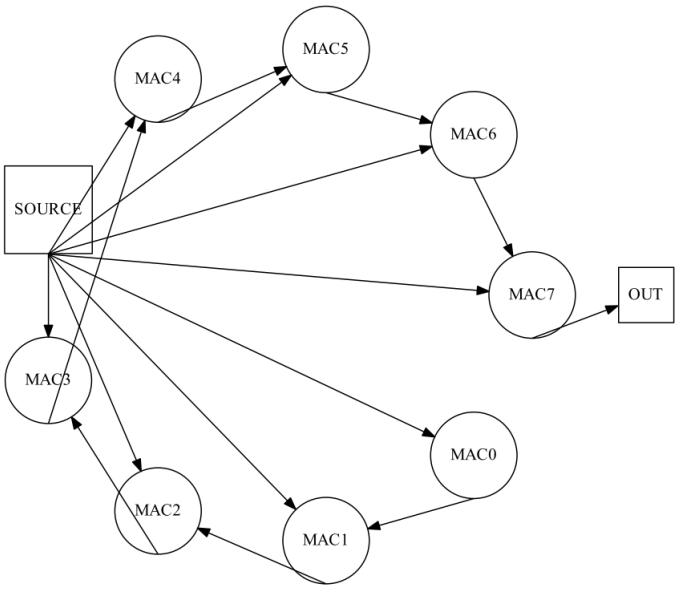


Figure 4.9. 8th MAC FIR visually rendered through *circo*

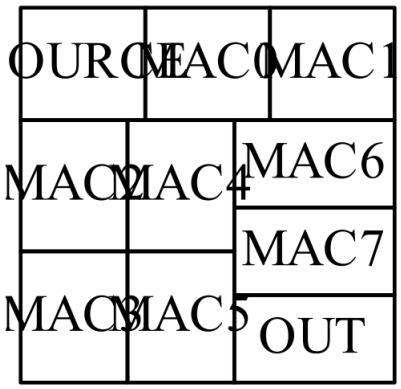


Figure 4.10. 8th MAC FIR visually rendered through *patchwork*

and mac5. Each function must contain a #pragma that prevents its "*inlining*", that is to say inhibiting the compiler from inserting its code at the address of each function call.
As a result, each MAC is seen as a separated unit by the compiler and their instances are portrayed in the Data Flow Graph generated after the synthesis.

4.1.2 Synthesis without resource sharing

In order to apply a smart resource sharing, it is necessary to know the position in which the MAC units would be placed after the synthesis of the unmodified code.
Therefore, the unrolled code containing the isolated functions is imported in Vivado and synthesized on the *kintex7* FPGA *xc7k160tfbg484-1*. This FPGA has been chosen because it is the smallest one on which this circuit can be synthesized: it is composed of 600 DSP Slices and 650 18k RAM blocks.
Working on a small FPGA is useful because it is very likely that a hardware alteration, like different resource sharing configurations, may result in a visible effect on its performances. This allows to obtain a more relevant feedback to each change done in the resource sharing configuration.

In Vivado it is possible to carry out the synthesis on the chosen FPGA through a previously written script. After the synthesis, the DFG is generated and saved in a DOT file.

4.1.3 Rendering of the DFG and K-Means Clustering

The Data Flow Graph is described in a DOT file that can be visually rendered by 4 different Graphviz tools: *dot, sfdp, circo* and *patchwork*. Afterwards, through a Python code, the nodes indicating the MAC units must be detected in the graph and their coordinates extracted.

Using the K-Means Clustering Algorithm 3 clusters have been created from the 6 original MACs of the FIR architecture.
In order to obtain balanced clusters, composed of the same number of elements, it is possible to use Equal Groups KMeans Algorithm with k=3.

This means that the filter area will be reduced by 3 MAC units and each couple of the former operations will have to share a single MAC.
From the following images is possible to appreciate how the two clustering algorithm work. The components are shown in a diagram depicting a 2-dimensional plane and are placed in the same position they occupy in the DFG. The items belonging to the same group are represented as dots of the same light colour, while the centroid of each cluster with the respective colour in a darker shade.

- **dot** In Figure 4.11 is shown the clustering generation on the MAC units

of the FIR filter DFG rendered through *dot*. The total number of elements is 6 and they are divided in 3 groups. This is achieved employing the K-Means clustering algorithm with k=3 that does not guarantee clusters composed by the same number of elements.

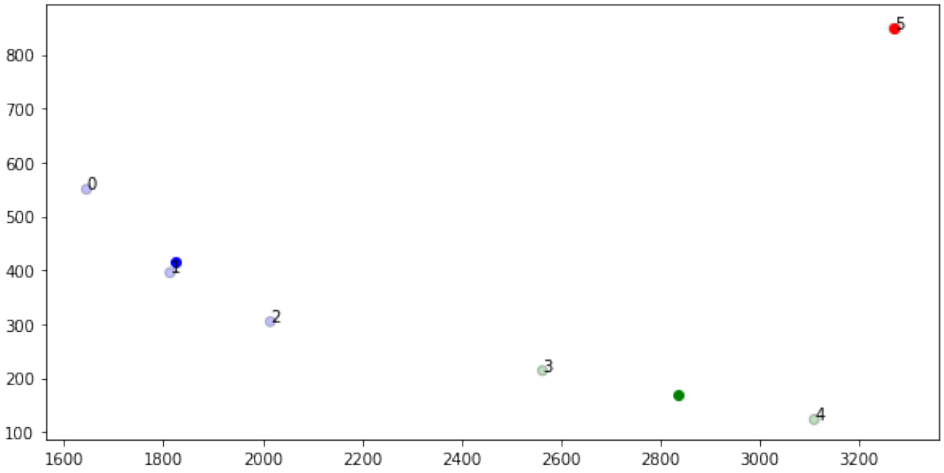


Figure 4.11. FIR fMACs - *dot* - n=6, k=3

On the other hand, Figure 4.12 shows the MAC units taken from the *dot* FIR filter architecture. The total number of elements is 6 and they are grouped in 3 couples employing the EqualGroupsKMeans clustering algorithm with k=3. This algorithm guarantees clusters composed by the same number of elements.

- **sfdp** In Figure 4.13 are shown the MAC units taken from the *sfdp* FIR filter architecture. The total number of elements is 6 and they are divided in 3 groups. This is achieved employing the K-Means clustering algorithm with k=3 that does not guarantee clusters composed by the same number of elements.

  Whilest, in Figure 4.14 the MAC units taken from the *sfdp* FIR filter architecture are shown. 6 elements are grouped in 3 couples employing the EqualGroupsKMeans clustering algorithm with k=3. This algorithm guarantees clusters composed by the same number of elements.

- **circo** Figure 4.15 depicts the MAC units taken from the *circo* FIR filter architecture. The 6 units are grouped in 3 couples employing the simple

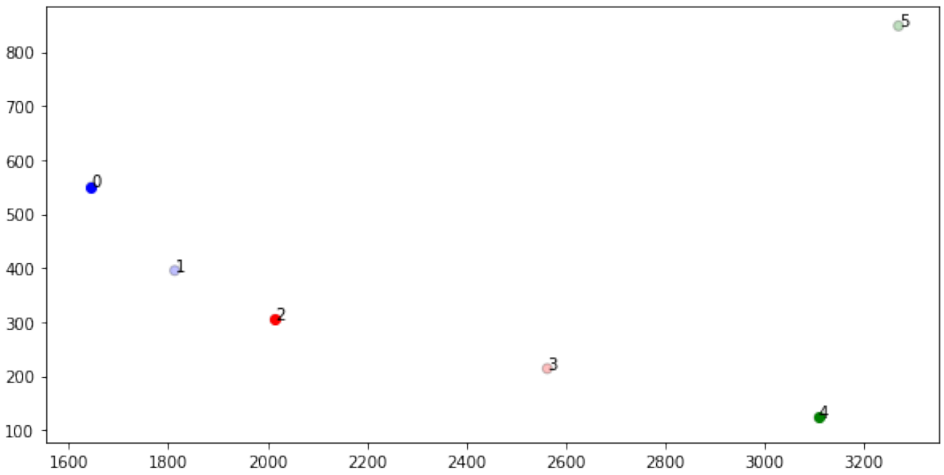


Figure 4.12. FIR fMACs - *dot* **balanced** - n=6, k=3

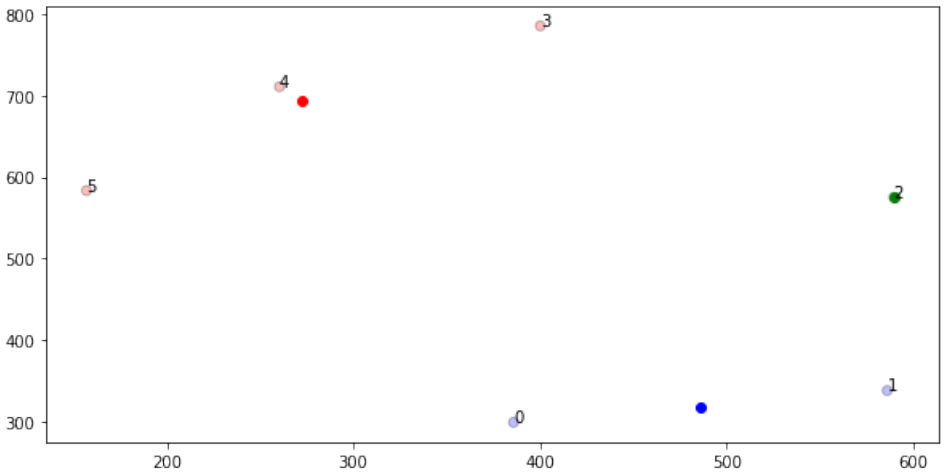


Figure 4.13. FIR fMACs - *sfdp* - n=6, k=3

K-Means clustering algorithm (k=3). This time it was not necessary to use the EqualGroupsKMeans algorithm to obtain clusters composed by the same number of elements.

- **patchwork** Figure 4.16 highlights the MAC units taken from the *patchwork* FIR filter architecture. The 6 MAC units are divided in 3 groups through the K-Means clustering algorithm (k=3) that does not ensure balanced clusters.

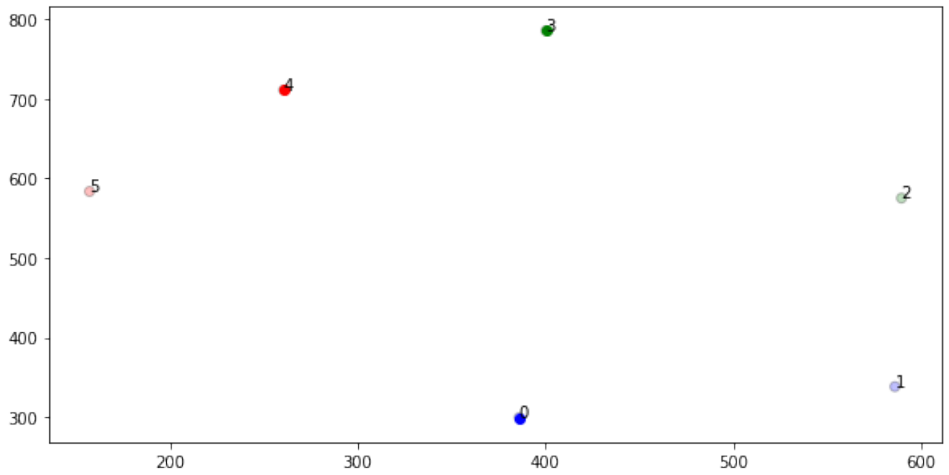


Figure 4.14. FIR fMACs - *sfdp* **balanced** - n=6, k=3

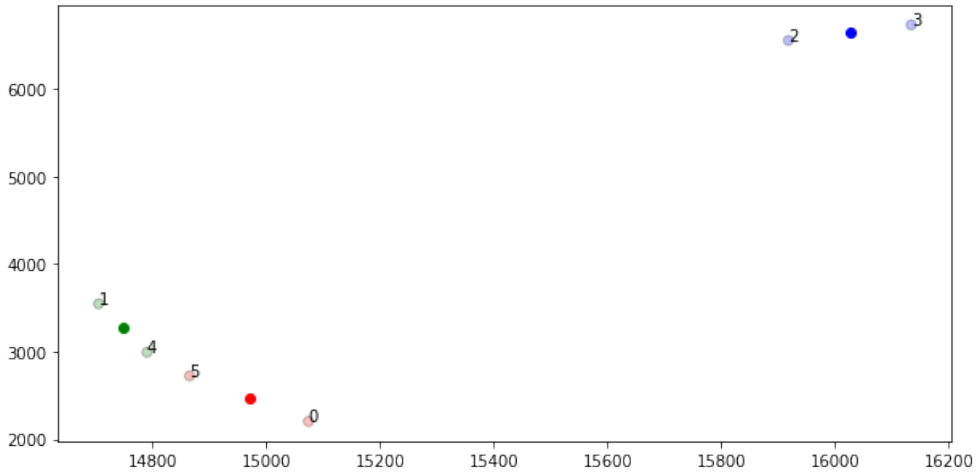


Figure 4.15. FIR fMACs - *circo* - n=6, k=3

Whereas, in Figure 4.17 the MAC units taken from the *patchwork* FIR filter architecture are depicted. 6 elements are grouped in 3 couples using the EqualGroupsKMeans clustering algorithm that leads to the creation of balanced clusters.

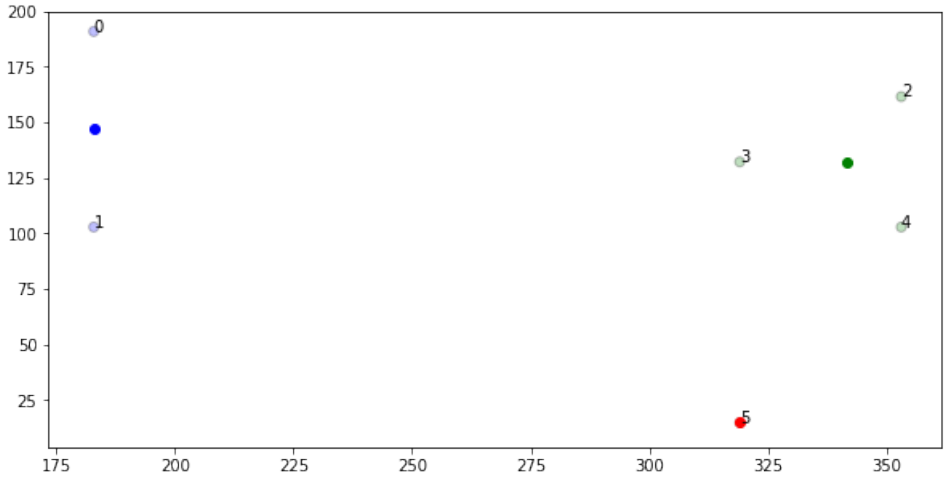


Figure 4.16. FIR fMACs - *patchwork* - n=6, k=3

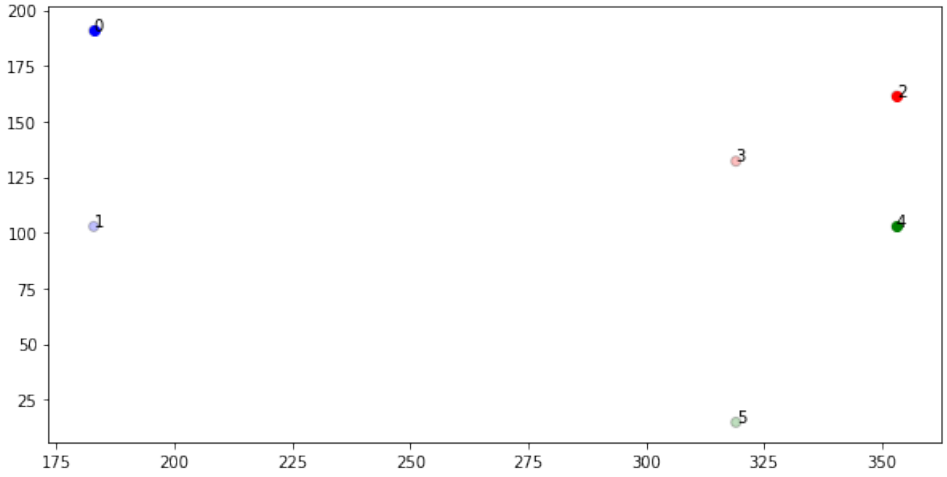


Figure 4.17. FIR fMACs - *patchwork* **balanced** - n=6, k=3

4.1.4 Generation of new code files

Taking into account the 4 optimum clustering solutions obtained through the 4 different Graphviz tool DFGs, it is possible to generate new header files on the basis of the original one.

In this case, also the cpp file has to be modified as well as the header since,

even though the architecture is described in the header file, the function definitions must be in the .cpp. This is carried out employing a similar process to the one used for the header file.

A Python code called "generate_clusters" has been written to read the original header code file and to detect the functions called "mac" followed by a number.
The name of each MAC instance is replaced with a new name representing the cluster it has been mapped in. For example, the function "mac1" is turned into "mac_new2" if it has been mapped into cluster 2 by K-Means.

Subsequently, a #pragma command is written to limit each new mac function to only 1 instance. This command is inserted in order to simulate the sharing of one MAC for each chosen group of function calls: the compiler can only have 1 instance for each cluster of functions as if it was a physical resource being used from different lines of the code.

Lastly, it is useful to create some random cluster solutions in order to assess the goodness of the optimum ones found through the smart resource sharing. This is obtained shuffling the vector "labels", it is an array generated by the K-Means algorithm in which each position corresponds to one original unit and the content of each cell to the clusters to whom the unit belongs.
Afterwards, the .cpp and the header file can be printed modifying the function names using the same process described previously.

4.1.5 Synthesis and Place & Route

The new codes, optimal solutions and random ones, are imported in Vivado. One script for each solution has been generated in order to carry out the synthesis and place & route. This process requires some hours to exploit the placement and routing of each project.
At the end of all the simulations, Vivado returns different report files from which it is possible to get the information necessary to evaluate the resulting circuits.

- The resource usage required by each process, that can be extracted from the *"fir_utilization_routed.rpt"* report file that contains the Utilization Design Information. The report is obtained after the place and route and shows how many items are needed in order to synthesize a project, these quantities are expressed as a percentage of the used resources over

the available ones.
As previously outlined, the value used to assess the goodness of a solution is the maximum percentage of utilization of one kind of resource, since this is the fundamental limit for the synthesis on FPGAs.

- The minimum and maximum latency, measured in number of clock cycles. This information is included in the *"fir_csynth"* file, a report file generated by Vivado after the synthesis.

- The maximum path delay, that corresponds to the critical path delay of the circuit. This parameter is included in the *"fir_timing_paths_routed"* file, a report file generated by Vivado after the place and route where the delays of the 10 longest paths are illustrated.
  This value puts an upper bound to the frequency of the designed circuit.

4.1.6 Results comparison and behaviour analysis

Another Python code called "compare_reports" is created to compare the computed circuit parameters and to generate new graphs to visually confront them.

All the *"fir_timing_paths_routed"* report files from all the different solutions are read and the path timings are extracted. They include 10 paths each, for every solution the maximum delay and the root mean square of the delays is computed and printed in the output file *"fir_random_delay.txt"*. First, the optimal solutions found with the clustering method are analyzed, followed by the 10 random ones.

Afterwards, the post-implementation resource usage is evaluated from the *"fir_utilization_routed.rpt"* file. All the report files from different solutions are read to extract information about the percentage of FPGA resources employed such as Slices, DSPs, Flip Flops, BRAMs and Shift Register.

Once all of the parameters have been extracted from the report files, it is possible to plot a **Delay-Area diagram**. In the y-axis, the total execution time of each solution is plotted as the multiplication of the maximum path delay and the numbers of clock cycles. There could be two versions of this graph since Vivado provides a minimum and a maximum latency value, but in this case, both values lead to the same graph proportionally.

In the x-axis is plotted the Resource Usage maximum percentage, this parameter gives an impression of how much the chip area increases or decreases changing the sharing arrangements of the MAC units.
The resulting graph includes several points, one for each resource sharing arrangement, and they are marked with the name of the respective solution or with a number if related to a random one.

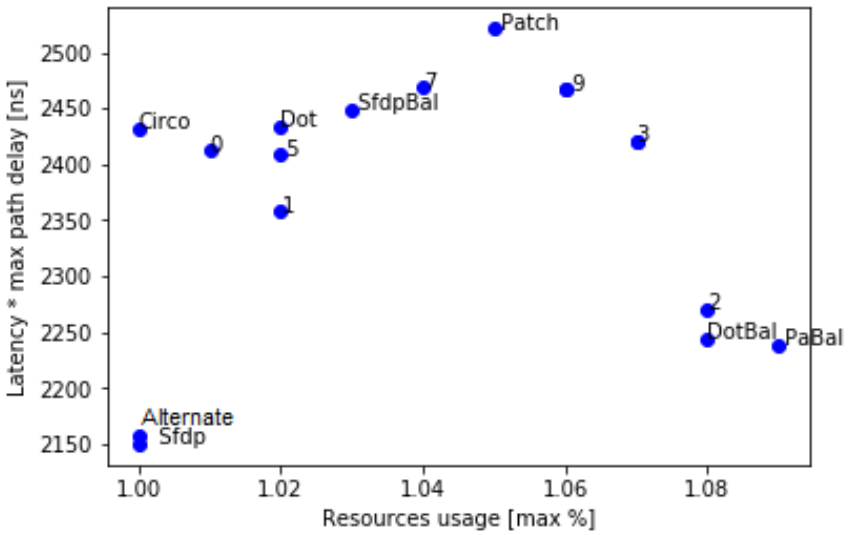


Figure 4.18. FIR filter - Delay-Area diagram

	Dot	**DotBal**	**Sfdp**	**SfdpBal**	**Circo**	**Patch**	**PatchBal**
Tex[µs]	2.434	2.243	2.150	2.449	2.432	2.521	2.238
res[%]	1.02	1.08	1.00	1.03	1.00	1.05	1.09

Table 4.1. FIR filter - Total execution time [ns] and Maximum FPGA Resource Usage percentage [%] of different sharing solutions

Conclusions on FIR filter analysis

From the graph in Figure 4.18 and Tables 4.1 and 4.2, it is possible to draw some conclusions on the smart resource sharing employed.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Tex[**µs**]	2.412	2.358	2.270	2.420	2.158	2.408	2.466	2.468	2.420	2.466
res[%]	1.01	1.02	1.08	1.07	1.00	1.02	1.06	1.04	1.07	1.06

Table 4.2. FIR filter - Total execution time [ns] and Maximum FPGA Resource Usage percentage [%] of different sharing solutions

First of all, it is worthwhile to examine the performance results obtained by the two theoretical optimal FIR sharing solutions.

- **In order optimal solution** This solution corresponds to the *sfdp balanced* configuration: [0 0 1 1 2 2].
  This simple combination was expected to minimize the muxing and routing due to the closeness of the shared cells between each other and the proximity of the elements that communicate with them
  On the basis of the reported simulation results, it is possible to confirm that the *In order* sharing configuration leads to a great resource-saving confronted with the majority of the other cases, 1,03% of FPGA Slices allocated, even if it does not correspond to the best trade-off found in the study.

- **Alternate optimal solution** This solution is obtained distributing the components alternately into the sharing clusters: [0 1 2 0 1 2].
  This second sharing configuration was expected to guarantee an optimal trade-off between area-saving and path delay reduction since this distribution allows to use different components in consecutive clock cycles. Therefore, the waiting time necessary to use the same resource again is minimized.
  The hypothesized delay reduction is particularly evident in the obtained results: the Alternate solution is confirmed to be one of the best resource sharing configurations for a direct form FIR filter, both in terms of area occupation and maximum path delay.

Furthermore, from the obtained plotting it appears clear that the *Alternate solution* is not the only sharing configuration that finds an optimum compromise in terms of both timing delay and resource usage.
The other best resource sharing configurations for a direct form FIR filter correspond to the **SFDP** solution: [0 0 1 2 2 2].
It is an unbalanced solution, detected applying the K-Means algorithm to the direct form FIR filter DFG visually rendered through the *sfdp* Graphviz tool.

Another remarkable solution is the unbalanced *dot* one [0 0 0 1 1 2], found through the rendering of the DFG using *dot* Graphviz tool. It leads to a great resource-saving confronted with the majority of the other cases.

In the end, it results clear that the first two combinations taken into account are the ones that minimize both area usage and the maximum path delay.
Therefore, this example highlights the goodness of the clustering algorithm applied to a DFG opened through the SFDP Graphviz tool.
On the other hand, the *dot* solution results in a significant Resource saving in spite of some increase in the critical path delay.

4.1.7 Clustering Cost Diagrams

In order to prove that the proximity method hypothesized at the beginning of this study produces improvements in Area Usage and, probably, also on Timing, a new parameter is computed:
The Clustering Cost, defined as the sum of the distances of each unit to its centroid. This parameter gives an impression of the goodness of the sharing configuration found: the closer the items are to their clustering centroid the smallest the clustering cost is.
The clustering cost of each solution is then plotted with the respective resource usage and maximum path delay. From these graphs, it is possible to analyze if there is a positive or negative correlation of the clustering cost with area and delay.
In order to obtain the desired plots some steps are required:

1. Some random cluster combinations are generated through the already outlined procedure. In this case, the clusters are kept balanced, 2 MACs each, in order to have a coherent comparison of their clustering costs.
   It is not possible to mix different graphical representation tools for this analysis because the various solutions must be allocated to the same Data Flow Graph drawing. Therefore two graphical tools, *dot* and *sfdp*, are adopted separately and only the final resulting graphs can be compared.

2. The newly written codes are imported in Vivado and the synthesis and

place & route are carried out.

3. Among all the simulations the solutions whose synthesis resulted in the same Initiation Interval (II) value were grouped. The Initiation Interval is the number of clock cycles that must pass between the start of two consecutive loop iterations.

4. The Clustering Cost is computed, from the coordinates of the MACs in the original DFG graph, as the sum of distances of each unit to its centroid, as shown in Figure 4.19
   Even if the DFG is the same, the distances computed for each sharing

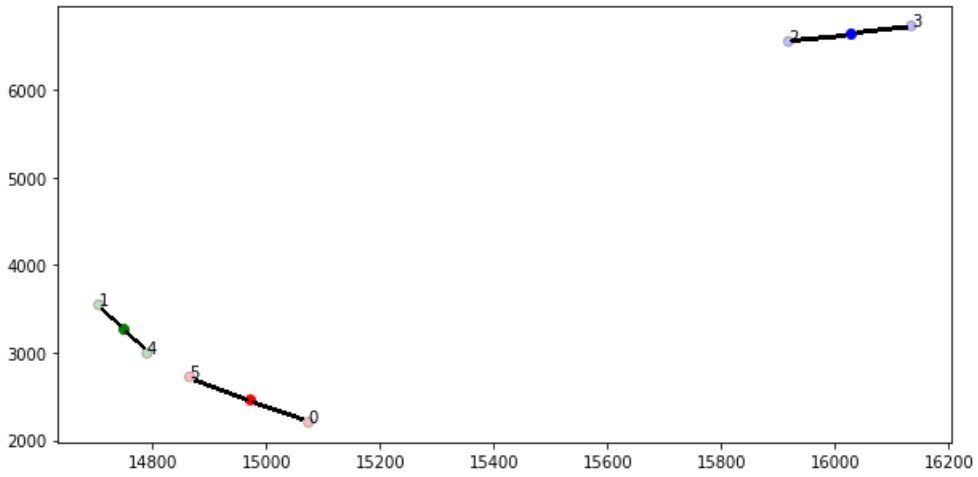


Figure 4.19. Computation of the Clustering Cost

solution are different from one another since the clusters and centroids change every time.

5. In a Python code two kinds of diagrams for each Graphviz tool are plotted:

   - **Delay-Clustering Cost diagram**: The total execution time of each solution is plotted as the multiplication of the maximum path delay and the numbers of clock cycles on the y axis. Once again there could have been two versions of this graph since Vivado provides a minimum and a maximum latency value but using one or the other did not bring any changes in the diagram.
     The Clustering Cost, the sum of distances of each unit to its centroid, is plotted on the x axis.

The resulting graphs, in Figure 4.20 and Figure 4.21, include several points, one for each resource sharing arrangement. They are marked with the name of the respective solution or with a number if related to a random solution.

From these data it is possible to extract the linear regression in

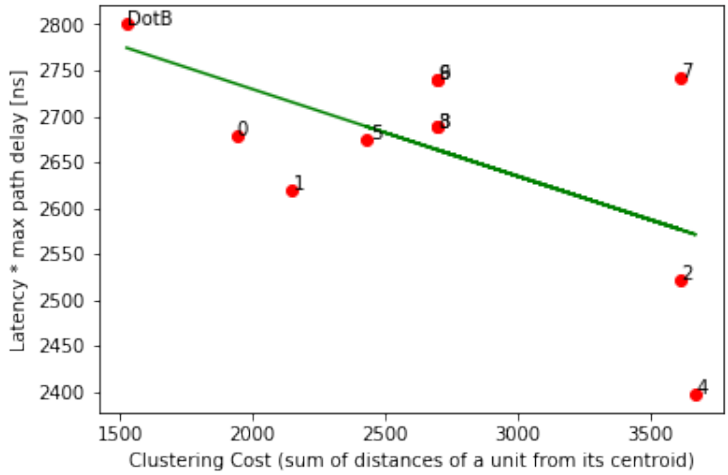


Figure 4.20. **dot** Max path delay - Clustering Cost diagram

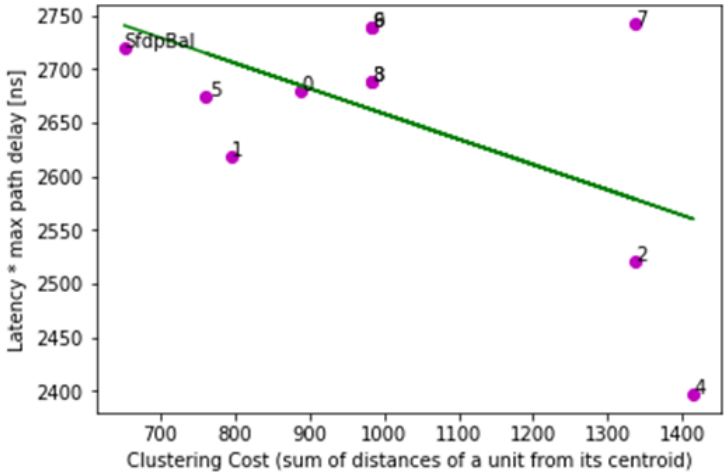


Figure 4.21. **sfdp** Max path delay - Clustering Cost diagram

order to model the evolution of the results. The resulting line shows

a decreasing trend along with the increasing of the clustering cost, both using *dot* and *sfdp*. This reflects the fact that applying a smart proximity-based resource sharing to a FIR filter architecture the delay is likely to increase.

- **Area-Clustering Cost diagram**: The Resource Usage maximum percentage is plotted on the y-axis, this parameter gives an impression of how much the chip area increases or decreases changing the sharing arrangements. Whereas, the Clustering Cost is plotted on the axis.
  The resulting graphs, Figure 4.22 and Figure 4.23, include one dot for each resource sharing arrangement. Each spot is marked with the respective solution's name or with a number if related to a random one.

  From these data it is possible to extract the linear regression

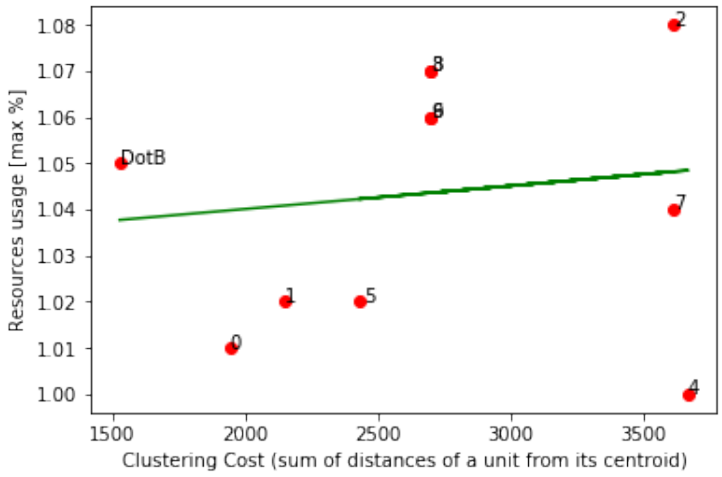


Figure 4.22. **dot** Max percentage resource usage - Clustering Cost diagram

in order to model the evolution of the results. The resulting line shows an increasing trend along with the growth of the clustering cost, both using *dot* and *sfdp*. This reflects the fact that applying a smart proximity-based resource sharing to a FIR filter architecture the FPGA Resource Usage is likely to decrease more the closer the items are to their cluster centroids.

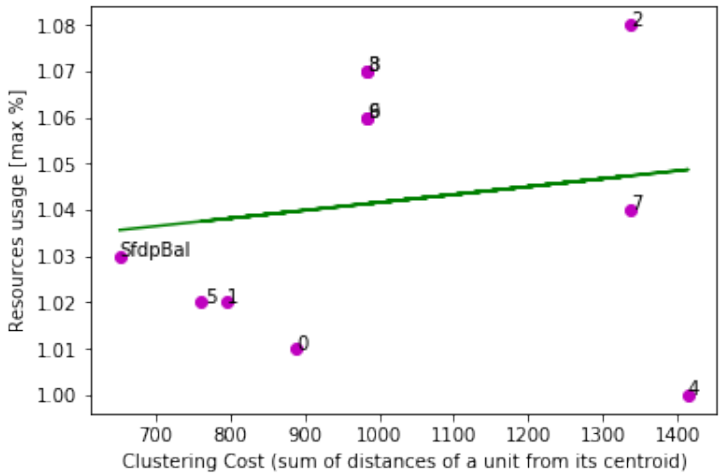


Figure 4.23. **sfdp** Max percentage resource usage - Clustering Cost diagram

Clustering cost analysis conclusion

The goal of this analysis was to find if there was a strong relationship between the Clustering Cost and the Resource Usage and the Maximum path delay of the resulting circuit.
The test has been carried out only on balanced cluster arrangements and has been repeated for two DFG representations generated through two Graphviz tools, dot and sfdp.
The resulting data have been plotted and compared in the previously shown diagrams. Extracting the linear regression it is easy to highlight the evolution of the area and delay along with the growth of the clustering cost.

The most important result is that employing both *dot* and *sfdp* the gradient of evolution is the same.
The total execution time tends to decrease with the increment of the clustering cost while the Area occupation increases.

These results confirm that the proximity based smart resource sharing works of this FIR filter example code reducing the resource allocation in an FPGA, while it may not have the same benefit on timing.

4.2 KNP_MOD

Kahn process networks (KPNs) are models of computation that describe sequential processes connected with each other by FIFO channels. The process network is characterized by a deterministic behaviour, this means that its computation results do not depend on the delay of each process.

4.2.1 Detecting and isolating units to be shared

The *knp_mod* cpp code A.4 is full of chained loops with numerous multiplications and additions. Therefore, it was evident that isolating and sharing multiply and accumulate units (MACs) could lead to significant improvements in terms of Area Usage of the resulting chip.

The main loop includes two kinds of operations that can be useful to isolate and share: the first one is a multiplication and subsequent addition of integer variables, while the second involves multiplication and accumulation of float variables. Therefore, two kinds of mac functions are created: *imac*, for integer MAC operations, and *fmac*, for float MAC operations.

There are 5 integer MAC operations and 6 floating ones, so 11 corresponding functions are created to cover all the biggest computational units of the main chained loop. Each newly written function must include a #pragma command that prevents the *inlining* in order to make the compiler consider each MAC as a separate functional unit: *#pragma HLS inline off*.

In this case, the operations are described in the .cpp file, therefore it is the one that must be modified. As a first step, the 11 functions must be written with different names in order to be distinguished between one another, so they are called: *fmac0, fmac1, fmac2, fmac3, fmac4, fmac5* and *imac6, imac7, imac8, imac9, imac10*.

4.2.2 Synthesis without sharing

In order to properly apply the smart resource sharing, it is necessary to know the position in which the MAC units would be placed after the synthesis of the last version of the knp_mode code. At this point, each MAC is described as an independent hardware component either computing integers or float variables.

Afterwards, the code is imported in Vivado where it is possible to carry out the synthesis on the *zynq* FPGA *xc7z007sclg400-1*. This FPGA has been chosen because it is the smallest one on which this circuit can be synthesized: it is composed of 66 DSP Slices and 100 18k RAM Blocks.
The perks of working on a well-fitted FPGA have already been exposed in the previous chapter: it allows to have relevant and visible feedback to the hardware changes introduced by the resource sharing configurations.

In Vivado it is possible to carry out the synthesis on the chosen FPGA through a previously written script. After the synthesis, the DFG is generated and saved in a DOT file.

4.2.3 Comparison of different sharing arrangements of int and float MACs

Multiple sharing configurations can be applied to the *knp_mod* architecture. Finding the best one is not trivial, in fact, it must satisfy both area and timing constraints.
This study aims to find the sharing solution that optimizes both these factors. Nevertheless, it is not possible to minimize area and delay at the same time for they have a negative correlation. Therefore, the goal is to find a good trade-off between the two parameters.

Development of different sharing arrangements

For each sharing arrangement, i.e. number of fMACs and iMACs, it was useful to analyze two solutions. The former one is the optimum proximity solution, obtained through the employment of the K-Means clustering algorithm on the DFG opened using *dot*. An example of the optimal sharing combination found for the *2+3* solution can be observed in Figure 4.24 and 4.25. The latter is the worst proximity solution, this configuration is achieved by observing the graphical representation of the *dot* DFG and manually allocate the furthest apart units to the same cluster. This procedure is meant to obtain exactly the opposite sharing configuration than the one achieved through the locality method.
Each solution is created modifying the original cpp code file.

First of all, the functions called "fmac" and "imac" followed by a number are renamed, their new name represents the cluster they have been mapped

into. For example the function "*fmac1*" might be turned into "*fmac_new2*" if it has been mapped into the cluster 2.

Subsequently, a #pragma command is written to limit each new mac function to only 1 instance. This command is inserted in order to simulate the sharing of one fMAC or iMAC for each chosen group of function calls: the compiler can only have 1 instance for each cluster of functions as if it was a physical resource being used from different lines of the code.

Subsequently, some #pragma commands, one for each cluster function, are written to limit function instances to only 1. E.g. *#pragma HLS allocation instances=fmac_new0 limit=1 function*.
This procedure is fundamental to create several .cpp code files that describe different sharing configuration of the knp_mod. Limiting the number of instances for fmac and imac functions corresponds, in fact, to forcing the sharing of resources: the compiler can only create one instance for each function as if it was one physical resource being used from different lines of the code.

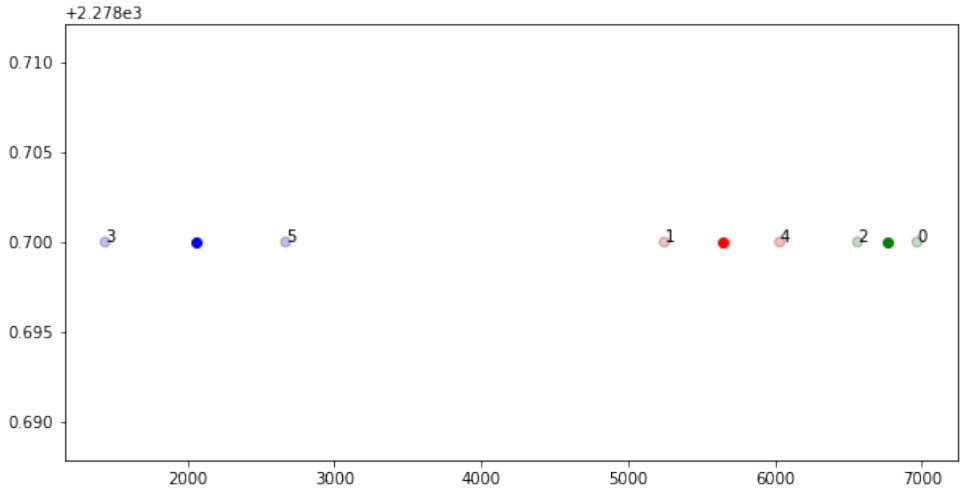


Figure 4.24. knp_mod 2+3 - *dot* float MACs k=3

Synthesis and parameters computing

The codes are imported in Vivado where the synthesis and Place & Route are carried out on the *zynq* FPGA *xc7z007sclg400-1*.

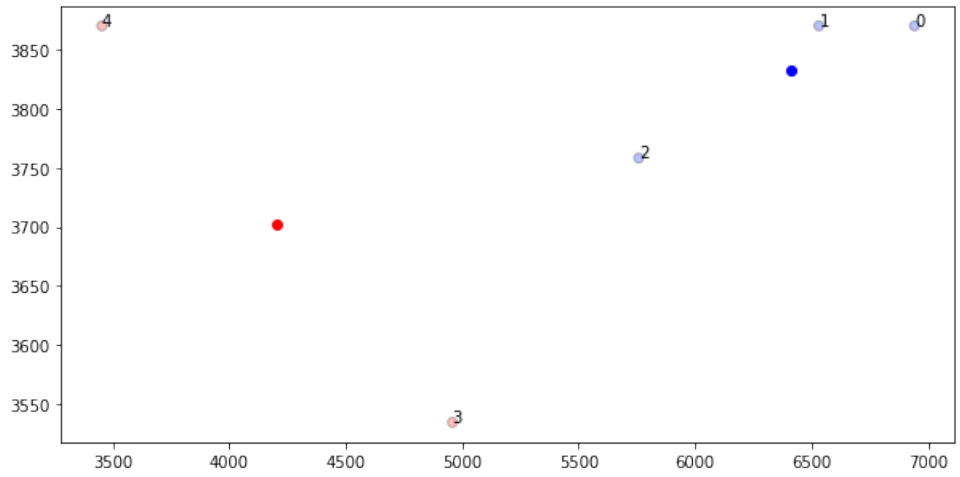


Figure 4.25. knp_mod 2+3 - *dot* float MACs k=3

When all the simulations have been developed, Vivado returns different report files.

- *"knp_utilization_routed.rpt"* is the report file that contains the Utilization Design Information. The report is obtained after the place and route and shows how many items are needed in order to synthesize a project. These quantities are expressed as a percentage of the used resources over the available ones.
- *"knp_csynth"* is the report file generated after the synthesis where it is possible to find information on the minimum and maximum latency, measured in number of clock cycles.
- *"knp_timing_paths_routed"* is the report file generated by Vivado after the place and route where the delays of the 10 longest paths are illustrated. The maximum path delay corresponds to the critical path delay of the circuit.

Sharing Arrangements Comparison

Once all the parameters have been obtained from the report files, it is possible to plot a *Delay-Area diagram*. The total execution time of each solution is plotted on the y-axis as the multiplication of the maximum path delay and the numbers of clock cycles.

On the x-axis is plotted the Resource Usage maximum percentage, this parameter gives an impression of how much, in percentage, the FPGA employed resources increase or decrease changing the sharing arrangements of the MAC units.

The resulting graph includes several points, one for each resource sharing arrangement, and they are marked with the name of the respective solution. The first number refers to the number of int MAC units while the second to the number of float MAC units. The ones marked with the "w" are the worst locality solutions, while the other ones are the results of the optimum clustering combinations.
E.g. *1+2*: 1 int MAC and 2 float MACs, the optimal solution for locality obtained through clustering. *1+3w*: 1 int MAC and 3 float MACs, wrong solution for locality obtained manually from the DFG.

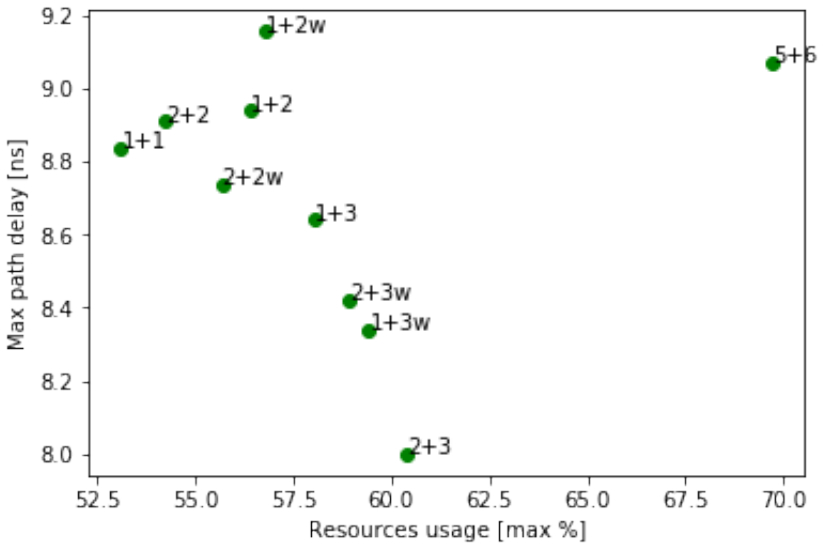


Figure 4.26. knp_mod - Sharing Arrangements Comparison

From Figure 4.26 and Table 4.3, it is possible to appreciate that, as expected, the worst version of each combination, indicated with *w*, is always employing a bigger number of resources and in some cases it requires a longer period.

	1+1	**1+2**	**1+2w**	**1+3**	**1+3w**	**2+2**	**2+2w**	**2+3**	**2+3w**	**5+6**
T[ns]	8.836	8.939	9.156	8.641	8.338	8.909	8.737	7.999	8.42	9.072
res[%]	53.11	56.43	56.79	58.04	59.43	54.23	55.68	60.41	58.91	69.70

Table 4.3. knp_mod - Maximum path delay [ns] and Maximum FPGA Resource Usage percentage [%] of different sharing solutions

From the diagram it results evident that the Pareto Optimal solutions are the ones arranged in a quarter-circle closer to the origin of the axes.
The **2+3** solution consists in sharing 2 integer MAC units and 3 MAC units, this arrangement is the best in terms of reduction of critical path delay but one of the worst in terms of Resource Usage.
The **1+1** solution, on the other hand, consists in sharing only 1 integer MAC unit and 1 MAC unit, this arrangement is the best in terms of Resource Usage, as can be easily imagined, but it results in the a longer critical path delay.

In order to obtain a good trade-off between area consumption and path delay, the solution **1+3** has been chosen as the best solution. It consists in the sharing of 1 int MAC and 3 float ones and must to be thoroughly analyzed though further simulations.
Also the **2+2** solution, 2 int and 2 float MACs, is studied as the previous arrangement in order to confront their results.

4.2.4 Smart Sharing through K-Means Clustering

The Data Flow Graph is described in a DOT file that can be visually rendered by 4 different Graphviz tools: *dot, sfdp, circo* and *patchwork*.
Afterward, through a Python code, the nodes indicating the int MAC units and the float MAC units must be detected in the graph and their coordinates extracted in two different arrays. The procedure of generating the optimal solution through a clustering algorithm can be repeated for each Graphviz tools, since they produce different outcomes.

It is possible to compose different sharing configuration to be applied to the *knp_mod* architecture, in particular, 2 combinations have been picked:

- **1 int MAC and 3 float MACs:** Using the K-Means Clustering Algorithm 3 clusters have been created from the 6 original float MACs while all the integer MAC units are grouped together.

This means that the filter area is reduced by 3 float MAC units and each couple of the former operations have to share a single fMAC, whereas all the former integer multiply and accumulate operations must share the same iMAC.

This solution has been chosen because it is the best trade-off between area consumption and path delay. Moreover, it seems clever to favour more complex units such as float MACs that require longer delays rather than the integer MAC units that need less computing time. Therefore, it seems reasonable to reduce the number of *imac* function instances down to 1 in order to keep the number of *fmac* instances up to 3.

In order to obtain balanced clusters, composed of the same number of elements, it is possible to apply Equal Groups KMeans Algorithm (k=3) to the float MACs.

In Figure 4.27 and 4.28 is shown the application of the K-Mean algorithm to the DFG opened through *dot* Graphviz tool.

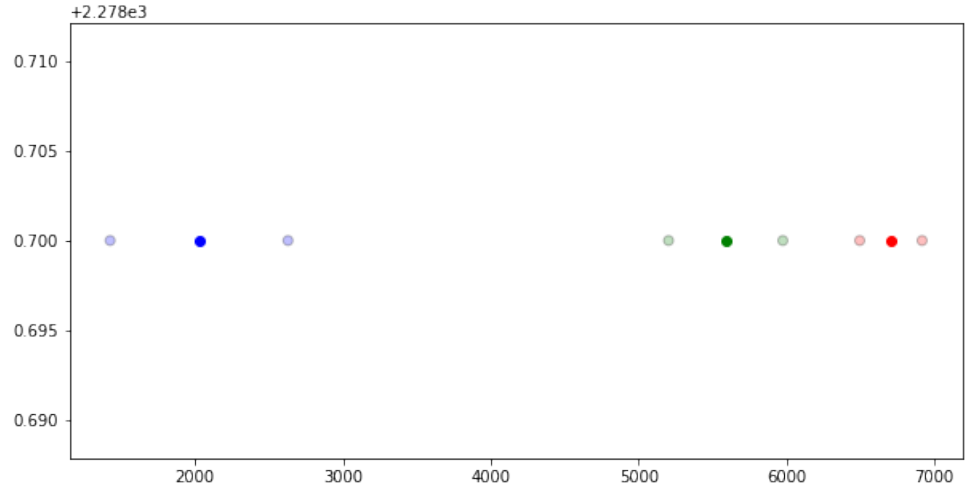


Figure 4.27. knp_mod 1+3 - *dot* float MACs k=3

- **2 int MACs and 2 float MAC:** Using the K-Means Clustering Algorithm 2 clusters have been created from the 6 original float MACs and 2 from the original integer MAC units.
  This means that the filter area is reduced by 4 float MAC units and each

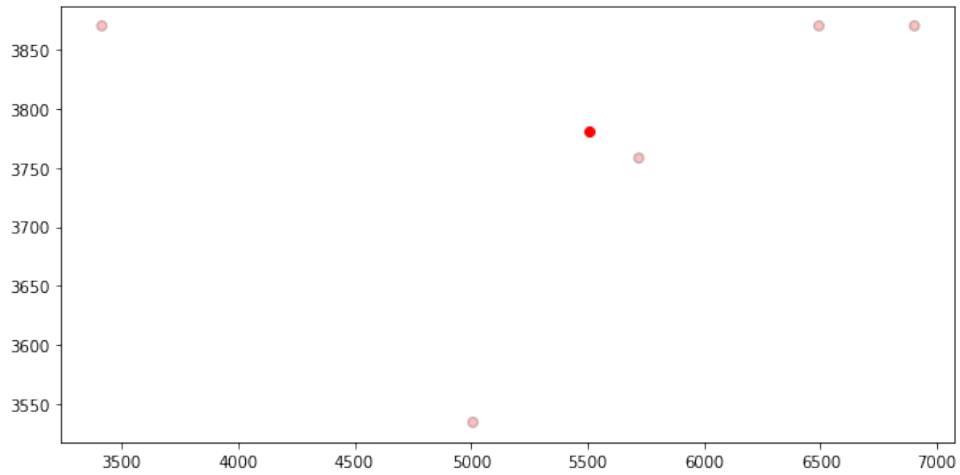


Figure 4.28. knp_mod 1+3 - *dot* int MACs k=1

triplet of the former operations have to share a single fMAC, whereas a couple and a triplet of the former integer multiply and accumulate operations must share the same iMAC.

This solution has been chosen because it seems reasonable to keep the number of *imac* and *fmac* the same, scaling the total amount of MAC units in the circuit down to 4.

In order to obtain balanced clusters, composed of the same number of elements, it is possible to apply Equal Groups KMeans Algorithm (k=3) to the float MACs.

In Figure 4.29 and 4.30 is shown the application of the K-Mean algorithm to the DFG opened through *dot* Graphviz tool.

In Figure 4.31 and 4.32 is shown the application of the EqualGroupsKMean algorithm to the DFG opened through *dot* Graphviz tool.

4.2.5 Generation of new code files

Taking into account the 4 optimum clustering solutions obtained through the 4 different Graphviz tool DFGs, it is possible to generate new .cpp files on the basis of the original one.
In this case, it is not necessary to change the header file since it only contains

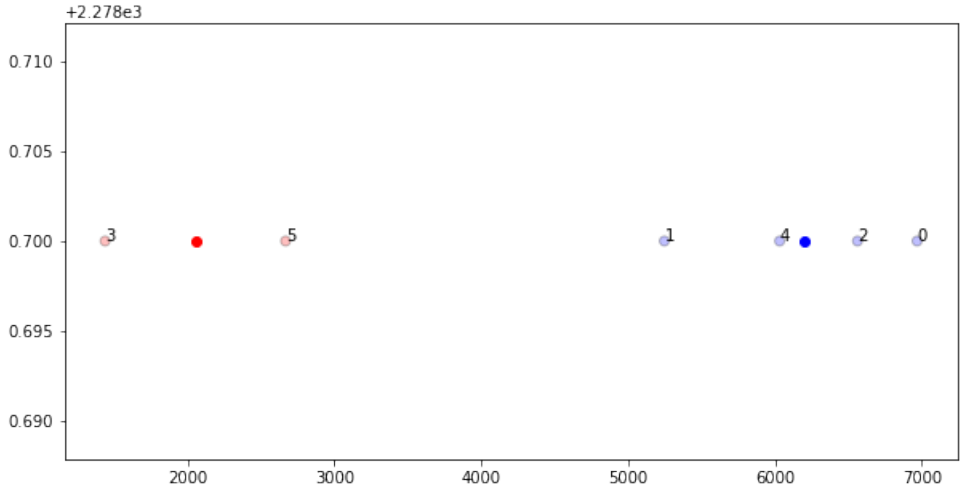


Figure 4.29. knp_mod 2+2 - *dot* float MACs k=2

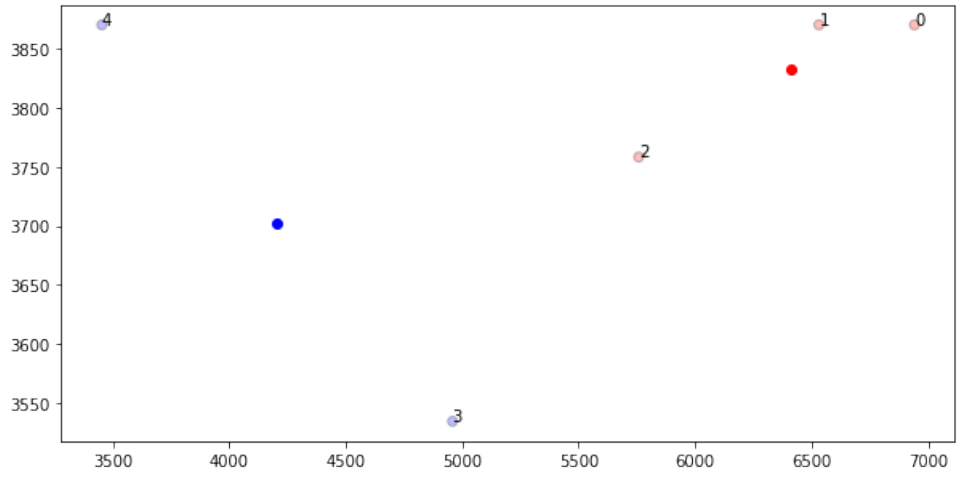


Figure 4.30. knp_mod 2+2 - *dot* int MACs k=2

the function declarations and it can be done once for all of the simulations.

A Python code called "*generate_clusters*" has been written to read the original cpp file and to detect the functions called "fmac" and "imac" followed by a number.
The name of each MAC instance is replaced with a new name representing the cluster it has been mapped in. For example the function "*fmac1*" might

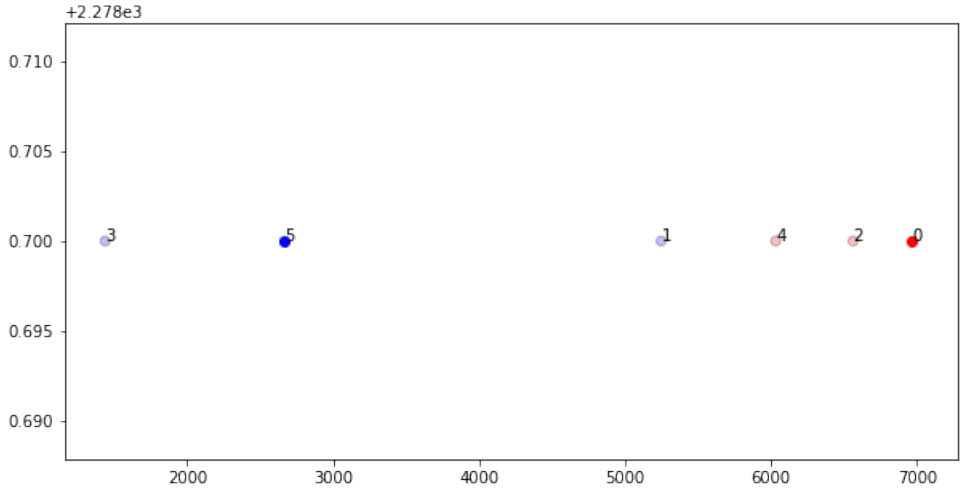


Figure 4.31. knp_mod 2+2 - *dot* float MACs k=2 - *balanced*

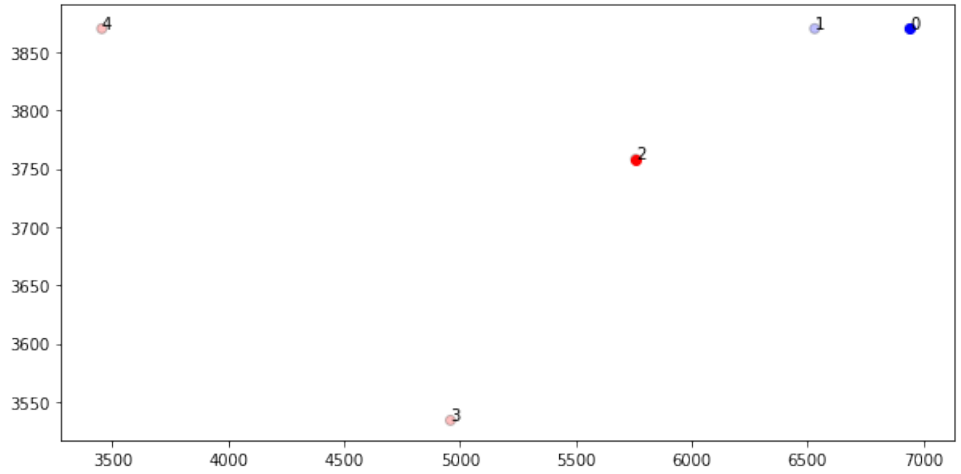


Figure 4.32. knp_mod 2+2 - *dot* int MACs k=2 - *balanced*

be turned into "*fmac_new2*" if it has been mapped into the cluster 2 by K-Means, or the function "*imac7*" might be turned into "*imac_new6*" .

Subsequently, a #pragma command is written to limit each new mac function to only 1 instance. This command is inserted in order to simulate the sharing of one fMAC or iMAC for each chosen group of function calls: the compiler can only have 1 instance for each cluster of functions as if it was a physical resource being used from different lines of the code.

In the same way, as the optimum solution has been described in the cpp file, it is useful to create some random cluster solutions in order to assess the goodness of the generated solutions.
This is obtained shuffling the vector "labels", it is an array generated by the K-Means algorithm in which each position corresponds to one original unit and the content of each cell to the clusters to whom the unit belongs.
In case of the **1 iMAC - 3 fMACs** arrangement only the float *labels* vector must be shuffled, while for the **2 iMACs - 2 fMACs** arrangement both the float *labels* vector and the integer *labels* vector must be randomly shuffled.

Afterwards, the .cpp and the header file can be printed modifying the function names using the same process employed for the optimal solutions.

4.2.6 Synthesis and Place & Route

The new codes, optimal solutions and random ones, are imported in Vivado. One script for each solution has been generated in order to carry out the synthesis and place & route. This process requires some hours to exploit the placement and routing of each project.
At the end of all of the simulations, Vivado returns different report files from whom it is possible to get the necessary information to evaluate the resulting circuits.

- The resource usage required by each process, that can be extracted from the *"knp_utilization_routed.rpt"* report file that contains the Utilization Design Information. The report is obtained after the place and rout and shows how many items are needed in order to synthesize a project, these quantities are expressed as a percentage of the used resources over the available ones.
  As previously outlined, the value used to assess the goodness of a solution is the maximum percentage of utilization of one kind of resource, since this is the fundamental limit for the synthesis on FPGAs.

- The minimum and maximum latency, measured in number of clock cycles. This information is included in the *"knp_csynth"* file, a report file generated by Vivado after the synthesis.

- The maximum path delay, that corresponds to the critical path delay of the circuit. This parameter is included in the *"knp_timing_paths_routed"*

file, a report file generated by Vivado after the place and route where the delays of the 10 longest paths are illustrated. This value puts an upper bound to the frequency of the designed circuit.

4.2.7 Results comparison and behaviour analysis

Two new Python codes called *compare_reports1+3* and *compare_reports2+2* are created to compare the computed circuit parameters of the two sharing arrangements: 1 iMAC + 3 fMACs and 2 iMACs and 2 fMACs.

1 int MAC + 3 float MACs

All of the parameters are extracted from the report files in order to plot a *Delay-Area diagram*.
The total execution time of each solution is plotted on the y-axis as the multiplication of the maximum path delay and the numbers of clock cycles. There could be two versions of this graph since Vivado provides a minimum and a maximum latency value, but in this case, both values lead to the same graph proportionally.

In the x-axis is plotted the Resource Usage maximum percentage, this parameter gives an impression of how much the chip area increases or decreases changing the sharing arrangements of the MAC units.

The resulting graph includes several points, one for each different sharing solution. Some of them are marked with the name of the respective solution, these correspond to the optimal solutions obtained using K-Means combined with the various Graphviz tools. If the name is followed by "balanced", *EqualGroupsKMeans* has been employed. When the label is just a number, the solution is obtained by the random sharing combinations.

	Opt	**Furthest**	**Sfdp**	**SfdpBal**	**Circo**	**CirBal**	**Patch**	**PatBal**
Tex[**s**]	138.4	199.2	202.4	174.0	194.4	181.6	137.3	178.9
res[%]	58.05	58.20	59.39	60.64	59.36	60.50	50.09	61.55

Table 4.4. knp_mod - Total execution time [ns] and Maximum FPGA Resource Usage percentage [%] of different sharing solutions

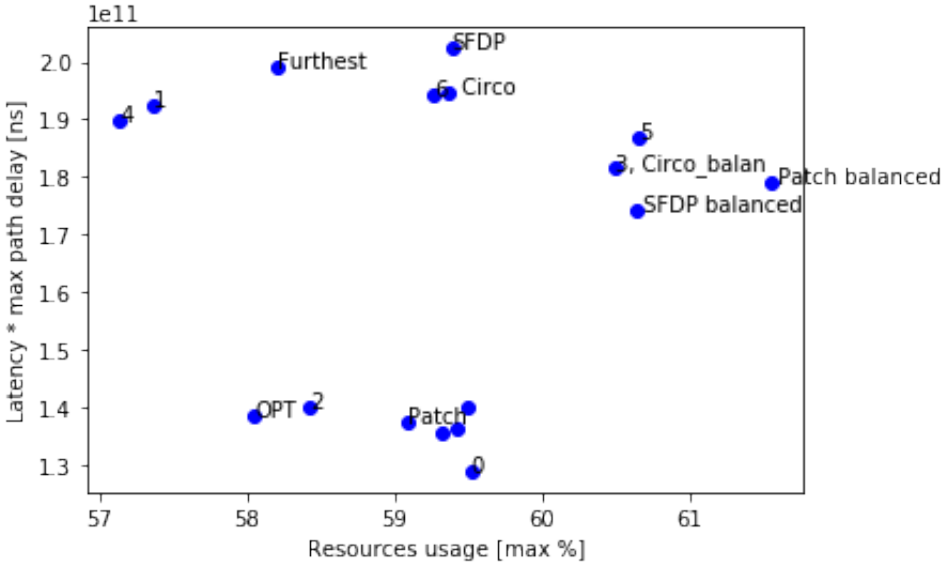


Figure 4.33. knp_mod 1+3 - Delay-Area diagram

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Tex[**s**]	128.8	192.5	139.8	181.6	189.7	186.8	194.2	135.4	136.2	139.8
res[%]	59.52	57.36	58.43	60.50	57.14	60.66	59.27	59.32	59.43	59.50

Table 4.5. knp_mod - Total execution time [ns] and Maximum FPGA Resource Usage percentage [%] of different sharing solutions

From the graph in Figure 4.33 it is possible to draw some conclusions on the smart resource sharing employed on the configuration with 1 int MAC and 3 float MACs. It is evident that the configuration marked as "OPT", that corresponds to the optimal clustering solution opened with *dot*, is the best trade-off in terms of both timing delay and resource usage.
Whereas, the SFDP optimal locality solution and "Furthest", the worst *dot* solution, result in a much longer critical path delay.

This highlights the goodness of the clustering algorithm applied on a DFG opened through "dot" Graphviz tool, that guarantees the best trade-off between area consumption and path delay.

2 int MACs + 2 float MACs

The parameters are again extracted from the report files in order to plot a *Delay-Area diagram*.
The total execution time of each solution is plotted on the y axis as the multiplication of the maximum path delay and the numbers of clock cycles.
In the x axis is plotted the Resource Usage maximum percentage, this parameter gives an impression of how much the chip area increases or decreases changing the sharing arrangements of the MAC units.

The resulting graph includes several points, one for each different sharing solution. Some of them are marked with the name of the respective solution, these correspond to the optimal solutions obtained using K-Means combined with the *dot* Graphviz tool. If the name is followed by "balanced", *EqualGroupsKMeans* has been employed, while *FURTH* indicates the worst case solution in which the furthest items are allocated in the same cluster. When the label is just a number, the solution is obtained by of the 14 random sharing combinations.

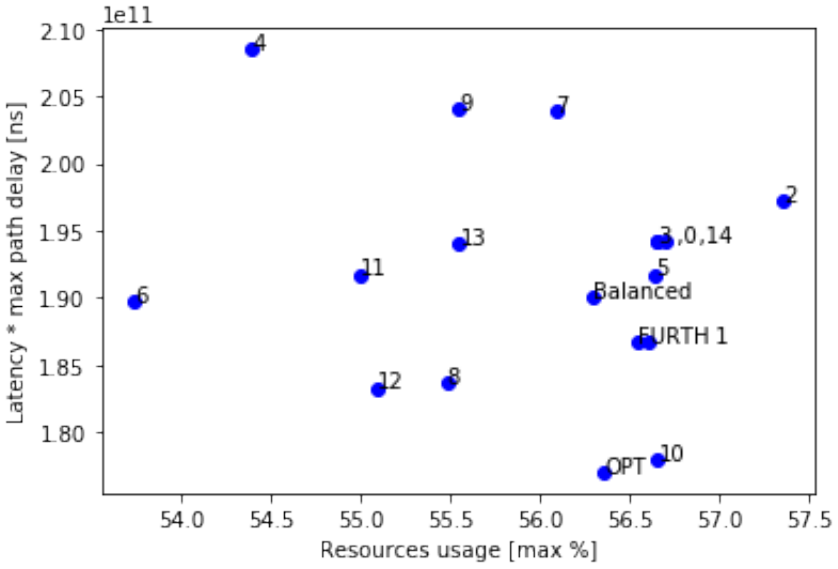


Figure 4.34. knp_mod 2+2 - Delay-Area diagram

From the graph in Figure 4.34 it is possible to draw some conclusions on

	Opt	**Bal.**	**Furth.**	**0**	**1**	**2**	**3**	**4**	**5**
Tex[**s**]	177.0	190.0	186.7	197.2	194.2	208.5	191.7	208.5	191.7
res[%]	56.36	56.30	56.55	56.66	56.61	57.36	56.66	54.39	56.64

Table 4.6. knp_mod - Total execution time [ns] and Maximum FPGA Resource Usage percentage [%] of different sharing solutions

	6	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
Tex[**s**]	189.8	203.9	183.7	204.0	178.0	191.7	183.3	194.1	194.2
res[%]	53.73	56.09	55.48	55.55	56.66	55.00	55.09	55.55	56.70

Table 4.7. knp_mod - Total execution time [ns] and Maximum FPGA Resource Usage percentage [%] of different sharing solutions

the smart resource sharing employed on the configuration with 2 int MACs and 2 float MACs.
The configuration marked as "OPT" corresponds to the optimal unbalanced proximity solution opened with *dot* obtained through the clustering algorithm *K-Means*. This solution is the one that minimizes the maximum path delay more that any of the others but is definitely not the best one for area occupation.
Whereas, the *Balanced* and *Furthest* solutions result in a much longer critical path delay for a similar maximum resource usage percentage.

Results

In conclusion, in this section 2 of the various sharing arrangements have been further studied.
Applying the smart sharing algorithm to a *dot* DFG, it results evident that the "1 int MAC and 3 float MACs" configuration produces better results in terms of path delay than the "2 int MACs and 2 float MACs" configuration. Indeed, all the solution of the 2+2 arrangement result in a critical path delay equal or superior to 1,8 ns while the "Optimal" one of the 1+3 arrangement stays under 1,4 ns. Nevertheless, this reduction in the timing delay is payed by a slight increase in area consumption.
In fact, the *1+3 OPT* requires 58% of one type of FPGA resources while the *2+2 OPT* only the 56,4%.

4.3 LDL

The Cholesky decomposition, in linear algebra, is a kind of matrix factorization. Matrix decompositions are used to deconstruct a matrix into a product of matrices.
A variant of the original Cholesky decomposition is the LDL decomposition, a factorization algorithm that involves two triangular matrices, L, and a diagonal matrix, D.
It is a perfect example code on which to apply the smart resource sharing algorithm since it is a complex and realistic design, that includes a huge number of hardware resources.

4.3.1 Units isolation and loop unrolling

The LDL cpp code is full of chained loops with numerous multiplications and subtractions. Therefore, there were several sections of the code that could have been optimized through sharing.
As a first attempt the middle loop has been unrolled, but the optimization did not result in significant improvements in the performances.
Therefore, the unrolling has been applied to the third and most internal loop that leads to the most relevant results in terms of area and time savings.

Since the loop mainly works with multiplications and subtractions the most worthwhile units to isolate and share are the newly constructed multiply and subtract units (multsub). The unrolling implemented on the loop is of the 8th order, thus the *multsub* number of units for each loop that can be optimized is 8.
Each newly written function must include a #pragma command that prevents the *inlining* in order to make the compiler consider each *multsub* as a separate functional unit: *#pragma HLS inline off*.

In this case, the operations are written in the .cpp file so that is the one that must be modified. In this first step, the 8 functions must be written with different names in order to be distinguished between one another, so they are called: *multsub1, multsub2, multsub3, multsub4, multsub5, multsub6, multsub7, multsub8*.

4.3.2 Synthesis without sharing and DFG analysis

In order to apply a smart resource sharing, it is necessary to know the position in which the multsub units would be placed after the synthesis of the original code.
Therefore, the unrolled code containing the isolated functions is imported in Vivado and synthesized on the *kintex7* FPGA *xcvu065-ffvc1517-1*. This FPGA has been chosen because it is the smallest one on which this circuit can be synthesized: it is composed of 600 DSP Slices, 358.080 CLB LUTs and 2520 18k RAM blocks.
In this case the code produces a very big design, so a suitable FPGA is required, anyway, a properly fitted FPGA is useful so that the amendments introduced through resource sharing may result in a more visible effect on its performances. This allows to have more relevant feedback to each change done in the resource sharing configuration.

In Vivado it is possible to carry out the synthesis on the chosen FPGA through a previously written script. After the synthesis, the DFG is generated and depicts a huge architecture with multiple octets of *multsub* units throughout its layout. In Figure 4.35 is shown the DFG of the LDL structure highlighting only the *multsub* octets.
The division of the units in octets is not accurate, the picture is only meant to give an idea of what the LDL DFG looks like. Despite the chaotic DFG,

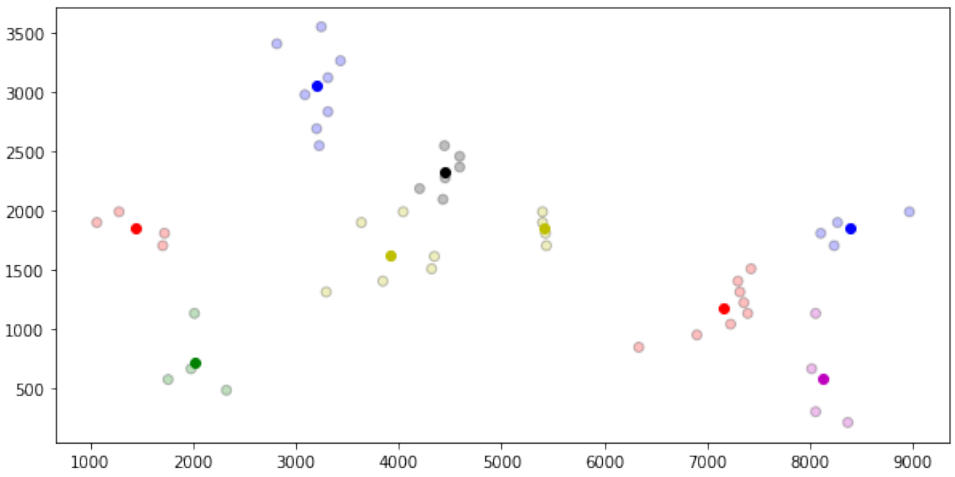


Figure 4.35. LDL DFG where only the *multsub* units are shown

it is possible to identify a recurring pattern of the 8 units. They are always placed in order, one below the other as shown in Figure 4.36.
Therefore it is possible to find an optimal locality clustering solution in case of k=4: 1-2, 3-4, 5-6, 7-8.

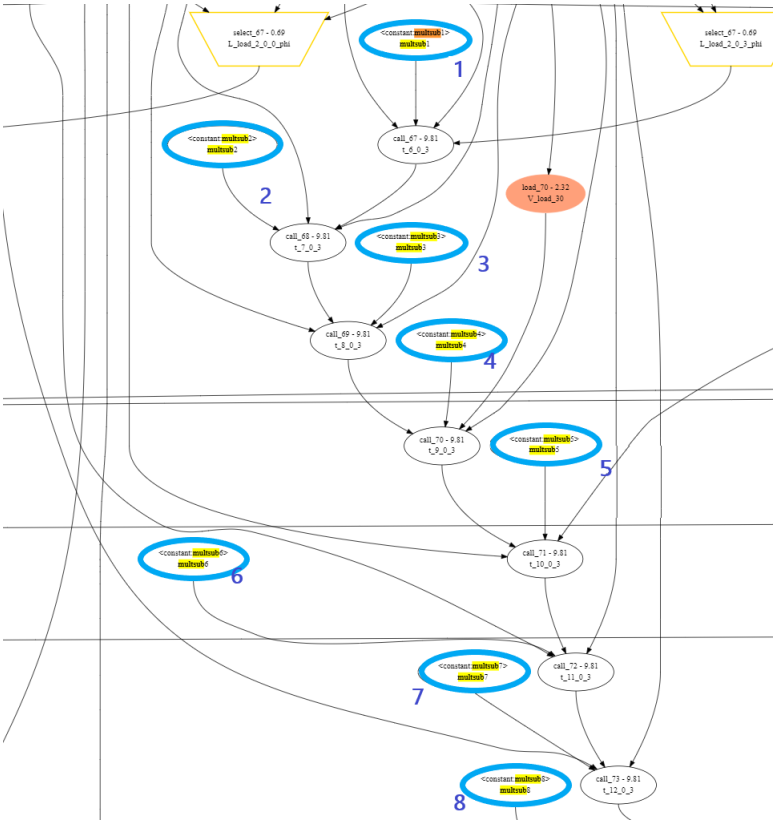


Figure 4.36. LDL DFG: closeup of a *multsub* units octet

4.3.3 Generation of new code files, Synthesis and Place & Route

Taking into account the optimum clustering solution obtained, it is possible to generate a new cpp file on the basis of the original one.
A Python code has been written to read the original cpp code file and to detect the functions called "sumsub" followed by a number. The name of each instance is replaced with a new name representing the cluster it has been assigned to.
Subsequently, a #pragma command is written to limit each new function to

only 1 instance. This command is inserted in order to simulate the sharing of one unit for each chosen group of function calls.
In the same way, as for the optimum solution, it is useful to create some random cluster solutions in order to assess the goodness of the optimal one. This is obtained shuffling the vector of the positions of the opt solution, it is an array where each position corresponds to one original unit and the content of each cell to the clusters to whom the unit belongs.
Afterwards, the random .cpp files can be printed modifying the function names using the same process employed for the optimal solution.

The new codes, optimal solutions and random ones, are imported in Vivado the synthesis and place & route are carried out. This process requires some hours to exploit the placement and routing of each project.

At the end of all of the simulations Vivado returns different report files from whom it is possible to get the information necessary to evaluate the resulting circuits:

- ldl_utilization_routed.rpt: the report file that contains information on the resources needed in order to synthesize a project, these quantities are expressed as a percentage of the used resources over the available ones.
  As previously outlined, the value used to assess the goodness of a solution is the maximum percentage of utilization of one kind of resource, since this is the fundamental limit for the synthesis on FPGAs.
- ldl_csynth file: the report file containing the number of clock cycles of the entire execution.
- ldl_timing_paths_routed: the report file where the delays of the 10 longest paths are illustrated.
  This value puts an upper bound to the frequency of the designed circuit.

4.3.4 Results comparison and behaviour analysis

Another Python code called "compare_results" is created to compare the computed circuit parameters and to generate new graphs to visually confront them.

All the report files from all the different solutions are read and the resulting parameters are extracted: the maximum percentage resource usage, the

maximum path delay and the total number of clock cycles.

Once all of the parameters have been extracted from the report files, it is possible to plot a *Delay-Area diagram*. The maximum data path delay is plotted into the y-axis. Vivado provides a minimum and a maximum latency value that has been used to generate two other versions of the graph taking plotting the total execution time on the y-axis but in this case, both values lead to the same graph proportionally.
The Resource Usage maximum percentage is plotted on the x-axis, this parameter quantifies how much the chip area increases or decreases with different sharing arrangements of the multsub units.
The resulting graph includes several points, one for each resource sharing arrangement, and they are marked with the name of the respective solution or with a number if related to a random solution.

Figure 4.37 and Table 4.8 show the Area-Delay diagram resulting from the unrolling of the second loop present in the LDL code A.6.

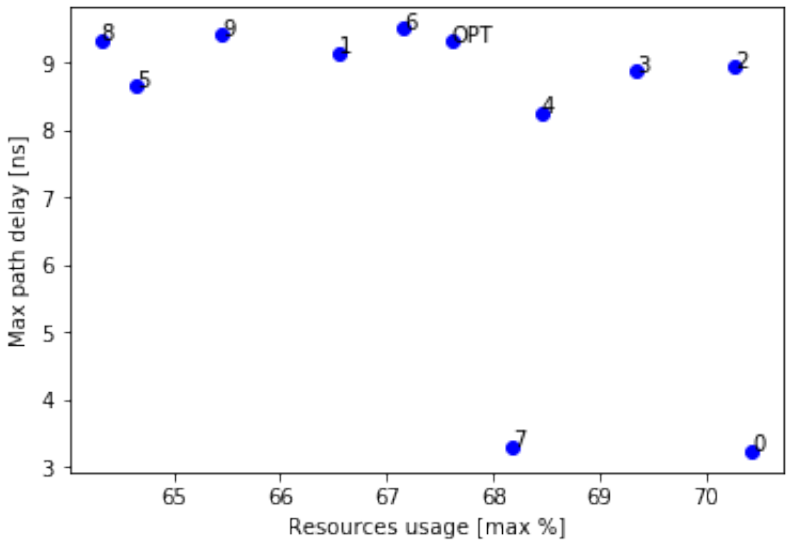


Figure 4.37. LDL unroll of the second loop - Delay-Area diagram k=4

Whereas, Figure 4.38 and Table 4.9 show the Area-Delay diagram resulting from the unrolling of the third and most internal loop present in the LDL code A.6.

	opt	0	1	2	3	4	5	6	7	8	9
T[ns]	9.317	3.237	9.143	8.94	8.879	8.261	8.657	9.515	3.308	9.348	9.415
res[%]	67.61	70.43	66.55	70.27	69.34	68.45	64.64	67.16	68.18	64.32	65.45

Table 4.8. LDL unroll of the 2nd loop - Maximum path delay [ns] and Maximum FPGA Resource Usage percentage [%] of different sharing solutions

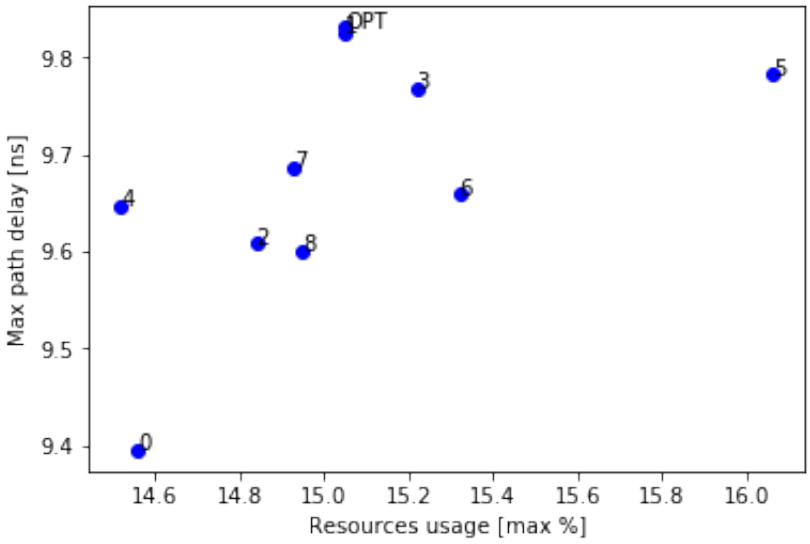


Figure 4.38. LDL unroll of the 3rd loop - Delay-Area diagram k=4

	OPT	0	1	2	3	4	5	6	7	8
T[ns]	9.831	9.395	9.826	9.608	9.768	9.646	9.782	9.659	9.687	9.601
res[%]	15.05	14.56	15.05	14.84	15.22	14.52	16.06	15.32	14.93	14.95

Table 4.9. LDL unroll of the 3rd loop - Maximum path delay [ns] and Maximum FPGA Resource Usage percentage [%] of different sharing solutions

4.3.5 Conclusions on LDL analysis

From the previous graphs, it is possible to draw some conclusions on the smart resource sharing employed on two different loops in the LDL code.

The configuration marked as "OPT" corresponds always to the ordered solution [1, 1, 2, 2, 3, 3, 4, 4] obtained through the proximity method assuming that the *multsub* octet is disposed in order as shown in Figure 4.36.
Nevertheless, this solution is not the one that minimizes the maximum path delay but it is still among the best solutions in terms of area occupation. This behaviour is explained by the fact that with such a large and complex Data Flow Graph it is impossible to predict accurately the disposition of each octet of units. The only feasible procedure was assuming that every octet is disposed in the same way: the octet taken under study is composed of units placed one below the other in order.

Whereas, exploring the third loop unroll solution in Code A.6, the "0" clustering solution stands out among the other for minimizing both the maximum path delay and area occupation.
The 0 random solution corresponds to the clustering arrangement: [3, 2, 1, 1, 2, 4, 4, 3] that does not reflect the optimal locality solution.

In conclusion, in this section, various sharing arrangements have been applied to 2 versions of the LDL code: unrolling 2 different loops of the code of order 8 and implementing the resource sharing to reduce each octet into 4 units.
The first simulations have been carried out unrolling the second and more external loop, while the second round of simulation involved the unrolling of the third and most internal loop, that better allowed to observe the changes in clustering arrangement into the final performance results.
Applying the smart sharing algorithm to a small section of the DFG obtained from the unroll of the third loop and repeating it for every octet of *multsub* unit, the results reveal that this does not produce better results in terms of path delay, it is closer to the minimum resource usage percentage found in this example.
The best configuration found is the [3, 2, 1, 1, 2, 4, 4, 3] configuration obtained randomly. The improvements that the "0" solution provides confronted with the supposed optimal locality solution are: 0,436 ns reduction of the maximum data path delay and only a 0,49% reduction of the resource usage maximum percentage.

Chapter 5

Conclusions

In the present work, an optimized resource sharing algorithm has been developed to make the employment of the resource sharing in an electronic system more efficient.
The smart sharing algorithm implemented is meant to generate the groups of operations that must be allocated to each remaining unit after the implementation of the resource sharing.
The method employed to optimally assign each operation to the respective shared unit is founded on the proximity method. The hypothesis is that a cluster of units placed closer to each other should be replaced by one shared unit, while units that are located further apart should use different resources. This can be reached examining the position in which the original operands were allocated before the reduction obtained through sharing and allocating each operation to the closest new operand.

The proximity based resource sharing is supposed to cause a greater improvement in the area reduction than a simple resource sharing since this method leads to shorter connections. The wires under discussion are the ones connecting the elements that used to interact with the original units that have been replaced with only one new one to be shared.
Using the smart sharing algorithm the new shared units are placed in proximity to all the elements belonging to the cluster that needs to reach it resulting in shorter wires.

A second hypothesized gain was that the proximity method would be able to limit the path delay increase, as well as resource usage. An increment in the

path delay is almost inevitable when implementing a resource sharing, nevertheless, the employment of shorter wire connections translates into quicker communications and therefore in shorter delays.

The proximity based resource sharing algorithm presented in this work gives positive results applied to proposed example codes: FIR filter, KNP and LDL codes. In particular, the simulations carried out on the FIR filter, KNP codes produced performance data that can validate the theorized proximity based resource sharing method.

On the basis of this outcome, the main result that has been observed is that, as hypothesized, the theory of locality gives positive results in almost every circumstance in terms of decreasing the number of necessary resources. This means that the proximity based resource sharing results in a greater improvement in the area reduction than a simple resource sharing. This is due to the shorter connections generated by this smart allocation of resources: the elements that used to interact with the original units are forced to interact with the closest new shared resource.
As a result, the wires are kept short and do not cross the entire chip to reach the new shared unit.

On the other hand, the simulations did not produce coherent results on the evolution of the maximum path delay with the employment of the smart resource sharing. Therefore, it is not possible to draw a model of the relationship between locality and decreasing of the delay since some results prove that it tends to decrease with the increment of the clustering cost.
This outcome does not affect the goodness of the smart sharing algorithm since the path delay is expected to grow at any rate when resource sharing is employed in an electronic system.

In the perspective of further improvements, the Proximity Based Resource Sharing Algorithm could be fully automatized and optimized even for bigger and more complex data flow graphs where the units to be shared are not easily detectable.
The possible applications are countless, since this resource sharing upgrade can be implemented on any hardware description code. In particular, the most effective outcome would probably be obtained implementing the proximity based algorithm on any floating point application, since they involve time and area consuming units whose number needs to be reduced in a smart

way.
For instance, finite elements models simulations, neural network training or financial simulations.

Bibliography

[1] Lloyd, Stuart P. *Least squares quantization in PCM*, Information Theory, IEEE Transactions on 28.2 (1982): 129-137.

[2] A. Trevino *Introduction to K-means Clustering*, Oracle Data Science Blog, URL: <https://blogs.oracle.com/datascience/introduction-to-k-means-clustering>, December 2016

[3] R. Kastner and S. Neuendorffer, *Parallel Programming for FPGAs*, 24th August, 2017.

[4] D. Bufistov, J. Cortadella, M. Kishinevsky and S. Sapatnekar, *A general model for performance optimization of sequential systems*, ICCAD '07 Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design, pages 362-369, San Jose, California — November 05 - 08, 2007

[5] Vivado High-Level Synthesis, *Vivado High-Level Synthesis*, URL: <https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html>

[6] Xilinx, *Vivado Design Suite User Guide - High-Level Synthesis*, URL: <https://www.xilinx.com/>, UG902 (v2017.4) February 2, 2018

[7] Xilinx, *Vivado Design Suite User Guide - Release Notes, Installation, and Licensing*, URL: <https://www.xilinx.com/>, UG973 (v2019.1) June 7, 2019

[8] Xilinx, Tom Feist, *Vivado Design Suite*, URL: <https://www.xilinx.com/>, WP416 (v1.1) June 22, 2012

[9] IEEE Xplore, *High-Level Synthesis: Past, Present, and Future*, IEEE Design & Test of Computers (Volume: 26, Issue: 4, July-Aug. 2009)

[10] S. Hadjis, A. Canis, J.H. Anderson, J. Choi, K. Nam, S. Brown, T. Czajkowski, *Impact of FPGA architecture on resource sharing in high-level synthesis*, FPGA '12 Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays, Pages 111-114, Monterey, California, USA — February 22 - 24, 2012

[11] IEEE, *Enabling High-Level Synthesis Resource Sharing Design Space Exploration in FPGAs Through Automatic Internal Bitwidth Adjustments*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (Volume: 36, Issue: 1, Jan. 2017)

[12] D. Wilson, A. Shastri, G. Stitt1, *A High-Level Synthesis Scheduling and Binding Heuristic for FPGA Fault Tolerance*, International Journal of Reconfigurable Computing Volume 2017, Article ID 5419767, 17 pages

[13] B. Carrión Schäfer, *Enabling High-Level Synthesis Resource Sharing Design Space Exploration in FPGAs Through Automatic Internal Bitwidth Adjustments*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 36(1):1-1, January 2016

[14] B. Liebig, A. Koch *High-level synthesis of resource-shared microarchitectures from irregular complex C-code*, Conference Paper, December 2016

[15] Graphviz - Graph Visualization Software *Graphviz - Graph Visualization Software*, 2001, `URL: <https://www.graphviz.org/>`

[16] M. Simionato *An Introduction to Graphviz and dot*, `URL: <http://www.linuxdevcenter.com/pub/a/linux/2004/05/06/graphviz_dot.html>`, June 2004

Appendices

Appendix A

C++ example codes

A.1 cpp_fir.h

```
/************************************************************
Vendor: Xilinx
Associated Filename: cpp_FIR.h
Purpose:Vivado HLS Coding Style example
Device: All
Revision History: May 30, 2008 - initial release

************************************************************
#- (c) Copyright 2011-2018 Xilinx, Inc. All rights reserved.
#-
#- This file contains confidential and proprietary
   information
#- of Xilinx, Inc. and is protected under U.S. and
#- international copyright and other intellectual property
#- laws.
#-
#- DISCLAIMER
#- This disclaimer is not a license and does not grant any
#- rights to the materials distributed herewith. Except as
#- otherwise provided in a valid license issued to you by
#- Xilinx, and to the maximum extent permitted by applicable
#- law: (1) THESE MATERIALS ARE MADE AVAILABLE "AS IS" AND
#- WITH ALL FAULTS, AND XILINX HEREBY DISCLAIMS ALL
   WARRANTIES
```

```
#- AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING
#- BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-
#- INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and
#- (2) Xilinx shall not be liable (whether in contract or
   tort,
#- including negligence, or under any other theory of
#- liability) for any loss or damage of any kind or nature
#- related to, arising under or in connection with these
#- materials, including for any direct, or any indirect,
#- special, incidental, or consequential loss or damage
#- (including loss of data, profits, goodwill, or any type of
#- loss or damage suffered as a result of any action brought
#- by a third party) even if such damage or loss was
#- reasonably foreseeable or Xilinx had been advised of the
#- possibility of the same.
#-
#- CRITICAL APPLICATIONS
#- Xilinx products are not designed or intended to be fail-
#- safe, or for use in any application requiring fail-safe
#- performance, such as life-support or safety devices or
#- systems, Class III medical devices, nuclear facilities,
#- applications related to the deployment of airbags, or any
#- other applications that could lead to death, personal
#- injury, or severe property or environmental damage
#- (individually and collectively, "Critical
#- Applications"). Customer assumes the sole risk and
#- liability of any use of Xilinx products in Critical
#- Applications, subject only to applicable laws and
#- regulations governing limitations on product liability.
#-
#- THIS COPYRIGHT NOTICE AND DISCLAIMER MUST BE RETAINED AS
#- PART OF THIS FILE AT ALL TIMES.
#- ************************************************


This file contains confidential and proprietary information
   of Xilinx, Inc. and
is protected under U.S. and international copyright and other
    intellectual
```

```
property laws.

DISCLAIMER
This disclaimer is not a license and does not grant any
   rights to the materials
distributed herewith. Except as otherwise provided in a valid
    license issued to
you by Xilinx, and to the maximum extent permitted by
   applicable law:
(1) THESE MATERIALS ARE MADE AVAILABLE "AS IS" AND WITH ALL
   FAULTS, AND XILINX
HEREBY DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS,
   IMPLIED, OR STATUTORY,
INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY,
   NON-INFRINGEMENT, OR
FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not
   be liable (whether
in contract or tort, including negligence, or under any other
    theory of
liability) for any loss or damage of any kind or nature
   related to, arising under
or in connection with these materials, including for any
   direct, or any indirect,
special, incidental, or consequential loss or damage (
   including loss of data,
profits, goodwill, or any type of loss or damage suffered as
   a result of any
action brought by a third party) even if such damage or loss
   was reasonably
foreseeable or Xilinx had been advised of the possibility of
   the same.

CRITICAL APPLICATIONS
Xilinx products are not designed or intended to be fail-safe,
    or for use in any
application requiring fail-safe performance, such as life-
   support or safety
devices or systems, Class III medical devices, nuclear
   facilities, applications
```

```
related to the deployment of airbags, or any other
   applications that could lead
to death, personal injury, or severe property or
   environmental damage
(individually and collectively, "Critical Applications").
   Customer assumes the
sole risk and liability of any use of Xilinx products in
   Critical Applications,
subject only to applicable laws and regulations governing
   limitations on product
liability.

THIS COPYRIGHT NOTICE AND DISCLAIMER MUST BE RETAINED AS PART
    OF THIS FILE AT
ALL TIMES.

**********************************************************/
#ifndef _CPP_FIR_H_
#define _CPP_FIR_H_

#include <fstream>
#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;

#define N 85

typedef int coef_t;
typedef int data_t;
typedef int acc_t;

acc_t mac_new0 (acc_t ac, data_t m, coef_t c);
acc_t mac_new1 (acc_t ac, data_t m, coef_t c);
acc_t mac_new2 (acc_t ac, data_t m, coef_t c);
acc_t mac_new3 (acc_t ac, data_t m, coef_t c);
acc_t mac_new4 (acc_t ac, data_t m, coef_t c);
acc_t mac_new5 (acc_t ac, data_t m, coef_t c);
```

```
// Class CFir definition
template<class coef_T, class data_T, class acc_T>
class CFir {
protected:
  static const coef_T c[N];
  data_T shift_reg[N-1];
private:
  //acc_t macx1 (acc_t ac, data_t m, coef_t c);
public:
  data_T operator()(data_T x);
  template<class coef_TT, class data_TT, class acc_TT>
  friend ostream&
  operator<<(ostream& o, const CFir<coef_TT, data_TT, acc_TT> &
     f);

};


// Load FIR coefficients
template<class coef_T, class data_T, class acc_T>
const coef_T CFir<coef_T, data_T, acc_T>::c[N] = {
       #include "cpp_FIR.inc"
};

// FIR main algorithm
template<class coef_T, class data_T, class acc_T>
data_T CFir<coef_T, data_T, acc_T>::operator()(data_T x) {


    int i;
    acc_t acc = 0;
    data_t m,m1,m2,m3,m4,m5;

    loop: for (i = N-1; i >= 0; i-=6) {
        if (i == 0) {
          m = x;
          shift_reg[0] = x;
        } else {
          m = shift_reg[i-1];
```

```
        m1 = shift_reg[i-2];
        m2 = shift_reg[i-3];
        m3 = shift_reg[i-4];
        m4 = shift_reg[i-5];
        m5 = shift_reg[i-6];
        if (i != (N-1))
          shift_reg[i] = shift_reg[i - 1];
          shift_reg[i-1] = shift_reg[i - 2];
          shift_reg[i-2] = shift_reg[i - 3];
          shift_reg[i-3] = shift_reg[i - 4];
          shift_reg[i-4] = shift_reg[i - 5];
          shift_reg[i-5] = shift_reg[i - 6];
      }
      acc= mac_new0(acc,m,c[i]);
      acc= mac_new1(acc,m1,c[i-1]);
      acc= mac_new2(acc,m2,c[i-2]);
   acc= mac_new3(acc,m3,c[i-3]);
      acc= mac_new4(acc,m4,c[i-4]);
   acc= mac_new5(acc,m5,c[i-5]);
}

   return acc;
}



// Operator for displaying results
template<class coef_T, class data_T, class acc_T>
ostream& operator<<(ostream& o, const CFir<coef_T,
 data_T, acc_T> &f) {
   for (int i = 0; i < (sizeof(f.shift_reg)/sizeof(data_T)); i
      ++)
     {
       o << "shift_reg[" << i << "]=␣" << f.shift_reg[i] <<
          endl;
     }
   o << "_______________" << endl;
   return o;
}
```

```
data_t cpp_FIR(data_t x);

#endif
```

A.2 cpp_fir.cpp

```
/************************************************************
Vendor: Xilinx
Associated Filename: cpp_FIR.cpp
Purpose:Vivado HLS Coding Style example
Device: All
Revision History: May 30, 2008 - initial release

************************************************************
#- (c) Copyright 2011-2018 Xilinx, Inc. All rights reserved.
#-
#- This file contains confidential and proprietary
   information
#- of Xilinx, Inc. and is protected under U.S. and
#- international copyright and other intellectual property
#- laws.
#-
#- DISCLAIMER
#- This disclaimer is not a license and does not grant any
#- rights to the materials distributed herewith. Except as
#- otherwise provided in a valid license issued to you by
#- Xilinx, and to the maximum extent permitted by applicable
#- law: (1) THESE MATERIALS ARE MADE AVAILABLE "AS IS" AND
#- WITH ALL FAULTS, AND XILINX HEREBY DISCLAIMS ALL
   WARRANTIES
#- AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING
#- BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-
#- INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and
#- (2) Xilinx shall not be liable (whether in contract or
   tort,
#- including negligence, or under any other theory of
```

```
#- liability) for any loss or damage of any kind or nature
#- related to, arising under or in connection with these
#- materials, including for any direct, or any indirect,
#- special, incidental, or consequential loss or damage
#- (including loss of data, profits, goodwill, or any type of
#- loss or damage suffered as a result of any action brought
#- by a third party) even if such damage or loss was
#- reasonably foreseeable or Xilinx had been advised of the
#- possibility of the same.
#-
#- CRITICAL APPLICATIONS
#- Xilinx products are not designed or intended to be fail-
#- safe, or for use in any application requiring fail-safe
#- performance, such as life-support or safety devices or
#- systems, Class III medical devices, nuclear facilities,
#- applications related to the deployment of airbags, or any
#- other applications that could lead to death, personal
#- injury, or severe property or environmental damage
#- (individually and collectively, "Critical
#- Applications"). Customer assumes the sole risk and
#- liability of any use of Xilinx products in Critical
#- Applications, subject only to applicable laws and
#- regulations governing limitations on product liability.
#-
#- THIS COPYRIGHT NOTICE AND DISCLAIMER MUST BE RETAINED AS
#- PART OF THIS FILE AT ALL TIMES.
#- ************************************************


This file contains confidential and proprietary information
   of Xilinx, Inc. and
is protected under U.S. and international copyright and other
    intellectual
property laws.

DISCLAIMER
This disclaimer is not a license and does not grant any
   rights to the materials
```

```
distributed herewith. Except as otherwise provided in a valid
    license issued to
you by Xilinx, and to the maximum extent permitted by
   applicable law:
(1) THESE MATERIALS ARE MADE AVAILABLE "AS IS" AND WITH ALL
   FAULTS, AND XILINX
HEREBY DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS,
   IMPLIED, OR STATUTORY,
INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY,
   NON-INFRINGEMENT, OR
FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not
   be liable (whether
in contract or tort, including negligence, or under any other
    theory of
liability) for any loss or damage of any kind or nature
   related to, arising under
or in connection with these materials, including for any
   direct, or any indirect,
special, incidental, or consequential loss or damage (
   including loss of data,
profits, goodwill, or any type of loss or damage suffered as
   a result of any
action brought by a third party) even if such damage or loss
   was reasonably
foreseeable or Xilinx had been advised of the possibility of
   the same.

CRITICAL APPLICATIONS
Xilinx products are not designed or intended to be fail-safe,
    or for use in any
application requiring fail-safe performance, such as life-
   support or safety
devices or systems, Class III medical devices, nuclear
   facilities, applications
related to the deployment of airbags, or any other
   applications that could lead
to death, personal injury, or severe property or
   environmental damage
```

```
(individually and collectively, "Critical Applications").
   Customer assumes the
sole risk and liability of any use of Xilinx products in
   Critical Applications,
subject only to applicable laws and regulations governing
   limitations on product
liability.

THIS COPYRIGHT NOTICE AND DISCLAIMER MUST BE RETAINED AS PART
    OF THIS FILE AT
ALL TIMES.

*********************************************************/
#include "cpp_FIR.h"

// Top-level function with class instantiated
data_t cpp_FIR(data_t x)
  {
    static CFir<coef_t, data_t, acc_t> fir1;

    //cout << fir1;

    return fir1(x);
  }

acc_t mac_new0 (acc_t ac, data_t m, coef_t c){
#pragma HLS inline off
  return ac+m*c;}
acc_t mac_new1 (acc_t ac, data_t m, coef_t c){
#pragma HLS inline off
  return ac+m*c;}
acc_t mac_new2 (acc_t ac, data_t m, coef_t c){
#pragma HLS inline off
  return ac+m*c;}
acc_t mac3 (acc_t ac, data_t m, coef_t c){
#pragma HLS inline off
  return ac+m*c;}
acc_t mac4 (acc_t ac, data_t m, coef_t c){
#pragma HLS inline off
```

```
  return ac+m*c;}
acc_t mac5 (acc_t ac, data_t m, coef_t c){
#pragma HLS inline off
  return ac+m*c;}
```

A.3 knp.h

```
/*************************************************************
Vendor: Xilinx
Associated Filename: knp.h
Purpose:Vivado HLS Coding Style example
Device: All
Revision History: May 30, 2008 - initial release

*************************************************************
#- (c) Copyright 2011-2018 Xilinx, Inc. All rights reserved.
#-
#- This file contains confidential and proprietary
   information
#- of Xilinx, Inc. and is protected under U.S. and
#- international copyright and other intellectual property
#- laws.
#-
#- DISCLAIMER
#- This disclaimer is not a license and does not grant any
#- rights to the materials distributed herewith. Except as
#- otherwise provided in a valid license issued to you by
#- Xilinx, and to the maximum extent permitted by applicable
#- law: (1) THESE MATERIALS ARE MADE AVAILABLE "AS IS" AND
#- WITH ALL FAULTS, AND XILINX HEREBY DISCLAIMS ALL
   WARRANTIES
#- AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING
#- BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-
#- INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and
#- (2) Xilinx shall not be liable (whether in contract or
   tort,
#- including negligence, or under any other theory of
```

```
#- liability) for any loss or damage of any kind or nature
#- related to, arising under or in connection with these
#- materials, including for any direct, or any indirect,
#- special, incidental, or consequential loss or damage
#- (including loss of data, profits, goodwill, or any type of
#- loss or damage suffered as a result of any action brought
#- by a third party) even if such damage or loss was
#- reasonably foreseeable or Xilinx had been advised of the
#- possibility of the same.
#-
#- CRITICAL APPLICATIONS
#- Xilinx products are not designed or intended to be fail-
#- safe, or for use in any application requiring fail-safe
#- performance, such as life-support or safety devices or
#- systems, Class III medical devices, nuclear facilities,
#- applications related to the deployment of airbags, or any
#- other applications that could lead to death, personal
#- injury, or severe property or environmental damage
#- (individually and collectively, "Critical
#- Applications"). Customer assumes the sole risk and
#- liability of any use of Xilinx products in Critical
#- Applications, subject only to applicable laws and
#- regulations governing limitations on product liability.
#-
#- THIS COPYRIGHT NOTICE AND DISCLAIMER MUST BE RETAINED AS
#- PART OF THIS FILE AT ALL TIMES.
#- ****************************************************


This file contains confidential and proprietary information
   of Xilinx, Inc. and
is protected under U.S. and international copyright and other
    intellectual
property laws.

DISCLAIMER
This disclaimer is not a license and does not grant any
   rights to the materials
```

```
distributed herewith. Except as otherwise provided in a valid
    license issued to
you by Xilinx, and to the maximum extent permitted by
   applicable law:
(1) THESE MATERIALS ARE MADE AVAILABLE "AS IS" AND WITH ALL
   FAULTS, AND XILINX
HEREBY DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS,
   IMPLIED, OR STATUTORY,
INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY,
   NON-INFRINGEMENT, OR
FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not
   be liable (whether
in contract or tort, including negligence, or under any other
    theory of
liability) for any loss or damage of any kind or nature
   related to, arising under
or in connection with these materials, including for any
   direct, or any indirect,
special, incidental, or consequential loss or damage (
   including loss of data,
profits, goodwill, or any type of loss or damage suffered as
   a result of any
action brought by a third party) even if such damage or loss
   was reasonably
foreseeable or Xilinx had been advised of the possibility of
   the same.

CRITICAL APPLICATIONS
Xilinx products are not designed or intended to be fail-safe,
    or for use in any
application requiring fail-safe performance, such as life-
   support or safety
devices or systems, Class III medical devices, nuclear
   facilities, applications
related to the deployment of airbags, or any other
   applications that could lead
to death, personal injury, or severe property or
   environmental damage
```

```
(individually and collectively, "Critical Applications").
   Customer assumes the
sole risk and liability of any use of Xilinx products in
   Critical Applications,
subject only to applicable laws and regulations governing
   limitations on product
liability.

THIS COPYRIGHT NOTICE AND DISCLAIMER MUST BE RETAINED AS PART
    OF THIS FILE AT
ALL TIMES.

**********************************************************/
#pragma once

#include <math.h>
#include <stdio.h>
#include "param.h"


//2.0 #define of constant variables
#ifndef M_PI
#define M_PI 3.1415926535f
#endif
// Flow vector scaling factor
#define FLOW_SCALING_FACTOR (0.25f) //(1.0f/4.0f)

typedef int mType;



extern "C" void knp(
              unsigned char * im1,
              unsigned char * im2,
              float * out
              );

  float fmac0(float a, float b, float c);
  float fmac1(float a, float b, float c);
```

```
float fmac2(float a, float b, float c);
float fmac3(float a, float b, float c);
float fmac4(float a, float b, float c);
float fmac5(float a, float b, float c);
int imac6(int a, int b, int c);
int imac7(int a, int b, int c);
int imac8(int a, int b, int c);
int imac9(int a, int b, int c);
int imac10(int a, int b, int c);
```

A.4 knp_mod.cpp

```
/************************************************************
Vendor: Xilinx
Associated Filename: knp_mod.cpp
Purpose:Vivado HLS Coding Style example
Device: All
Revision History: May 30, 2008 - initial release

************************************************************
#- (c) Copyright 2011-2018 Xilinx, Inc. All rights reserved.
#-
#- This file contains confidential and proprietary
   information
#- of Xilinx, Inc. and is protected under U.S. and
#- international copyright and other intellectual property
#- laws.
#-
#- DISCLAIMER
#- This disclaimer is not a license and does not grant any
#- rights to the materials distributed herewith. Except as
#- otherwise provided in a valid license issued to you by
#- Xilinx, and to the maximum extent permitted by applicable
#- law: (1) THESE MATERIALS ARE MADE AVAILABLE "AS IS" AND
#- WITH ALL FAULTS, AND XILINX HEREBY DISCLAIMS ALL
   WARRANTIES
#- AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING
```

```
#- BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-
#- INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and
#- (2) Xilinx shall not be liable (whether in contract or
   tort,
#- including negligence, or under any other theory of
#- liability) for any loss or damage of any kind or nature
#- related to, arising under or in connection with these
#- materials, including for any direct, or any indirect,
#- special, incidental, or consequential loss or damage
#- (including loss of data, profits, goodwill, or any type of
#- loss or damage suffered as a result of any action brought
#- by a third party) even if such damage or loss was
#- reasonably foreseeable or Xilinx had been advised of the
#- possibility of the same.
#-
#- CRITICAL APPLICATIONS
#- Xilinx products are not designed or intended to be fail-
#- safe, or for use in any application requiring fail-safe
#- performance, such as life-support or safety devices or
#- systems, Class III medical devices, nuclear facilities,
#- applications related to the deployment of airbags, or any
#- other applications that could lead to death, personal
#- injury, or severe property or environmental damage
#- (individually and collectively, "Critical
#- Applications"). Customer assumes the sole risk and
#- liability of any use of Xilinx products in Critical
#- Applications, subject only to applicable laws and
#- regulations governing limitations on product liability.
#-
#- THIS COPYRIGHT NOTICE AND DISCLAIMER MUST BE RETAINED AS
#- PART OF THIS FILE AT ALL TIMES.
#- ****************************************************


This file contains confidential and proprietary information
   of Xilinx, Inc. and
is protected under U.S. and international copyright and other
    intellectual
property laws.
```

```
DISCLAIMER
This disclaimer is not a license and does not grant any
   rights to the materials
distributed herewith. Except as otherwise provided in a valid
    license issued to
you by Xilinx, and to the maximum extent permitted by
   applicable law:
(1) THESE MATERIALS ARE MADE AVAILABLE "AS IS" AND WITH ALL
   FAULTS, AND XILINX
HEREBY DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS,
   IMPLIED, OR STATUTORY,
INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY,
   NON-INFRINGEMENT, OR
FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not
   be liable (whether
in contract or tort, including negligence, or under any other
    theory of
liability) for any loss or damage of any kind or nature
   related to, arising under
or in connection with these materials, including for any
   direct, or any indirect,
special, incidental, or consequential loss or damage (
   including loss of data,
profits, goodwill, or any type of loss or damage suffered as
   a result of any
action brought by a third party) even if such damage or loss
   was reasonably
foreseeable or Xilinx had been advised of the possibility of
   the same.

CRITICAL APPLICATIONS
Xilinx products are not designed or intended to be fail-safe,
    or for use in any
application requiring fail-safe performance, such as life-
   support or safety
devices or systems, Class III medical devices, nuclear
   facilities, applications
```

```
related to the deployment of airbags, or any other
   applications that could lead
to death, personal injury, or severe property or
   environmental damage
(individually and collectively, "Critical Applications").
   Customer assumes the
sole risk and liability of any use of Xilinx products in
   Critical Applications,
subject only to applicable laws and regulations governing
   limitations on product
liability.

THIS COPYRIGHT NOTICE AND DISCLAIMER MUST BE RETAINED AS PART
    OF THIS FILE AT
ALL TIMES.

**********************************************************/
#include "knp.h"

int PO(int x, int y){
       return x + y * WIDTH;
}
int Px(int x){
#pragma HLS INLINE
       if(x >=WIDTH)
              x = WIDTH -1;
       else if (x < 0)
              x = 0;
       return x;
}
int Py(int y){
#pragma HLS INLINE
       if(y >=HEIGHT)
              y = HEIGHT - 1;
       else if (y < 0)
              y = 0;
       return y;
}
int P(int x, int y){
```

```
#pragma HLS INLINE
	return Py(y) * WIDTH + Px(x);
}
int get_matrix_inv(mType* G, float* G_inv){
	float detG = (float)G[0] * G[3] - (float)G[1] * G[2];
	if (detG <= 1.0f) { return 0; }
	float detG_inv = 1.0f / detG;
	G_inv[0] = G[3] * detG_inv;
	G_inv[1] = -G[1] * detG_inv;
	G_inv[2] = -G[2] * detG_inv;
	G_inv[3] = G[0] * detG_inv;
	return 1;
}
extern "C"
void knp(
		unsigned char * im1,
		unsigned char * im2,
		float* out
		)
{
#pragma HLS INTERFACE m_axi port=im1 offset=slave bundle=gmem0
#pragma HLS INTERFACE m_axi port=im2 offset=slave bundle=gmem1
#pragma HLS INTERFACE m_axi port=out offset=slave bundle=gmem2

#pragma HLS INTERFACE s_axilite port=im1 bundle=control
#pragma HLS INTERFACE s_axilite port=im2 bundle=control
#pragma HLS INTERFACE s_axilite port=out bundle=control
#pragma HLS INTERFACE s_axilite port=return bundle=control

loop_main:for(int j=0;j<HEIGHT;j++)
	for(int i = 0; i<WIDTH;i++){
		float G_inv[4] = { 0.0f, 0.0f, 0.0f, 0.0f };
#pragma HLS ARRAY_PARTITION variable=G_inv complete dim=1
		mType G[4] = { 0.0f, 0.0f, 0.0f, 0.0f };
#pragma HLS ARRAY_PARTITION variable=G complete dim=1
		mType b_k[2] = { 0.0f, 0.0f };
#pragma HLS ARRAY_PARTITION variable=b_k complete dim=1

loop_column:for (int wj = -WINDOW_SIZE; wj<=WINDOW_SIZE;wj++)
```

```
        {
loop_row: for (int wi = -WINDOW_SIZE; wi<=WINDOW_SIZE;wi++)
        {

        int px = Px(i+wi), py = Py(j+wj);
        int temp6=imac6(py, WIDTH, px);
        int im2_val = im2[temp6];

        int deltaIk = im1[temp6] - im2_val;
        int a=Px(i + wi +1), b=Px(i + wi -1);
        int cx=imac7(py, WIDTH,a), dx = imac8(py, WIDTH,b);
        int cIx=im1[cx];

        cIx-= im1[dx];
        cIx >>= 1;
        int c=Py(j + wj +1), d=Py(j + wj -1);
        int cy=imac9(WIDTH,c,px), dy=imac10(WIDTH,d,px);

        int cIy =im1[cy];
        cIy-= im1[dy];
        cIy >>=1;

        G[0] = fmac0(cIx, cIx, G[0]);
        G[1] = fmac1(cIx, cIy, G[1]);
        G[2] = fmac2(cIx, cIy, G[2]);
        G[3] = fmac3(cIy, cIy, G[3]);
        b_k[0] = fmac4(deltaIk, cIx, b_k[0]);
        b_k[1] = fmac5(deltaIk, cIy, b_k[1]);
        }
}
        get_matrix_inv(G, G_inv);
        float fx = 0.0f, fy = 0.0f;
        fx = G_inv[0] * b_k[0] + G_inv[1] * b_k[1];
        fy = G_inv[2] * b_k[0] + G_inv[3] * b_k[1];

        out[2*(j*WIDTH+i)]=fx;
        out[2*(j*WIDTH+i)+1]=fy;
        }
}
```

A.5 ldl_top.h

```
/*********************************************************
Vendor: Xilinx
Associated Filename: ldl_top.h
Purpose:Vivado HLS Coding Style example
Device: All
Revision History: May 30, 2008 - initial release
*********************************************************
#- (c) Copyright 2011-2018 Xilinx, Inc. All rights reserved.
#-
#- This file contains confidential and proprietary
   information
#- of Xilinx, Inc. and is protected under U.S. and
#- international copyright and other intellectual property
#- laws.
#-
#- DISCLAIMER
#- This disclaimer is not a license and does not grant any
#- rights to the materials distributed herewith. Except as
#- otherwise provided in a valid license issued to you by
#- Xilinx, and to the maximum extent permitted by applicable
#- law: (1) THESE MATERIALS ARE MADE AVAILABLE "AS IS" AND
#- WITH ALL FAULTS, AND XILINX HEREBY DISCLAIMS ALL
   WARRANTIES
#- AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING
#- BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-
#- INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and
#- (2) Xilinx shall not be liable (whether in contract or
   tort,
#- including negligence, or under any other theory of
#- liability) for any loss or damage of any kind or nature
#- related to, arising under or in connection with these
#- materials, including for any direct, or any indirect,
#- special, incidental, or consequential loss or damage
#- (including loss of data, profits, goodwill, or any type of
```

```
#- loss or damage suffered as a result of any action brought
#- by a third party) even if such damage or loss was
#- reasonably foreseeable or Xilinx had been advised of the
#- possibility of the same.
#-
#- CRITICAL APPLICATIONS
#- Xilinx products are not designed or intended to be fail-
#- safe, or for use in any application requiring fail-safe
#- performance, such as life-support or safety devices or
#- systems, Class III medical devices, nuclear facilities,
#- applications related to the deployment of airbags, or any
#- other applications that could lead to death, personal
#- injury, or severe property or environmental damage
#- (individually and collectively, "Critical
#- Applications"). Customer assumes the sole risk and
#- liability of any use of Xilinx products in Critical
#- Applications, subject only to applicable laws and
#- regulations governing limitations on product liability.
#-
#- THIS COPYRIGHT NOTICE AND DISCLAIMER MUST BE RETAINED AS
#- PART OF THIS FILE AT ALL TIMES.
#- ************************************************


This file contains confidential and proprietary information
   of Xilinx, Inc. and
is protected under U.S. and international copyright and other
    intellectual
property laws.

DISCLAIMER
This disclaimer is not a license and does not grant any
   rights to the materials
distributed herewith. Except as otherwise provided in a valid
    license issued to
you by Xilinx, and to the maximum extent permitted by
   applicable law:
(1) THESE MATERIALS ARE MADE AVAILABLE "AS IS" AND WITH ALL
   FAULTS, AND XILINX
```

```
HEREBY DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS,
   IMPLIED, OR STATUTORY,
INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY,
   NON-INFRINGEMENT, OR
FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not
   be liable (whether
in contract or tort, including negligence, or under any other
    theory of
liability) for any loss or damage of any kind or nature
   related to, arising under
or in connection with these materials, including for any
   direct, or any indirect,
special, incidental, or consequential loss or damage (
   including loss of data,
profits, goodwill, or any type of loss or damage suffered as
   a result of any
action brought by a third party) even if such damage or loss
   was reasonably
foreseeable or Xilinx had been advised of the possibility of
   the same.

CRITICAL APPLICATIONS
Xilinx products are not designed or intended to be fail-safe,
    or for use in any
application requiring fail-safe performance, such as life-
   support or safety
devices or systems, Class III medical devices, nuclear
   facilities, applications
related to the deployment of airbags, or any other
   applications that could lead
to death, personal injury, or severe property or
   environmental damage
(individually and collectively, "Critical Applications").
   Customer assumes the
sole risk and liability of any use of Xilinx products in
   Critical Applications,
subject only to applicable laws and regulations governing
   limitations on product
liability.
```

```
THIS COPYRIGHT NOTICE AND DISCLAIMER MUST BE RETAINED AS PART
    OF THIS FILE AT
ALL TIMES.

*********************************************************/
#define N 32

void ldl_top(float V[N], float L[N][N], float D[N], float A[N
   ][N]);
```

A.6 ldl_top.cpp

```
/*********************************************************
Vendor: Xilinx
Associated Filename: ldl_top.cpp
Purpose:Vivado HLS Coding Style example
Device: All
Revision History: May 30, 2008 - initial release

*********************************************************
#- (c) Copyright 2011-2018 Xilinx, Inc. All rights reserved.
#-
#- This file contains confidential and proprietary
   information
#- of Xilinx, Inc. and is protected under U.S. and
#- international copyright and other intellectual property
#- laws.
#-
#- DISCLAIMER
#- This disclaimer is not a license and does not grant any
#- rights to the materials distributed herewith. Except as
#- otherwise provided in a valid license issued to you by
#- Xilinx, and to the maximum extent permitted by applicable
#- law: (1) THESE MATERIALS ARE MADE AVAILABLE "AS IS" AND
#- WITH ALL FAULTS, AND XILINX HEREBY DISCLAIMS ALL
   WARRANTIES
```

```
#- AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING
#- BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-
#- INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and
#- (2) Xilinx shall not be liable (whether in contract or
   tort,
#- including negligence, or under any other theory of
#- liability) for any loss or damage of any kind or nature
#- related to, arising under or in connection with these
#- materials, including for any direct, or any indirect,
#- special, incidental, or consequential loss or damage
#- (including loss of data, profits, goodwill, or any type of
#- loss or damage suffered as a result of any action brought
#- by a third party) even if such damage or loss was
#- reasonably foreseeable or Xilinx had been advised of the
#- possibility of the same.
#-
#- CRITICAL APPLICATIONS
#- Xilinx products are not designed or intended to be fail-
#- safe, or for use in any application requiring fail-safe
#- performance, such as life-support or safety devices or
#- systems, Class III medical devices, nuclear facilities,
#- applications related to the deployment of airbags, or any
#- other applications that could lead to death, personal
#- injury, or severe property or environmental damage
#- (individually and collectively, "Critical
#- Applications"). Customer assumes the sole risk and
#- liability of any use of Xilinx products in Critical
#- Applications, subject only to applicable laws and
#- regulations governing limitations on product liability.
#-
#- THIS COPYRIGHT NOTICE AND DISCLAIMER MUST BE RETAINED AS
#- PART OF THIS FILE AT ALL TIMES.
#- ************************************************


This file contains confidential and proprietary information
   of Xilinx, Inc. and
is protected under U.S. and international copyright and other
    intellectual
```

```
property laws.

DISCLAIMER
This disclaimer is not a license and does not grant any
   rights to the materials
distributed herewith. Except as otherwise provided in a valid
    license issued to
you by Xilinx, and to the maximum extent permitted by
   applicable law:
(1) THESE MATERIALS ARE MADE AVAILABLE "AS IS" AND WITH ALL
   FAULTS, AND XILINX
HEREBY DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS,
   IMPLIED, OR STATUTORY,
INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY,
   NON-INFRINGEMENT, OR
FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not
   be liable (whether
in contract or tort, including negligence, or under any other
    theory of
liability) for any loss or damage of any kind or nature
   related to, arising under
or in connection with these materials, including for any
   direct, or any indirect,
special, incidental, or consequential loss or damage (
   including loss of data,
profits, goodwill, or any type of loss or damage suffered as
   a result of any
action brought by a third party) even if such damage or loss
   was reasonably
foreseeable or Xilinx had been advised of the possibility of
   the same.

CRITICAL APPLICATIONS
Xilinx products are not designed or intended to be fail-safe,
    or for use in any
application requiring fail-safe performance, such as life-
   support or safety
devices or systems, Class III medical devices, nuclear
   facilities, applications
```

```
related to the deployment of airbags, or any other
   applications that could lead
to death, personal injury, or severe property or
   environmental damage
(individually and collectively, "Critical Applications").
   Customer assumes the
sole risk and liability of any use of Xilinx products in
   Critical Applications,
subject only to applicable laws and regulations governing
   limitations on product
liability.

THIS COPYRIGHT NOTICE AND DISCLAIMER MUST BE RETAINED AS PART
    OF THIS FILE AT
ALL TIMES.

**********************************************************/
#include "ldl_top.h"

float multsubnew1 (float t, float a, float b) {
#pragma HLS inline off
  return t-a*b;
}
float multsubnew2 (float t, float a, float b) {
#pragma HLS inline off
  return t-a*b;
}
float multsubnew3 (float t, float a, float b) {
#pragma HLS inline off
  return t-a*b;
}
float multsubnew4 (float t, float a, float b) {
#pragma HLS inline off
  return t-a*b;
}

void ldl_top(float V[N],float L[N][N],float D[N],float A[N][N
   ])
{
```

```
#pragma HLS allocation instances=multsubnew1 limit=1 function
#pragma HLS allocation instances=multsubnew2 limit=1 function
#pragma HLS allocation instances=multsubnew3 limit=1 function
#pragma HLS allocation instances=multsubnew4 limit=1 function
#pragma HLS allocation instances=multsubnew5 limit=1 function
#pragma HLS allocation instances=multsubnew6 limit=1 function
#pragma HLS allocation instances=multsubnew7 limit=1 function
#pragma HLS allocation instances=multsubnew8 limit=1 function

  float t;
#pragma HLS ARRAY_PARTITION variable=L complete dim=0
for (int j = 0; j < N; j++) {
       for (int k = 0; k < N; k++) {
              if(k < j - 1) {
         V[k] = L[j][k]*D[k];
              }
       }
       t = A[j][j];
       for (int k = 0; k < j - 1; k+=8) {
              if(1) {
         t = multsub1(t,L[j][k],V[j]);
         t = multsub2(t,L[j][k+1],V[j]);
         t = multsub3(t,L[j][k+2],V[j]);
         t = multsub4(t,L[j][k+3],V[j]);
         t = multsub5(t,L[j][k+4],V[j]);
         t = multsub6(t,L[j][k+5],V[j]);
         t = multsub7(t,L[j][k+6],V[j]);
         t = multsub8(t,L[j][k+7],V[j]);
              }
       }
       D[j] = t;
       for (int i = j + 1; i < N; i++) {
              t = A[i][j];
              for (int k = 0; k < N; k++) {
                     if(k < j - 1) {
         t -= L[i][k]*V[k];
                     }
              }
       L[i][j]=t/D[j];
```

```
        }
}
}
```

Appendix B

Data-processing python codes

B.1 Finding optimal proximity solution through clustering - knp_mod 2+2

All packages required are imported.

```
In [1]: import pandas as pd
        import numpy as np
        from sklearn.cluster import KMeans
        import matplotlib.pyplot as plt
        import pydotplus
        import random
```

Firstly, the dot file with the Data Flow Graph is read to extract the coordinates of the MAC nodes. Reading the label it is possible to select the float MAC instances and the coordinates are saved in vectors xcoord and ycoord.

```
In [2]: G = pydotplus.graph_from_dot_file('knp_o.dot')

        names = []
        xcoord = []
        ycoord = []
        for n in G.get_node_list():
            name = n.get_name()
            label = n.get_label();
```

```
            if name in ['graph', 'node']: continue
            if not "fmac" in label: continue
            x, y = n.get('pos').replace('"','').split(',')
            x, y = float(x), float(y)
            print ("Position:", name, label, "=", x, y)
            xcoord.append(x)
            ycoord.append(y)
```

```
Position: node_369 "<constant:fmac0>\nfmac0" = 6968.0 2278.7
Position: node_414 "<constant:fmac1>\nfmac1" = 5247.0 2278.7
Position: node_457 "<constant:fmac2>\nfmac2" = 6562.0 2278.7
Position: node_500 "<constant:fmac3>\nfmac3" = 1439.0 2278.7
Position: node_545 "<constant:fmac4>\nfmac4" = 6032.0 2278.7
Position: node_588 "<constant:fmac5>\nfmac5" = 2668.0 2278.7
```

K-means clustering is a library used to generate an arbitrary number of clusters, selected at this point.

```
In [3]: df = pd.DataFrame({'x': xcoord, 'y': ycoord})
        kmeans = KMeans(n_clusters=3)
        kmeans.fit(df)
        df
```

```
Out[3]:          x       y
        0  6968.0  2278.7
        1  5247.0  2278.7
        2  6562.0  2278.7
        3  1439.0  2278.7
        4  6032.0  2278.7
        5  2668.0  2278.7
```

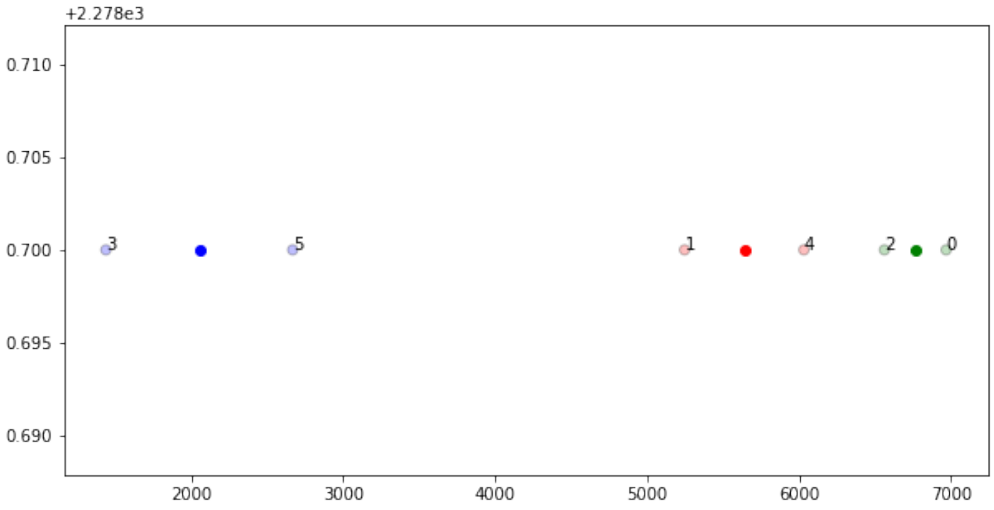
Kmeans generated the clusters indicating each element as a number form 0 to number_clusters -1, this number is then associated to the instance at the corresponding position. The points and centroids of different clusters are indicated by different colours.

```
In [4]: labels = kmeans.predict(df)
        centroids = kmeans.cluster_centers_
        colmap = {0: 'r', 1: 'b', 2: 'g', 3: 'k', 4:'y', 5:'m'}
        colors = list(map(lambda x: colmap[x], labels))
        labels
```

```
Out[4]: array([2, 0, 2, 1, 0, 1])
```

The points representing the instances of MACs are plotted, the centroids are highlighted.

```
In [5]: fig = plt.figure(figsize=(10, 5))
        plt.scatter(df['x'], df['y'], color=colors, alpha=0.25, edgecolor='k')
        na=["0", "1", "2","3","4","5"]
        for i, txt in enumerate(na):
            plt.annotate(txt, (df['x'][i], df['y'][i]))
        for idx, centroid in enumerate(centroids):
            plt.scatter(*centroid, color=colmap[idx])
        plt.show()
```



Afterwards, the same process is repeated for the integer MACs: The coordinates of the MAC nodes are extracted from the Data Flow Graph, then K-means generates the chosen number of clusters. Finally, the points and centroids are plotted.

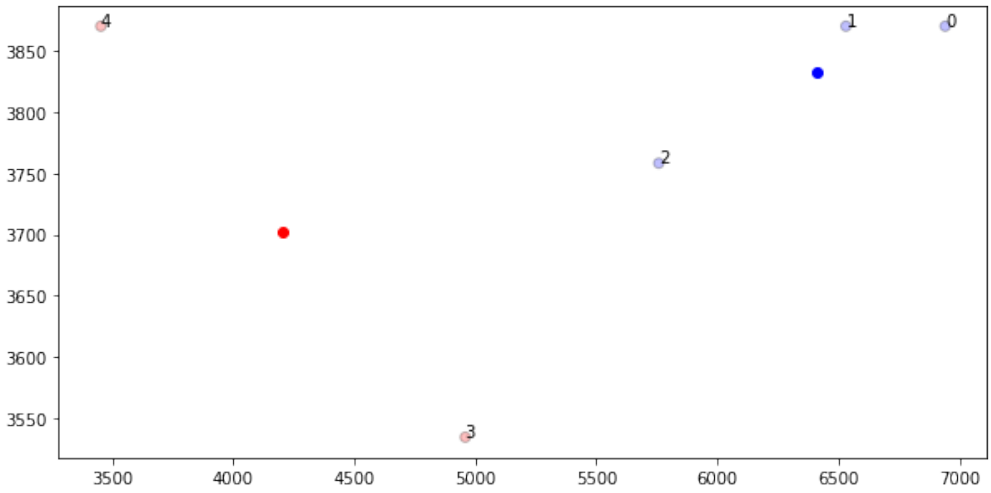
```
In [6]: names2 = []
        xcoord2 = []
        ycoord2 = []
        for n in G.get_node_list():
            name = n.get_name()
            label = n.get_label();
```

```
        if name in ['graph', 'node']: continue
        if not "imac" in label: continue
        x, y = n.get('pos').replace('"','').split(',')
        x, y = float(x), float(y)
        print ("Position:", name, label, "=", x, y)
        xcoord2.append(x)
        ycoord2.append(y)

df = pd.DataFrame({'x': xcoord2, 'y': ycoord2})
kmeans = KMeans(n_clusters=2)
kmeans.fit(df)

labels2 = kmeans.predict(df)
centroids = kmeans.cluster_centers_
colmap = {0: 'r', 1: 'b', 2: 'g', 3: 'k', 4:'y', 5:'m'}
colors = list(map(lambda x: colmap[x], labels2))
labels2

fig = plt.figure(figsize=(10, 5))
plt.scatter(df['x'], df['y'], color=colors, alpha=0.25, edgecolor='k')
na=["0", "1", "2","3","4"]
for i, txt in enumerate(na):
    plt.annotate(txt, (df['x'][i], df['y'][i]))
for idx, centroid in enumerate(centroids):
    plt.scatter(*centroid, color=colmap[idx])
plt.show()
```



```
Position: node_256 "<constant:imac6>\nimac6" = 6941.0 3870.4
Position: node_308 "<constant:imac7>\nimac7" = 6531.0 3870.4
Position: node_310 "<constant:imac8>\nimac8" = 5758.0 3758.4
Position: node_337 "<constant:imac9>\nimac9" = 4959.0 3534.4
Position: node_339 "<constant:imac10>\nimac10" = 3454.0 3870.4
```

Lastly, a new .cpp with the optimum locality solution is generated modifying the original .cpp file. The name of the MACs instances are replaced with a new name representing the cluster they have been mapped in. Then that MAC function is going to be limited to 1 instance only so that the resource is going to be shared.

```
In [7]: with open('knp_mod.cpp', 'r') as file:
            p = file.readlines()

        keyword = "fmac"
        for i in range(len(labels)):
            original_name=keyword+str(i)
            new_name=keyword+"_new"+str(labels[i])
            for x in range(0, len(p)):
                if "#pragma" in p[x] or "float" in p[x]\
                or "int" in p[x]: continue
                p[x] = p[x].replace(original_name, new_name)

        keyword2 = "imac"
        for i in range(len(labels2)):
            original_name2=keyword2+ str(i+6)
            new_name2=keyword2+"_new"+ str(labels2[i]+6)
            for x in range(0, len(p)):
                if "#pragma" in p[x] or "float" in p[x] \
                or "int imac_new" in p[x]: continue
                p[x] = p[x].replace(original_name2, new_name2)

        s = open("1+3circo.cpp", 'w')
        for item in p:
                s.write("%s" % item)
        s.close()
```

Once the optimum solution according to the locality method has been generated, it is useful to generate some random cluster solutions in order to evaluate the goodness of the solution found. To do so, it is necessary

to shuffle the vector in which the clusters are indicated as a number form 0 to #clusters-1. This number is then associated to the instance at the corresponding position.

```
In [8]: for y in range(0, 20):
            random.shuffle(labels)
            random.shuffle(labels2)
            with open('knp_mod.cpp', 'r') as file:
                p = file.readlines()

            keyword = "fmac"
            for i in range(len(labels)):
                original_name=keyword+str(i)
                new_name=keyword+"_new"+str(labels[i])
                for x in range(0, len(p)):
                    if "#pragma" in p[x] or "float" in p[x] \
                    or "int" in p[x]: continue
                    p[x] = p[x].replace(original_name, new_name)

            keyword2 = "imac"
            for i in range(len(labels2)):
                original_name2=keyword2+ str(i+6)
                new_name2=keyword2+"_new"+ str(labels2[i]+6)
                for x in range(0, len(p)):
                    if "#pragma" in p[x] or "float" in p[x] or \
                    "int imac_new" in p[x]: continue
                    p[x] = p[x].replace(original_name2, new_name2)

            s = open("2+2random" +str(y)+".cpp", 'w')
            for item in p:
                s.write("%s" % item)
            s.write("\n //"+ str(labels)+"\n //"+ str(labels2))
            s.close()
```

B.2 Comparison of post implementation timing and resource usage - FIR filter

Importing all the required packages

```
In [1]: import pandas as pd
        import numpy as np
```

```
        import os
        import math
        import matplotlib.pyplot as plt
        from sklearn.cluster import KMeans
        import pydotplus
        from sklearn.linear_model import LinearRegression
        from c.clustering.equal_groups import EqualGroupsKMeans
```

First of all, the post routing timing report are analyzed from the cpp_fir_timing_paths_routed.rpt files. All the report files, from different solutions are read and the path timings are extracted. They include 10 paths each, for every solution the maximum delay is computed.

Afterwards, the post implementation resource usage is evaluated: all the report files from different solutions are read to extract information about the percentage of FPGA resources employed. The maximum percentage is computed.

```
In [2]: pari = open(r"C:\Users\robyp\Google Drive\Tesi\FIR\3mac\fircirco\
\report\cpp_FIR_timing_paths_routed.rpt")
        p = pari.readlines()
        max_delay=0
        array_delay=[]
        latencymin=[]
        latencymax=[]
        array_area=[]

        for i in range(0, len(p)):
            if not "Data Path Delay:" in p[i]: continue
            delay = float(p[i].strip("  Data Path Delay:        ")\
            .split(" ")[0].strip("ns"))
            if delay> max_delay:
                max_delay=delay
        array_delay = np.append (array_delay, [max_delay])

        pa = open(r"C:\Users\robyp\Google Drive\Tesi\FIR\3mac\fircirco\
        \syn\report\cpp_FIR_csynth.rpt")
        p = pa.readlines()
        for i in range(0, len(p)):
            if "+ Latency (clock cycles):" in p[i]:
                latencymin=np.append(latencymin, int(p[i+6].split("|")[1]\
        .strip("  ")))
                latencymax=np.append(latencymax,\
```

```
         int(p[i+6].split("|")[2].strip("  ")))

        pari = open(r"C:\Users\robyp\Google Drive\Tesi\FIR\3mac\fircirco\
        \report\cpp_FIR_utilization_routed.rpt")
        p = pari.readlines()
        for i in range(0, len(p)):
            if "| Slice LUTs" in p[i]:
                LUT = (p[i].split("|")[5]\
        .strip(" "))
            if "| Slice Registers" in p[i]:
                FF = (p[i].split("|")[5]\
        .strip(" "))
            if "| Slice                                          |" in p[i]:
                SLICE = (p[i].split("|")[5].strip(" "))
            if "| DSPs" in p[i]:
                DSP = (p[i].split("|")[5].strip(" "))
            if "| Block RAM Tile" in p[i]:
                BRAM = (p[i].split("|")[5].strip(" "))
            if "| LUT as Memory" in p[i]:
                SRL = (p[i].split("|")[5].strip(" "))
        area= max(float(SLICE), float(LUT), float(FF), float(DSP) ,float(BRAM))
        array_area = np.append (array_area, [area])

In [3]: pari = open(r"C:\Users\robyp\Google Drive\Tesi\FIR\3mac\firdot\
        \report\cpp_FIR_timing_paths_routed.rpt")
        p = pari.readlines()
        max_delay=0

        for i in range(0, len(p)):
            if not "Data Path Delay:" in p[i]: continue
            delay = float(p[i].strip("  Data Path Delay:        ")\
        .split(" ")[0].strip("ns"))
            if delay> max_delay:
                max_delay=delay
        array_delay = np.append (array_delay, [max_delay])
        print(max_delay)

        pa = open(r"C:\Users\robyp\Google Drive\Tesi\FIR\3mac\firdot\
        \syn\report\cpp_FIR_csynth.rpt")
        p = pa.readlines()
        for i in range(0, len(p)):
            if "+ Latency (clock cycles):" in p[i]:
                latencymin=np.append(latencymin, int(p[i+6]\
```

```
        .split("|")[1].strip("  ")))
                latencymax=np.append(latencymax, int(p[i+6]\
        .split("|")[2].strip("  ")))

        pari = open(r"C:\Users\robyp\Google Drive\Tesi\FIR\3mac\firdot\
        \report\cpp_FIR_utilization_routed.rpt")
        p = pari.readlines()
        for i in range(0, len(p)):
            if "| Slice LUTs" in p[i]:
                LUT = (p[i].split("|")[5].strip(" "))
            if "| Slice Registers" in p[i]:
                FF = (p[i].split("|")[5].strip(" "))
            if "| Slice                                          |" in p[i]:
                SLICE = (p[i].split("|")[5].strip(" "))
            if "| DSPs" in p[i]:
                DSP = (p[i].split("|")[5].strip(" "))
            if "| Block RAM Tile" in p[i]:
                BRAM = (p[i].split("|")[5].strip(" "))
            if "| LUT as Memory" in p[i]:
                SRL = (p[i].split("|")[5].strip(" "))
        area= max(float(SLICE), float(LUT), float(FF), float(DSP) ,float(BRAM))
        array_area = np.append (array_area, [area])
```

3.595

```
In [4]: pari = open(r"C:\Users\robyp\Google Drive\Tesi\FIR\3mac\firdotbal\\
        report\cpp_FIR_timing_paths_routed.rpt")
        p = pari.readlines()
        max_delay=0

        for i in range(0, len(p)):
            if not "Data Path Delay:" in p[i]: continue
            delay = float(p[i].strip("  Data Path Delay:        ")\
                    .split(" ")[0].strip("ns"))
            if delay> max_delay:
                max_delay=delay
        array_delay = np.append (array_delay, [max_delay])
        print(max_delay)

        pa = open(r"C:\Users\robyp\Google Drive\Tesi\FIR\3mac\firdotbal\
        \syn\report\cpp_FIR_csynth.rpt")
        p = pa.readlines()
```

```
        for i in range(0, len(p)):
            if "+ Latency (clock cycles):" in p[i]:
                latencymin=np.append(latencymin, int(p[i+6]\
                    .split("|")[1].strip("  ")))
                latencymax=np.append(latencymax, int(p[i+6]\
                    .split("|")[2].strip("  ")))

        pari = open(r"C:\Users\robyp\Google Drive\Tesi\FIR\3mac\firdotbal\
        \report\cpp_FIR_utilization_routed.rpt")
        p = pari.readlines()
        for i in range(0, len(p)):
            if "| Slice LUTs" in p[i]:
                LUT = (p[i].split("|")[5].strip(" "))
            if "| Slice Registers" in p[i]:
                FF = (p[i].split("|")[5].strip(" "))
            if "| Slice                                          |" in p[i]:
                SLICE = (p[i].split("|")[5].strip(" "))
            if "| DSPs" in p[i]:
                DSP = (p[i].split("|")[5].strip(" "))
            if "| Block RAM Tile" in p[i]:
                BRAM = (p[i].split("|")[5].strip(" "))
            if "| LUT as Memory" in p[i]:
                SRL = (p[i].split("|")[5].strip(" "))
        area= max(float(SLICE), float(LUT), float(FF), float(DSP) ,float(BRAM))
        array_area = np.append (array_area, [area])
```

```
3.314
```

```
In [5]: pari = open(r"C:\Users\robyp\Google Drive\Tesi\FIR\3mac\firpatch\
        \report\cpp_FIR_timing_paths_routed.rpt")
        p = pari.readlines()

        for i in range(0, len(p)):
            if not "Data Path Delay:" in p[i]: continue
            delay = float(p[i].strip("  Data Path Delay:        ")\
                    .split(" ")[0].strip("ns"))
            if delay> max_delay:
                max_delay=delay
        array_delay = np.append (array_delay, [max_delay])
        print(max_delay)

        pa = open(r"C:\Users\robyp\Google Drive\Tesi\FIR\3mac\firpatch\
```

```
        \syn\report\cpp_FIR_csynth.rpt")
        p = pa.readlines()
        for i in range(0, len(p)):
            if "+ Latency (clock cycles):" in p[i]:
                latencymin=np.append(latencymin, int(p[i+6]\
                    .split("|")[1].strip("  ")))
                latencymax=np.append(latencymax, int(p[i+6]\
                    .split("|")[2].strip("  ")))

        pari = open(r"C:\Users\robyp\Google Drive\Tesi\FIR\3mac\firpatch\
        \report\cpp_FIR_utilization_routed.rpt")
        p = pari.readlines()
        for i in range(0, len(p)):
            if "| Slice LUTs" in p[i]:
                LUT = (p[i].split("|")[5].strip(" "))
            if "| Slice Registers" in p[i]:
                FF = (p[i].split("|")[5].strip(" "))
            if "| Slice                                        |" in p[i]:
                SLICE = (p[i].split("|")[5].strip(" "))
            if "| DSPs" in p[i]:
                DSP = (p[i].split("|")[5].strip(" "))
            if "| Block RAM Tile" in p[i]:
                BRAM = (p[i].split("|")[5].strip(" "))
            if "| LUT as Memory" in p[i]:
                SRL = (p[i].split("|")[5].strip(" "))
        area= max(float(SLICE), float(LUT), float(FF), float(DSP) ,float(BRAM))
        array_area = np.append (array_area, [area])
```

3.724

In [6]:
```
        pari = open(r"C:\Users\robyp\Google Drive\Tesi\FIR\3mac\firpatchbal\
        \report\cpp_FIR_timing_paths_routed.rpt")
        p = pari.readlines()
        max_delay=0

        for i in range(0, len(p)):
            if not "Data Path Delay:" in p[i]: continue
            delay = float(p[i].strip("  Data Path Delay:        ")\
                    .split(" ")[0].strip("ns"))
            if delay> max_delay:
                max_delay=delay
        array_delay = np.append (array_delay, [max_delay])
```

```
        print(max_delay)

        pa = open(r"C:\Users\robyp\Google Drive\Tesi\FIR\3mac\firpatchbal\
        \syn\report\cpp_FIR_csynth.rpt")
        p = pa.readlines()
        for i in range(0, len(p)):
            if "+ Latency (clock cycles):" in p[i]:
                latencymin=np.append(latencymin, int(p[i+6]\
                    .split("|")[1].strip("  ")))
                latencymax=np.append(latencymax, int(p[i+6]\
                    .split("|")[2].strip("  ")))

        pari = open(r"C:\Users\robyp\Google Drive\Tesi\FIR\3mac\firpatchbal\
        \report\cpp_FIR_utilization_routed.rpt")
        p = pari.readlines()
        for i in range(0, len(p)):
            if "| Slice LUTs" in p[i]:
                LUT = (p[i].split("|")[5].strip(" "))
            if "| Slice Registers" in p[i]:
                FF = (p[i].split("|")[5].strip(" "))
            if "| Slice                                          |" in p[i]:
                SLICE = (p[i].split("|")[5].strip(" "))
            if "| DSPs" in p[i]:
                DSP = (p[i].split("|")[5].strip(" "))
            if "| Block RAM Tile" in p[i]:
                BRAM = (p[i].split("|")[5].strip(" "))
            if "| LUT as Memory" in p[i]:
                SRL = (p[i].split("|")[5].strip(" "))
        area= max(float(SLICE), float(LUT), float(FF), float(DSP) ,float(BRAM))
        array_area = np.append (array_area, [area])
        print(array_delay)
        print(array_area)
```

```
3.306
[3.592 3.595 3.314 3.724 3.306]
[1.   1.02 1.08 1.05 1.09]
```

```
In [7]: pari = open(r"C:\Users\robyp\Google Drive\Tesi\FIR\3mac\firsfdp\
        \report\cpp_FIR_timing_paths_routed.rpt")
        p = pari.readlines()
        max_delay=0
```

```
        for i in range(0, len(p)):
            if not "Data Path Delay:" in p[i]: continue
            delay = float(p[i].strip("  Data Path Delay:        ")\
 .split(" ")[0].strip("ns"))
            if delay> max_delay:
                max_delay=delay
        array_delay = np.append (array_delay, [max_delay])
        print(max_delay)

        pa = open(r"C:\Users\robyp\Google Drive\Tesi\FIR\3mac\firsfdp\
        \syn\report\cpp_FIR_csynth.rpt")
        p = pa.readlines()
        for i in range(0, len(p)):
            if "+ Latency (clock cycles):" in p[i]:
                latencymin=np.append(latencymin, int(p[i+6]\
 .split("|")[1].strip("  ")))
                latencymax=np.append(latencymax, int(p[i+6]\
 .split("|")[2].strip("  ")))

        pari = open(r"C:\Users\robyp\Google Drive\Tesi\FIR\3mac\firsfdp\
        \report\cpp_FIR_utilization_routed.rpt")
        p = pari.readlines()
        for i in range(0, len(p)):
            if "| Slice LUTs" in p[i]:
                LUT = (p[i].split("|")[5].strip(" "))
            if "| Slice Registers" in p[i]:
                FF = (p[i].split("|")[5].strip(" "))
            if "| Slice                                            |" in p[i]:
                SLICE = (p[i].split("|")[5].strip(" "))
            if "| DSPs" in p[i]:
                DSP = (p[i].split("|")[5].strip(" "))
            if "| Block RAM Tile" in p[i]:
                BRAM = (p[i].split("|")[5].strip(" "))
            if "| LUT as Memory" in p[i]:
                SRL = (p[i]\
                    .split("|")[5].strip(" "))
        area= max(float(SLICE), float(LUT), float(FF), float(DSP) ,float(BRAM))
        array_area = np.append (array_area, [area])
```

```
3.176
```

```
In [8]: pari = open(r"C:\Users\robyp\Google Drive\Tesi\FIR\3mac\firsfdpbal\
        \report\cpp_FIR_timing_paths_routed.rpt")
```

```
p = pari.readlines()

for i in range(0, len(p)):
    if not "Data Path Delay:" in p[i]: continue
    delay = float(p[i].strip("  Data Path Delay:        ")\
            .split(" ")[0].strip("ns"))
    if delay> max_delay:
        max_delay=delay
array_delay = np.append (array_delay, [max_delay])
print(max_delay)

pa = open(r"C:\Users\robyp\Google Drive\Tesi\FIR\3mac\firsfdpbal\
\syn\report\cpp_FIR_csynth.rpt")
p = pa.readlines()
for i in range(0, len(p)):
    if "+ Latency (clock cycles):" in p[i]:
        latencymin=np.append(latencymin, int(p[i+6]\
            .split("|")[1].strip("  ")))
        latencymax=np.append(latencymax, int(p[i+6]\
            .split("|")[2].strip("  ")))

pari = open(r"C:\Users\robyp\Google Drive\Tesi\FIR\3mac\firsfdpbal\
\report\cpp_FIR_utilization_routed.rpt")
p = pari.readlines()
for i in range(0, len(p)):
    if "| Slice LUTs" in p[i]:
        LUT = (p[i].split("|")[5].strip(" "))
    if "| Slice Registers" in p[i]:
        FF = (p[i].split("|")[5].strip(" "))
    if "| Slice                                        |" in p[i]:
        SLICE = (p[i].split("|")[5].strip(" "))
    if "| DSPs" in p[i]:
        DSP = (p[i].split("|")[5].strip(" "))
    if "| Block RAM Tile" in p[i]:
        BRAM = (p[i].split("|")[5].strip(" "))
    if "| LUT as Memory" in p[i]:
        SRL = (p[i].split("|")[5].strip(" "))
area= max(float(SLICE), float(LUT), float(FF), float(DSP) ,float(BRAM))
array_area = np.append (array_area, [area])
```

```
3.617
```

The other 10 random examples are analized through a for loop.

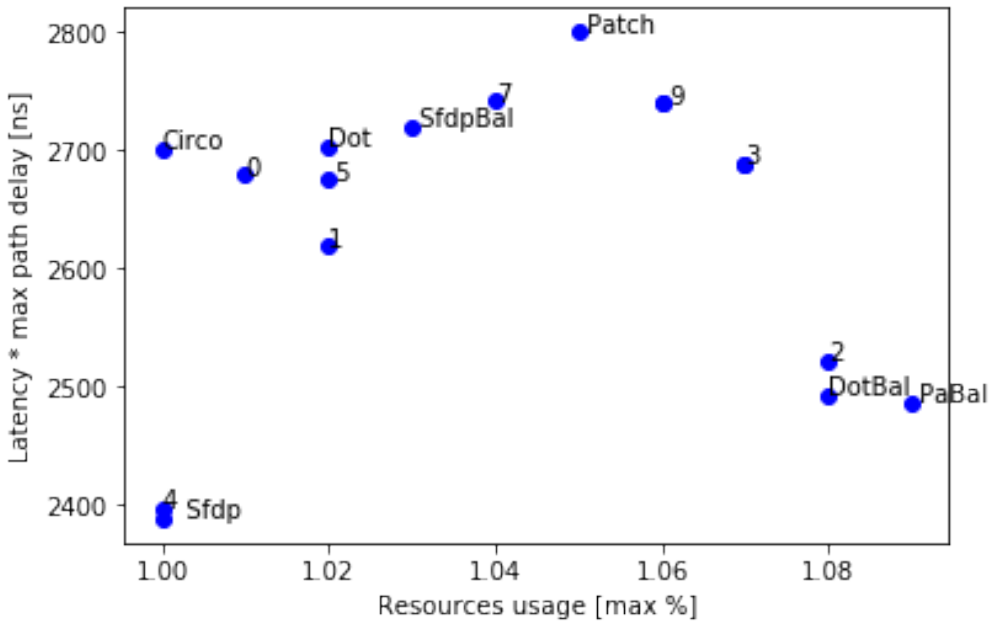
```
In [9]: for x in range(0, 10):
            pari = open(r"C:\Users\robyp\Google Drive\Tesi\FIR\3mac\
        \FIRrandom"+ str(x)+"\\report\cpp_FIR_timing_paths_routed.rpt")
            p = pari.readlines()
            max_delay=0
            for i in range(0, len(p)):
                if not "Data Path Delay:" in p[i]: continue
                delay = float(p[i].strip("  Data Path Delay:        ")\
                    .split(" ")[0].strip("ns"))
                if delay> max_delay:
                    max_delay=delay
            array_delay = np.append (array_delay, [max_delay])
            pari = open(r"C:\Users\robyp\Google Drive\Tesi\FIR\3mac\
        \FIRrandom"+str(x)+"\\syn\\report\cpp_FIR_csynth.rpt")
            p = pari.readlines()
            for i in range(0, len(p)):
                if "+ Latency (clock cycles):" in p[i]:
                    latencymin=np.append(latencymin, int(p[i+6]\
                    .split("|")[1].strip("  ")))
                    latencymax=np.append(latencymax, int(p[i+6]\
                    .split("|")[2].strip("  ")))

            pari = open(r"C:\Users\robyp\Google Drive\Tesi\FIR\3mac\
        \FIRrandom"+str(x)+"\\report\cpp_FIR_utilization_routed.rpt")
            p = pari.readlines()
            for i in range(0, len(p)):
                if "| Slice LUTs" in p[i]:
                    LUT = (p[i].split("|")[5].strip(" "))
                if "| Slice Registers" in p[i]:
                    FF = (p[i].split("|")[5].strip(" "))
                if "| Slice                                        |" in p[i]:
                    SLICE = (p[i].split("|")[5].strip(" "))
                if "| DSPs" in p[i]:
                    DSP = (p[i].split("|")[5].strip(" "))
                if "| Block RAM Tile" in p[i]:
                    BRAM = (p[i].split("|")[5].strip(" "))
                if "| LUT as Memory" in p[i]:
                    SRL = (p[i].split("|")[5].strip(" "))
            area= max(float(SLICE), float(LUT), float(FF), float(DSP) ,float(BRAM))
            array_area = np.append (array_area, [area])
```

Lastly, The results in terms of total area usage and maximum delay are plotted in a Area-Delay diagram. One diagram is relative to the minimum latency. the other to the maximum one.

```
In [10]: h= array_delay * latencymax
         print(h)
         plt.plot(array_area ,h, 'bo')
         plt.ylabel('Latency * max path delay [ns]')
         plt.xlabel('Resources usage [max %]')
         n=["Circo","Dot", "DotBal"," Patch"," PaBal","   Sfdp",\
        " SfdpBal","0", "1", "2","3","4"," 5","","7","","9"]
         for i, txt in enumerate(n):
             plt.annotate(txt, (array_area[i], h[i]))
         plt.show()
```

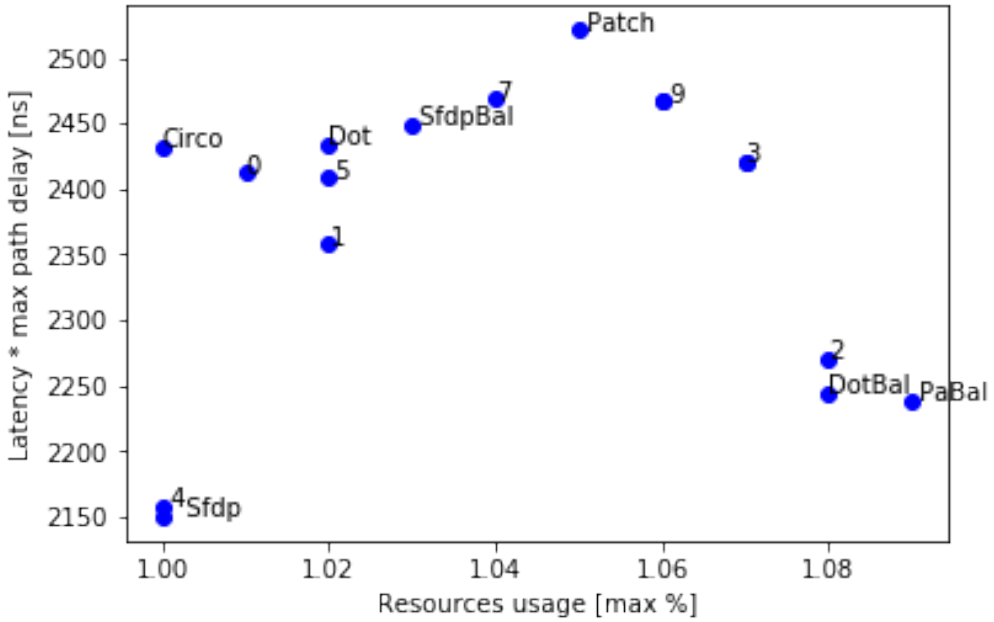
```
[2701.184 2703.44  2492.128 2800.448 2486.112 2388.352 2719.984 2679.376
 2619.216 2521.456 2688.4   2397.376 2674.864 2739.536 2741.792 2688.4
 2739.536]
```




```
In [11]: l=array_delay * latencymin
         print(l)
```

```
        plt.plot(array_area ,l, 'bo')
        plt.ylabel('Latency * max path delay [ns]')
        plt.xlabel('Resources usage [max %]')
        n=["Circo","Dot", "DotBal"," Patch"," PaBal","   Sfdp\
        "," SfdpBal","0", "1", "2","3"," 4"," 5","","7","","9"]
        for i, txt in enumerate(n):
            plt.annotate(txt, (array_area[i], l[i]))
        plt.show()
```

```
[2431.784 2433.815 2243.578 2521.148 2238.162 2150.152 2448.709 2412.151
 2357.991 2269.981 2420.275 2158.276 2408.089 2466.311 2468.342 2420.275
 2466.311]
```



In order to carry out the Clustering Cost comparision on the dot DFG, the balanced dot solution and the previously generated 10 random solutions are used to compute the clustering costs and plot them with the delay and the area.

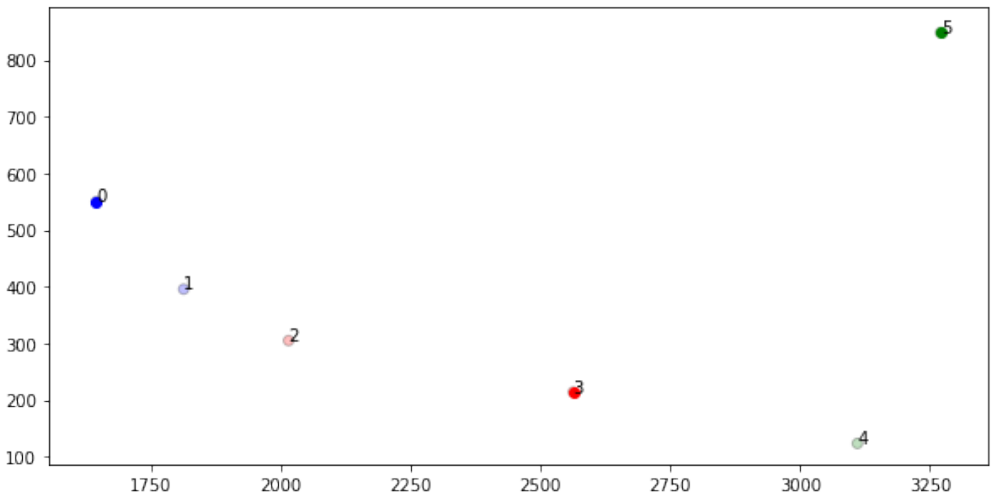
```
In [12]: G = pydotplus.graph_from_dot_file('opdot.dot')
         xcoord = []
         ycoord = []
```

```
cost_array=[]
for n in G.get_node_list():
    name = n.get_name()
    label = n.get_label();
    if name in ['graph', 'node']: continue
    if not "mac" in label: continue
    x, y = n.get('pos').replace('"','').split(',')
    x, y = float(x), float(y)
    xcoord.append(x)
    ycoord.append(y)

df = pd.DataFrame({'x': xcoord, 'y': ycoord})
kmeans = EqualGroupsKMeans(n_clusters=3)
kmeans.fit(df)

labels = kmeans.predict(df)
centroids = kmeans.cluster_centers_
colmap = {0: 'r', 1: 'b', 2: 'g', 3: 'y', 4:'k', 5:'m'}
colors = list(map(lambda x: colmap[x], labels))
labels

fig = plt.figure(figsize=(10, 5))
plt.scatter(df['x'], df['y'], color=colors, alpha=0.25, edgecolor='k')
na=["0", "1", "2","3", "4", "5"]
for i, txt in enumerate(na):
    plt.annotate(txt, (df['x'][i], df['y'][i]))
for idx, centroid in enumerate(centroids):
    plt.scatter(*centroid, color=colmap[idx])
plt.show()
cost=0
for a in range(0,len(labels)):
    for b in range(0,len(labels)):
        if labels[a]==labels[b] and (a != b):
            distance = np.sqrt((df.iat[b,0]-df.iat[a,0])**2\
           +(df.iat[b,1]-df.iat[a,1])**2)/2
            cost+=distance
cost_array = np.append (cost_array, [cost])
```




```
In [13]: for x in range(0, 10):
             pari = open(r"C:\Users\robyp\Google Drive\Tesi\FIR\codes\
             \codici\scarica\FIRrandom"+ str(x)+".h")
             p = pari.readlines()
             for i in range(0, len(p)):
                 if not "//[" in p[i]: continue
                 labell= p[i].split("[")[1].strip("]")
                 labels=labell.split(' ')
                 cost=0
                 for a in range(0,len(labels)):
                     for b in range(0,len(labels)):
                         if labels[a]==labels[b] and (a != b):
                             distance = np.sqrt((df.iat[b,0]-df.iat[a,0])**2\
                        +(df.iat[b,1]-df.iat[a,1])**2)/2
                             cost+=distance
                 cost_array = np.append (cost_array, [cost])
```

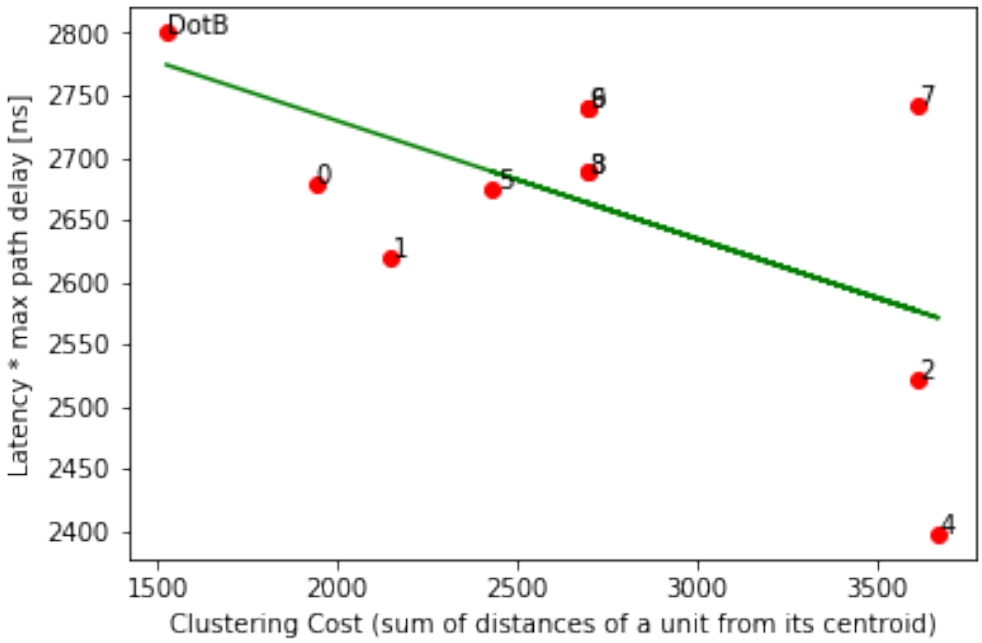
Appending the Dot Balanced solution to the "timing" vector

```
In [14]: timing = np.append (h[3], [h[7:18]])
```

Plotting of the maximum execution time vs Clustering cost obtained opening the DFGs through "dot". Afterwards, the linear regression is computed

```
In [15]: plt.plot(cost_array, timing, 'ro')
         plt.ylabel('Latency * max path delay [ns]')
```

```
plt.xlabel('Clustering Cost')
n=["DotB", "0", "1", "2","3","4"," 5","6","7","8","9"]
for i, txt in enumerate(n):
    plt.annotate(txt, (cost_array[i], timing[i]))
x = cost_array.reshape((-1, 1))
y = LinearRegression().fit(x, timing).predict(x)
plt.plot(x, y, color='green')
plt.show()
```



Computation of the Coefficient of determination R^2

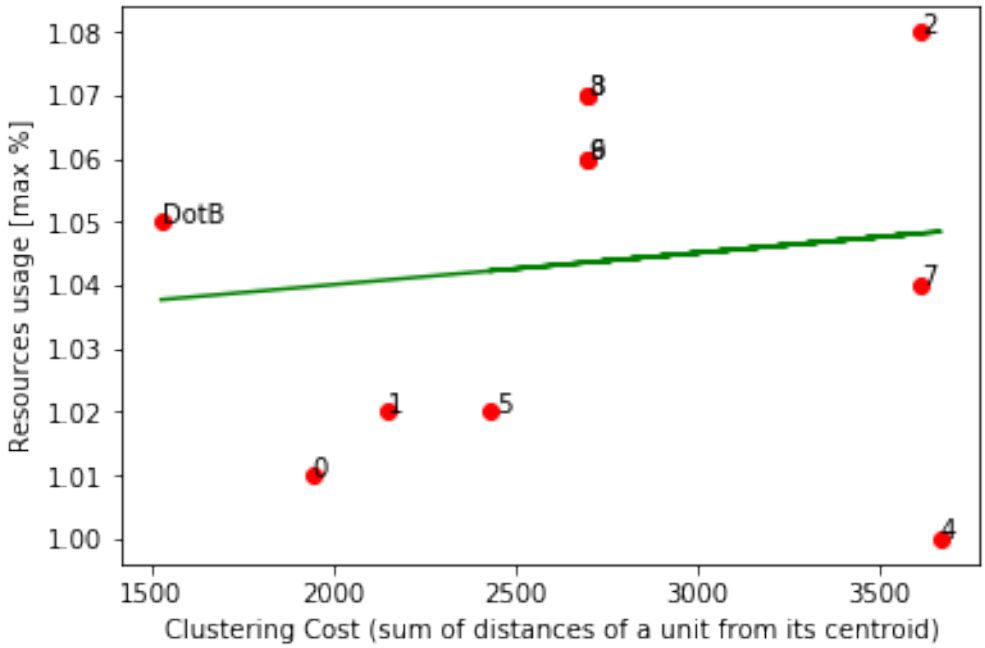
```
In [16]: LinearRegression().fit(x, timing).score(x, timing)

Out[16]: 0.33931090957003107
```

Plotting of the maximum resource usage vs Clustering cost obtained opening the DFGs through "dot". Afterwards, the linear regression is computed

```
In [17]: areas = np.append (array_area[3], [array_area[7:18]])
         plt.plot(cost_array, areas, 'ro')
         plt.ylabel('Resources usage [max %]')
         plt.xlabel('Clustering Cost')
```

```
n=[ "DotB", "0", "1", "2","3","4"," 5","6","7","8","9"]
for i, txt in enumerate(n):
    plt.annotate(txt, (cost_array[i], areas[i]))
x = cost_array.reshape((-1, 1))
y = LinearRegression().fit(x, areas).predict(x)
plt.plot(x, y, color='green')
plt.show()
```



Computation of the Coefficient of determination R^2

```
In [18]: LinearRegression().fit(x, areas).score(x, areas)

Out[18]: 0.016649140067634516
```

The same procedure has been repeated for the DFGs opened using SFDP Graaphviz tool

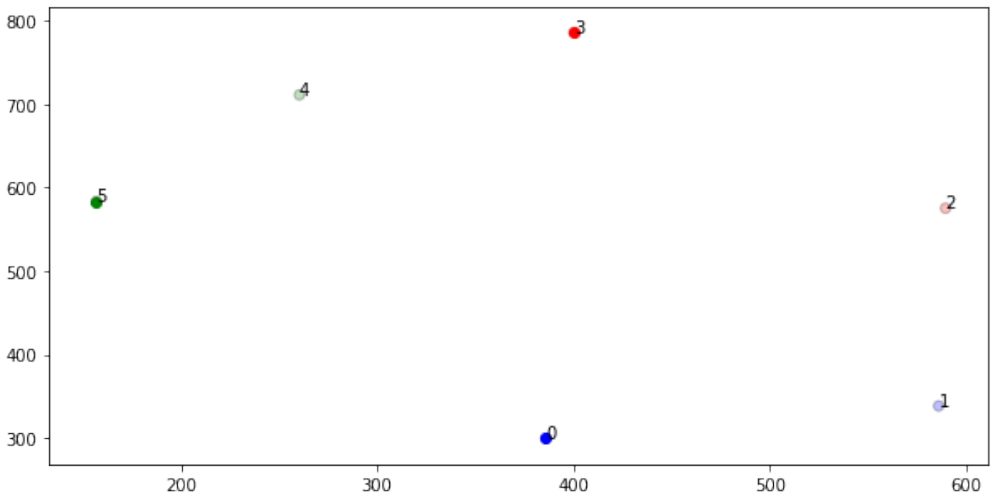
```
In [19]: cost_array_sfdp=[]
         G = pydotplus.graph_from_dot_file('opsfdp.dot')
         xcoord = []
         ycoord = []
         for n in G.get_node_list():
```

```
    name = n.get_name()
    label = n.get_label();
    if name in ['graph', 'node']: continue
    if not "mac" in label: continue
    x, y = n.get('pos').replace('"','').split(',')
    x, y = float(x), float(y)
    xcoord.append(x)
    ycoord.append(y)
df = pd.DataFrame({'x': xcoord, 'y': ycoord})

kmeans = EqualGroupsKMeans(n_clusters=3)
kmeans.fit(df)
labels = kmeans.predict(df)
centroids = kmeans.cluster_centers_
colmap = {0: 'r', 1: 'b', 2: 'g', 3: 'k', 4:'y', 5:'m'}
colors = list(map(lambda x: colmap[x], labels))
labels

fig = plt.figure(figsize=(10, 5))
plt.scatter(df['x'], df['y'], color=colors, alpha=0.25, edgecolor='k')
na=["0", "1", "2","3", "4", "5"]
for i, txt in enumerate(na):
    plt.annotate(txt, (df['x'][i], df['y'][i]))
for idx, centroid in enumerate(centroids):
    plt.scatter(*centroid, color=colmap[idx])
plt.show()

cost=0
for a in range(0,len(labels)):
    for b in range(0,len(labels)):
        if labels[a]==labels[b] and (a != b):
            distance = np.sqrt((df.iat[b,0]-df.iat[a,0] \
            )**2+(df.iat[b,1]-df.iat[a,1])**2)/2
            cost+=distance
cost_array_sfdp = np.append (cost_array_sfdp, [cost])
```

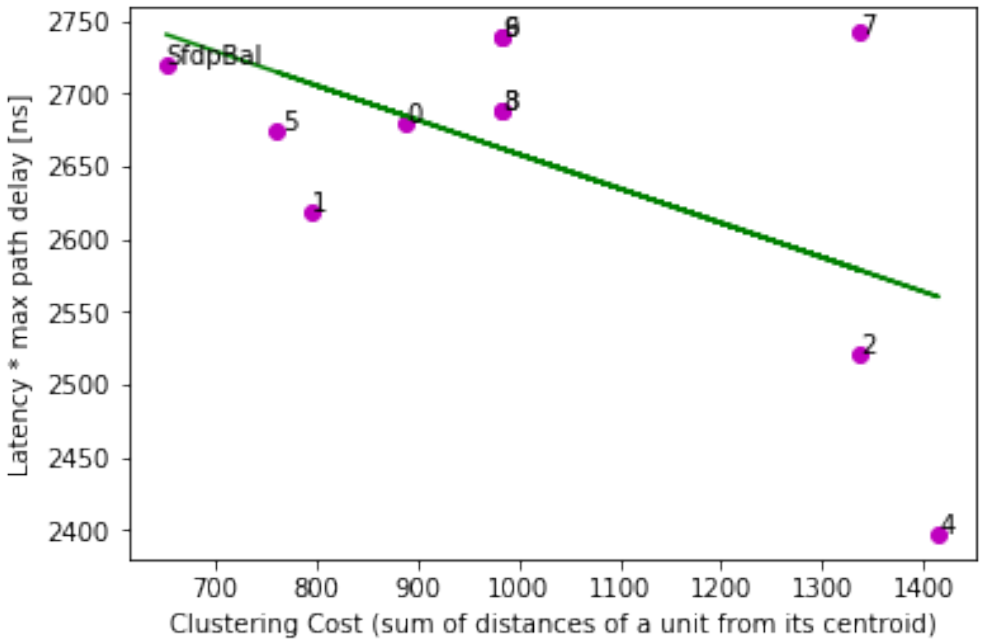



```
In [20]: for x in range(0, 10):
             pari = open(r"C:\Users\robyp\Google Drive\Tesi\FIR\\
             codes\codici\scarica\FIRrandom"+ str(x)+".h")
             p = pari.readlines()
             for i in range(0, len(p)):
                 if not "//[" in p[i]: continue
                 labell= p[i].split("[")[1].strip("]")
                 labels=labell.split(' ')
                 cost=0
                 for a in range(0,len(labels)):
                     for b in range(0,len(labels)):
                         if labels[a]==labels[b] and (a != b):
                             distance = np.sqrt((df.iat[b,0]-df.iat[\
                        PYna,0])**2+(df.iat[b,1]-df.iat[a,1])**2)/2
                             cost+=distance
                 cost_array_sfdp = np.append (cost_array_sfdp, [cost])
```

Plotting of the maximum execution time vs Clustering cost obtained opening the DFGs through “sfdp”. Afterwards, the linear regression is computed

```
In [21]: timing = h[6:18]
         plt.plot(cost_array_sfdp, timing, 'mo')
         plt.ylabel('Latency * max path delay [ns]')
         plt.xlabel('Clustering Cost')
         n=["SfdpBal","0", "1", "2","3","4"," 5","6","7","8","9"]
```

```
    for i, txt in enumerate(n):
        plt.annotate(txt, (cost_array_sfdp[i], timing[i]))
    x = cost_array_sfdp.reshape((-1, 1))
    y = LinearRegression().fit(x, timing).predict(x)
    plt.plot(x, y, color='green')
    plt.show()
    LinearRegression().fit(x, timing).score(x, timing)
```

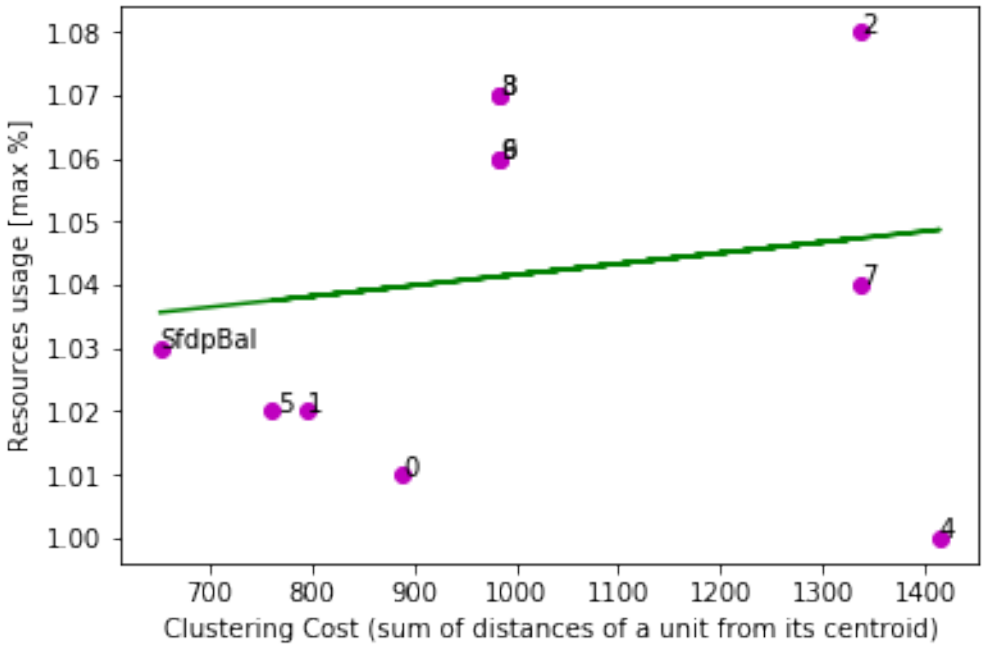



```
Out[21]: 0.30669511679283074
```

Plotting of the maximum resource usage vs Clustering cost obtained opening the DFGs through "sfdp". Afterwards, the linear regression is computed

```
In [22]: areas=array_area[6:18]
         plt.plot(cost_array_sfdp, areas, 'mo')
         plt.ylabel('Resources usage [max %]')
         plt.xlabel('Clustering Cost')
         n=["SfdpBal","0", "1", "2","3","4"," 5","6","7","8","9"]
         for i, txt in enumerate(n):
             plt.annotate(txt, (cost_array_sfdp[i], areas[i]))
         x = cost_array_sfdp.reshape((-1, 1))
```

```
y = LinearRegression().fit(x, areas).predict(x)
plt.plot(x, y, color='green')
plt.show()
LinearRegression().fit(x, areas).score(x, areas)
```



Out[22]: 0.02447402729604997