POLITECNICO DI TORINO

Master's Course
in Electronic Engineering

Master's Degree Thesis

Modeling and Mapping of a GoogLeNet CNN on a Grid of Processing Cells

Supervisors:
Prof. Rainer Dömer (UC Irvine)
Prof. Guido Masera

Candidate:
Claudio Raccomandato

Academic Year 2021-2022

Abstract

System-level design methodologies evolve in response to increasing complexity of applications. Transaction-level modeling (TLM) is one technique that allows the designer to capture the specifications of complex digital systems without defining low-level implementation details. The Grid of Processing Cells (GPC) has been proposed as a highly scalable many-core architecture and is modeled using SystemC TLM-2.0 methodology. This thesis describes the modeling of a GoogLeNet Convolutional Neural Network (CNN) on the GPC architecture and evaluates its performance and scalability. The models feature a new modular Memory Access Resources and Interfaces (MARI) library to improve communication between modules and assist during profiling. This work also introduces a graphical CAD software called Map Grid-based Layouts (MapGL) to facilitate the design process, automatically generate SystemC models, and generate performance reports. Experimental results evaluate and compare the generated models and show the achieved improvements in terms of memory usage and speed.

Acknowledgements

I wish to thank Prof. Rainer Dömer and his research group at the University of California Irvine for allowing me to contribute to their study. Thank you for kindly including me and making me feel at home during my stay in Irvine.
I also thank Prof. Guido Masera for his presence and valuable advice despite the distance.
I acknowledge my parents and family, who always encouraged me to follow my passions and laid the foundation for my professional life.
Last but not least, I wish to thank my girlfriend and all the friends I met in Turin who supported me during these beautiful university years.

Desidero ringraziare il Prof. Rainer Dömer e il suo gruppo di ricerca della University of California Irvine per avermi permesso di contribuire al loro lavoro. Grazie per avermi accolto calorosamente e fatto sentire come se fossi a casa durante la mia permanenza a Irvine.
Ringrazio anche il Prof. Guido Masera per la sua disponibilità e i preziosi consigli nonostante la distanza.
Ringrazio i miei genitori e parenti, i quali mi hanno sempre incoraggiato a perseguire le mie passioni e hanno posto le basi per la mia vita professionale.
Ultimi ma non per importanza, desidero ringraziare la mia ragazza e tutti gli amici incontrati a Torino che mi hanno sostenuto durante questi bellissimi anni di università.

You'll never be good at anything before being bad at it first.

[Unknown]

Contents

List of Figures i

List of Tables iii

List of Listings iv

1 Introduction 1
- 1.1 Thesis structure . . . 2
- 1.2 Related Works . . . 2
  - 1.2.1 SystemC . . . 2
  - 1.2.2 GPC Architecture . . . 2
  - 1.2.3 GoogLeNet CNN . . . 4
- 1.3 Goals . . . 5

2 GoogLeNet CNN on GPC (Version 1) 7
- 2.1 Pen and Paper Mapping . . . 7
  - 2.1.1 MemTools . . . 8
  - 2.1.2 Fork and Merge . . . 10
  - 2.1.3 Inception Block . . . 11
  - 2.1.4 Complete structure . . . 12
- 2.2 Implementation . . . 12
  - 2.2.1 Parsing . . . 13
  - 2.2.2 Mapping Validation . . . 15
  - 2.2.3 Code Generation . . . 15
  - 2.2.4 Compilation . . . 19
  - 2.2.5 Profiling . . . 19
- 2.3 Limits of the model . . . 19

3 MARI: Memory Access Resources and Interfaces library 23
- 3.1 Memory Interfaces . . . 24
- 3.2 FIFO Interfaces . . . 26
  - 3.2.1 Ring Buffering . . . 27
- 3.3 Profiling . . . 27

4 MapGL **31**
4.1 Memories and Channels views . . . 31
4.2 Modules . . . 33
4.2.1 How to create a module . . . 34
4.3 Memory Structure . . . 36
4.4 Project Export . . . 37
4.4.1 Dependencies . . . 39
4.5 Profiling . . . 39
4.5.1 Memories Usage Analysis . . . 40
4.5.2 Timing Analysis . . . 42

5 GoogLeNet CNN on GPC (Version 2) **45**
5.1 Inception Block . . . 45
5.2 Complete Structure . . . 46
5.3 Profiling . . . 48

6 Experiments and Results **49**
6.1 Version 1 . . . 49
6.2 Version 2 . . . 50
6.2.1 Memories Usage . . . 51
6.2.2 Timing . . . 52
6.3 Comparison . . . 57

7 Conclusion **59**
7.1 Future works . . . 59

Bibliography **61**

List of Figures

1.1 Representation of a 4 by 4 GPC model with "OFF-chip" DRAM memories [5]. . . . 3
1.2 R-type cell inside the GPC architecture, redrawn from [5]. . . . 3
1.3 GoogLeNet CNN structure, redrawn from [13]. . . . 4

2.1 The structure of the inception blocks inside the GoogLeNet CNN, redrawn from [16]. . . . 8
2.2 MemTools memory organization, redrawn from [10]. . . . 9
2.3 Fork and Merge operations with MemTools code. . . . 10
2.4 Fork and Merge used as forwarding module. . . . 10
2.5 GoogLeNet CNN on GPC (version 1) inception block design. . . . 11
2.6 GoogLeNet CNN on GPC (version 1) structure. . . . 12
2.7 GoogLeNet CNN on GPC (version 1) complete implementation steps. . . . 14
2.8 GoogLeNet CNN on GPC (version 1) spreadsheet description of the first inception block. The four paths are highlighted in the bottom image. . . . 16
2.9 Validation algorithm. . . . 17
2.10 Spreadsheet unclear description. . . . 20

3.1 Example of communication between cores using MARI library, Channels View. . . . 24
3.2 Example of communication between cores using MARI library, Memories View. . . . 24
3.3 Example of ring buffering inside an 8-byte long FIFO. . . . 27
3.4 Memory access encoding inside the *mari.log* binary file. . . . 28

4.1 MapGL startup window. . . . 32
4.2 MapGL Memory Structure window. . . . 37
4.3 Error dialog that appears when at least one parameter has still the "val" attribute null. . . . 37
4.4 Dependencies window. . . . 39
4.5 Error dialog that appears when at least one dependency file cannot be found. 39
4.6 Memories Usage Analysis window. . . . 40
4.7 Timing Analysis window. . . . 42
4.8 Channel communication life span with delays contributions. . . . 43
4.9 Module life span with delays contributions. . . . 43

5.1 GoogLeNet CNN on GPC (version 2) inception block design. 46
5.2 GoogLeNet CNN on GPC (version 2) structure. 47

6.1 Memories usage analysis heatmaps of the high-speed (on the left) and low-speed (on the right) implementations of the GoogLeNet on GPC (version 2). 51
6.2 Timing analysis channel delays' heatmaps of the high-speed (on the left) and low-speed (on the right) implementations of the GoogLeNet on GPC (version 2). 53
6.3 Timing analysis real execution delays' heatmaps of the high-speed (on the left) and low-speed (on the right) implementations of the GoogLeNet on GPC (version 2). 55

List of Tables

3.1 FLAGS field encoding used inside the mari.log file. 28

6.1 Communication delays used for the timing profiling [11]. 53
6.2 Results of the first and second versions of the GoogLeNet on GPC model. The second version has high-speed and low-speed implementations. 57
6.3 Comparison between the first model and the high-speed and low-speed implementations. 57

List of Listings

2.1 First Convolutional layer description inside the layers Python dictionary . 13

2.2 Implementation of the conv1_7x7_s2 layer inside `checkerboard_user.cpp` file. 18

3.1 Methods declarations of the Mem_if class inside the MARI library. 25

3.2 MARI library FIFO interfaces constructors. 26

4.1 Example of an adder MapGL module implementation. 33

4.2 Example of the JSON file used to import the adder module. 34

4.3 Example of Memories usage analysis report. 40

4.4 Example of Timing analysis report. 44

6.1 Extract of the GoogLeNet CNN on GPC (version 1) profiling report. . . . 49

6.2 Extract of the GoogLeNet CNN on GPC (version 2) memories usage report for the high-speed implementation. 51

6.3 Extract of the GoogLeNet CNN on GPC (version 2) memories usage report for the low-speed implementation. 52

6.4 Extract of the GoogLeNet CNN on GPC (version 2) timing report for the high-speed implementation. 54

6.5 Extract of the GoogLeNet CNN on GPC (version 2) timing report for the low-speed implementation. 54

6.6 Extract of the GoogLeNet CNN on GPC (version 2) timing report with layers delays for the high-speed implementation. 56

6.7 Extract of the GoogLeNet CNN on GPC (version 2) timing report with layers delays for the low-speed implementation. 56

Chapter 1

Introduction

Over the last two decades, due to power limitations, computer systems focus shifted from raising the clock frequency toward the increase in the number of processors [1]. This phenomenon led to higher design complexity and shared memory contention caused by the "memory wall" problem [2].

The growth in complexity of applications drives the need for modeling systems at a higher level of abstraction called Electronic System Level (ESL) [3]. Then, Transaction Level Modeling (TLM) techniques allow refining portions of the system model towards lower levels. Nowadays, SystemC represents the language of choice to launch the adoption of ESL and TLM modeling [4]. It is based on C++, a common language for software and hardware, and provides simulation concurrency for both of them.

The Grid of Processing Cells (GPC) architecture uses the SystemC TLM2.0 methodology and has been proposed to reduce the shared memory bandwidth limitation using distributed local memories [5]. The structure follows a "checkerboard" pattern in which processing cores and local memories alternate on a regular 2D mesh. This design increases the number of pathways between cores, which mitigates the memory contention effect.

This thesis aims to push the GPC architecture's scalability by designing on it a GoogLeNet convolutional neural network (CNN) application.
The GoogLeNet is a low-parameter image classification CNN presented during the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) in 2014 [6].
The regularity of the GoogLeNet CNN structure and the high parallelization of its inception blocks should allow a straightforward mapping of its layers into the GPC architecture.

1.1 Thesis structure

Every chapter represents one design step towards creating and profiling the final model.

This introductory chapter discusses the related works and goals of this thesis. Then in Chapter 2, the first implementation will be presented along with its limitations. Chapter 3 and Chapter 4 describe the two tools created to aid the design of the final model presented in Chapter 5. Chapter 6 shows the two models' experimental results and compares them to quantify the improvements. Finally, Chapter 7 will discuss if the goals have been achieved and the future work ideas.

1.2 Related Works

This section will briefly describe the three major works related to the thesis: the SystemC language, the GPC architecture and the GoogLeNet CNN.

1.2.1 SystemC

SystemC is a System Level Design Language (SLDL) that provides constructors required for modeling hardware within the context of C++ [7][8].

The fundamental design components are the Modules, which are classes derived from the sc_module class. Similar to VHDL's entities and architectures, SystemC separates the interface and implementation of the module by using header and source files.

The SystemC simulator uses a cooperative multitasking model that allows concurrency simulation of the model. A class called sc_time can be used to track simulation time with 64 bits of resolution. Without introducing any delay, the execution time of every module is zero. In this case, the SystemC simulator uses delta cycles to perform operations in parallel.

With SystemC TLM2.0, the communication between modules takes place through memory-mapped bus models [9]. The TLM2.0 classes are built on top of the SystemC class library and contain interfaces, sockets, generic payloads and protocols.

1.2.2 GPC Architecture

The GPC is a highly scalable many-processor architecture designed with SystemC TLM2.0 methodology. The structure is constructed by alternating cores and local memory as shown in Figure 1.1 for a 4 by 4 grid. Since the pattern resembles a checkerboard, the structure was initially introduced as the "checkerboard" model [5].

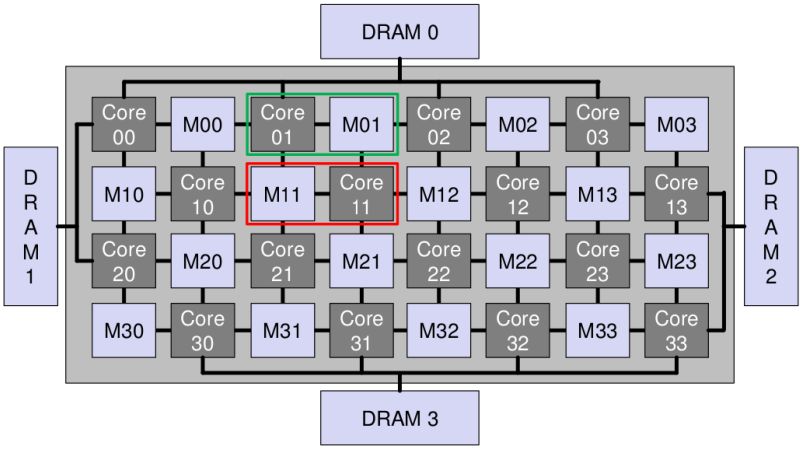

Figure 1.1. Representation of a 4 by 4 GPC model with "OFF-chip" DRAM memories [5].

The alternating placement allows for identifying the block called "cell" formed by a core and a memory. Looking at Figure 1.1, it is possible to recognize two types of cells: the L-type with the green border and the R-type with the red one.

Every core has a TLM2.0 initiator socket that binds to a maximum of 4 priority-based multiplexers, one for each surrounding memory. Figure 1.2 shows a detailed representation of an R-type cell, with multiplexer and the TML2.0 sockets.

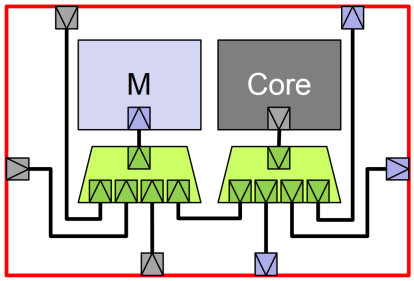

Figure 1.2. R-type cell inside the GPC architecture, redrawn from [5].

The advantage of using a distributed memory organization is that every core connects to smaller memories, which are expected to be faster than large ones.
The contention is reduced by allowing cores only to access the surrounding memories. In this way, the worst-case scenario is four cores that want to access the same memory. This is a significant improvement compared to regular multi-processor architectures in which each core communicates with the same memory. On the other hand, local memories limit cores to connect with just their neighbors. External DRAM memories act as "highways"

through which cores far from each other can communicate directly. Of course, external memories are expected to be slower than the ones "ON-chip", but for large structures, that can be a valid option to speed up communication.

Some computer vision applications that make use of the GPC architecture have been designed by the CECS group at the University of California, Irvine [10][11][12].
These works have shown the superiority of the GPC over traditional multi-processor structures based on shared memory.

1.2.3 GoogLeNet CNN

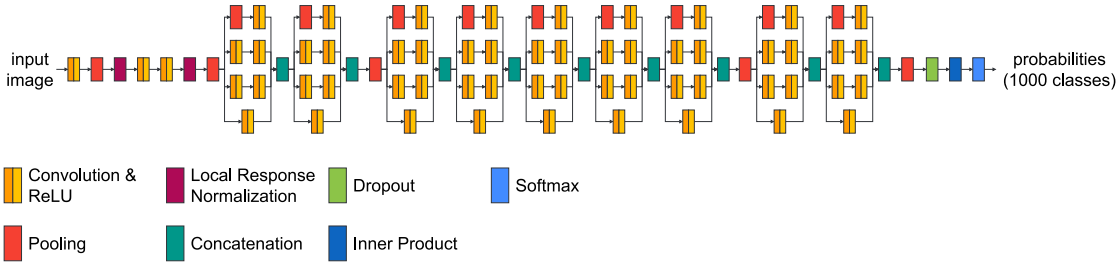

Figure 1.3. GoogLeNet CNN structure, redrawn from [13].

The problem of assigning a predetermined descriptive label to an input image is known as image classification.
Convolutional neural networks (CNN) have been used to solve this problem since they allow fast and relatively accurate classification.

The GoogLeNet is a state-of-the-art CNN for image classification, winner of the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) 2014 with only 6.67% top-5 error [6]. The network is composed of 22 layers when counting only layers with parameters or 142 if not. Figure 1.3 shows the entire structure of the GoogLeNet CNN.

Caffe is a deep learning framework created by the Berkeley AI Research group (BAIR) in 2014 that provides a complete toolkit for training, testing, finetuning, and deploying CNN models [14]. In this thesis, the GoogLeNet Caffe model will be used to extract the network structure information.

OpenCV is a C/C++ open-source library that provides tools needed to solve real-time computer vision problems [15]. It also provides constructors that can be used in CNN applications like the Mat data structure or the Layer class.

A SystemC model of the GoogLeNet CNN that use the Caffe model and OpenCV has been designed by the CECS group at the University of California, Irvine [16]. This thesis will use that model as a reference for the new one mapped on the GPC architecture.

1.3 Goals

This thesis work aims to achieve the three goals described below.

1. **Exploit the scalability of the GPC architecture with a CNN application.**
   Mapping all the GoogLeNet CNN layers will require at least 142 cores, which is not trivial for ordinary multi-processor architecture. We want to demonstrate that the GPC can handle such large applications by creating and testing a working model.

2. **Improve the mapping process for GPC-based applications.**
   Mapping is accomplished by copying and pasting C/C++ algorithms inside the cores *main()* function definition in the `checkerboard_user.cpp` file. Even if this method works well for small models, it cannot be applied to large applications due to the time the task demand and the possibility of creating typos inside the code.

3. **Create a new way of profiling GPC models.**
   Obtaining experimental results about the architecture delays, occupied area and power consumption is challenging for high-level applications. Estimating these results can be essential to confirm the feasibility of the model. For this reason, we want to generate a tool that allows evaluating at least part of these data. Ideally, we also want to use it to profile future models without modifications.

Chapter 2

GoogLeNet CNN on GPC (Version 1)

The aim of this chapter is to analyze the design process of the first GoogLeNet CNN on GPC model, from the Pen and Pencil mapping to the application profiling. The last paragraph describes the limits of this model which will then be overcome by the new model in Chapter 5. The actual comparison between the two versions will be analyzed in Chapter 6.

2.1 Pen and Paper Mapping

The main challenge of this model was its size. To take advantage of the high scalability of the GPC architecture, it was decided to assign one CNN layer per core. The GoogLeNet CNN has 142 layers[1], so the grid must contain at least 142 cores.

Analyzing the GoogLeNet CNN structure in Figure 1.3, it is possible to notice that the first and last layers are connected in series, while the nine inception blocks in the middle repeat themself without any significant change. The inception block has the structure shown in Figure 2.1.

Considering all this, the real challenge has become finding the best mapping for the inception block. Two rules were used:

[1] 143 if we consider also the first layer called "data". However, the testbench already generates the input data so this layer implementation is not needed.

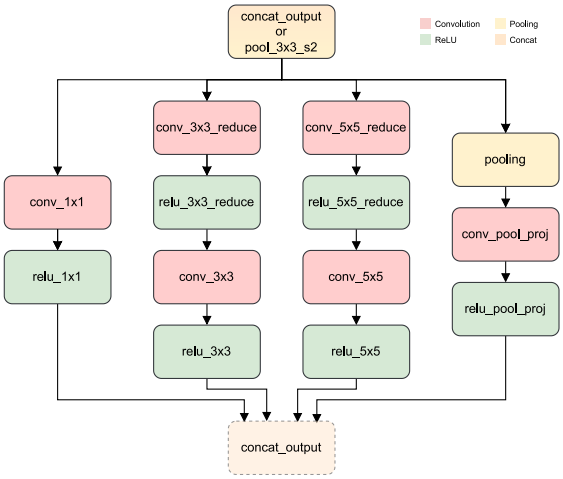

Figure 2.1. The structure of the inception blocks inside the GoogLeNet CNN, redrawn from [16].

1. Trying to not leave any unused cores to reduce area occupation.
2. The last core of the inception block should be placed close to the first core of the next block. In this way, repeated blocks will match perfectly.

Looking at Figure 2.1, it is possible to identify 4 *paths* connected in parallel: 1 formed by 2 layers, 1 by 3 layers and 2 by 4 layers. The last layer occurs to be part of the block only two times in the entire structure when the first layer of the next block is a pooling layer. This means that we have 7 blocks that contain 14 layers/cores and 2 that contain 15 layers/cores.

For the sake of simplicity, we can assume every block to contain *15 layers/cores*, this allows us to have a more regular and modular pattern at the minor cost of 7 extra cores.

2.1.1 MemTools

MemTools is a library written in C++ that allows a FIFO-like communication between adjacent cores [10]. The shared memory is divided into equal partitions depending on the required number of channels. Each partition is split into two parts: the counters and the queue. Every time some data is pushed inside the queue the *sent counter* is incremented, while the *received counter* is incremented when the data is popped. When the queue is full or empty, the operation remains blocked until a new pop or push from another core occurs.

Figure 2.2 shows the memory mapping organization. Both counters use four bytes, while the queue size can be adapted to the quantity of data to store. The designer can also

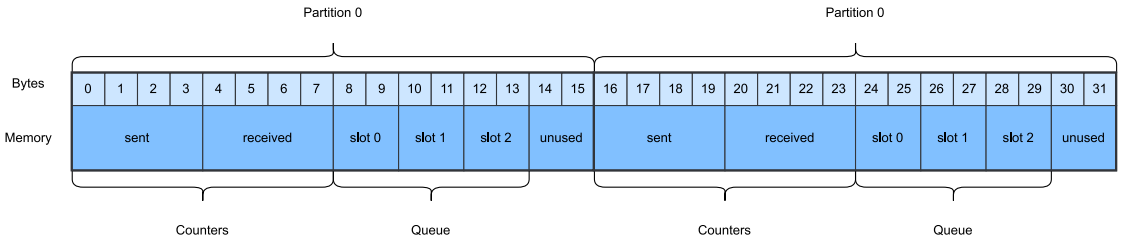

Figure 2.2. MemTools memory organization, redrawn from [10].

increase the number of partitions, but the memory size can limit its range.

MemTools Limits

The advantage of using partitions is that it avoids memory segmentation. Other than that, it comes with limitations.

- Every partition has the same maximum number of slots.
- Every slot has the same size in each partition.
- Even if only one partition is used, the user had to specify the partition id at every pop or push.

Overall, partitions could only work in specific cases.

Other limitations related to the MemTools library are:

- Push and pop parameters are redundant, this can lead to multiple typographical errors.
- The memory part occupied by the channel must be cleaned by the user before being used.
- In a FIFO-like structure, having indexable content is not needed, for this reason, having slots do not bring any benefit.

2.1.2 Fork and Merge

Due to the MemTools partitions limitations, it was decided to not use them and instead find an alternative. The solution was to specialize some core at splitting or combining data. These modules were called *Fork* and *Merge* and they are basically a deserializer and a serializer. The way they work is shown in Figure 2.3.

Figure 2.3. Fork and Merge operations with MemTools code.

The Fork is needed when the user wants to *push* data to more than one core using the same memory, while the Merge is needed when the user wants to *pop* data from more than one core using the same memory.
In the Fork module, the slot length of the input must be equal to the maximum slot length between all the outputs. In the Merge module, the slot length of the output must be equal to the maximum slot length between all the inputs.

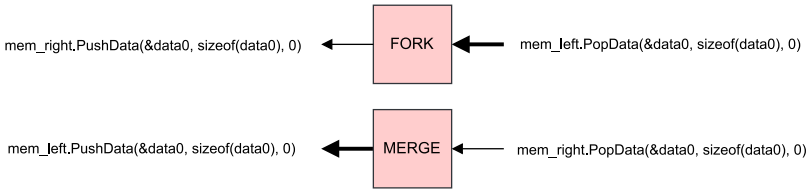

Figure 2.4. Fork and Merge used as forwarding module.

Both of the modules can be used as a "forwarding" module if only one input and one output are present, as shown in Figure 2.4.

2.1.3 Inception Block

Figure 2.5 shows the final version of the inception block that was used for this model. It is formed of 20 cores, 3 of which are empty, however, the 2 on the top part can be used by other layers.

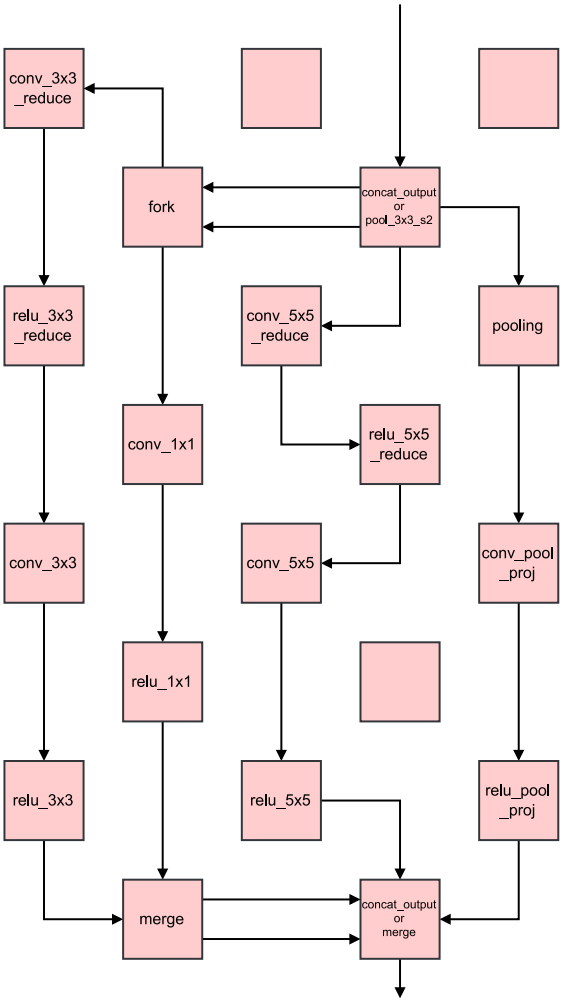

Figure 2.5. GoogLeNet CNN on GPC (version 1) inception block design.

Overall, this block requires only 5 cores more than the theoretical 15 due to the use of Forks and Merges modules.

Looking at Figure 2.5 it is possible to confirm that the two rules imposed for this design in Section 2.1 have been observed.

2.1.4 Complete structure

The spreadsheet with the complete mapping of the first GoogLeNet model is shown in Figure 2.6.
The three main parts were colored to give the reader a rough idea of the structure.

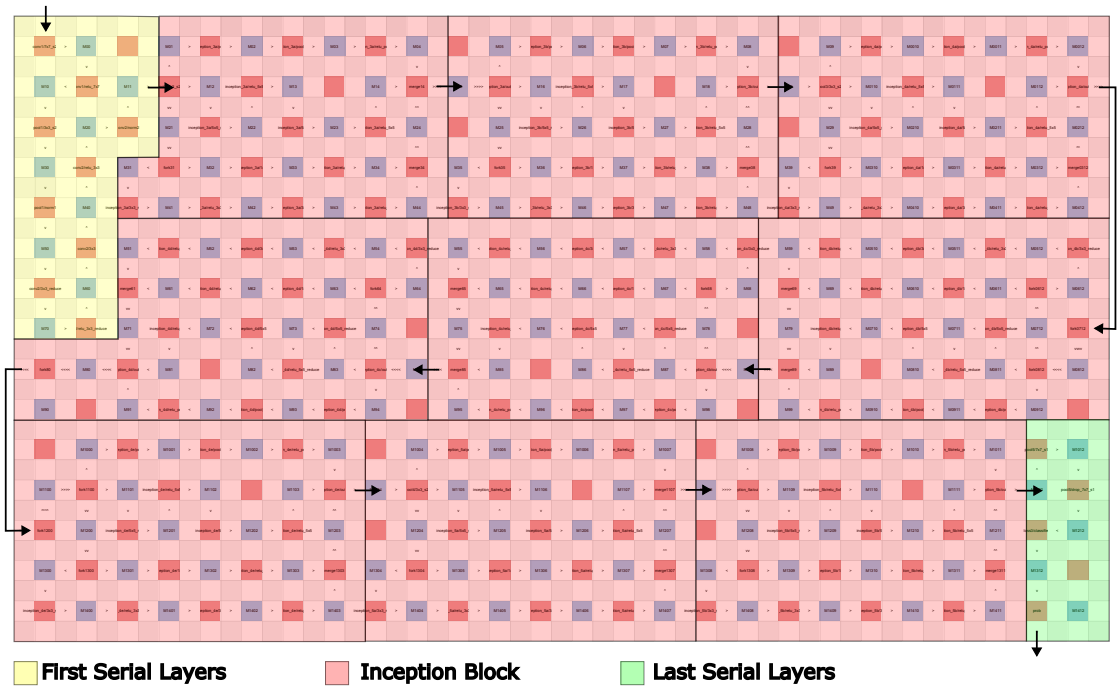

Figure 2.6. GoogLeNet CNN on GPC (version 1) structure.

The grid used measures 15 by 13 cells. Of the 195 available cores, 26 were not used. Some arrows indicate the direction of the stream of data through the grid. The input image is fed to the model from the top using the external memory. Then, the two memories to the sides shorten the path between inception blocks on different "rows". Finally, the generated array of probabilities is stored in the bottom external memory.

Inside the inception blocks (in red), the four paths of layers run in parallel, while the first (in yellow) and last (in green) layers of the CNN are connected in series.

2.2 Implementation

The implementation of the GoogLeNet model requires different steps to follow. Figure 2.7 shows the overall process used to automatically generate the model and profile the application. The following paragraphs explain in detail all the steps required.

2.2.1 Parsing

The GoogLeNet Caffe (Convolutional Architecture for Fast Feature Embedding) model was used to obtain pre-trained network parameters [14]. The model includes two files: a binary .caffemodel file that contains pre-trained parameters such as weights and biases and a human-readable .prototxt file that describes the network architecture.

The *parser.py* takes these two files as input and generates, among many other things, a Python dictionary containing all the required information about the layers, like inputs, outputs and other parameters. This parser is part of the Netspec generator used to automatically generate the SystemC model of the GoogLeNet [16].
Listing 2.1 shows the parsing result of the first Convolutional layer.

```
"conv1/7x7_s2": {
    "type": "Convolution",
    "num_output": 64,
    "pad": 3,
    "kernel": 7,
    "stride": 2,
    "dilation": 1,
    "group": 1,
    "weight": [
        64,
        3,
        7,
        7
    ],
    "bias": 64,
    "inputs": [
        "data"
    ],
    "outputs": [
        "conv1/relu_7x7"
    ],
    "input_shape": [
        [
            1,
            3,
            224,
            224
        ]
    ],
    "output_shape": [
        [
            1,
            64,
            112,
            112
        ]
    ]
}
```

Listing 2.1. First Convolutional layer description inside the layers Python dictionary

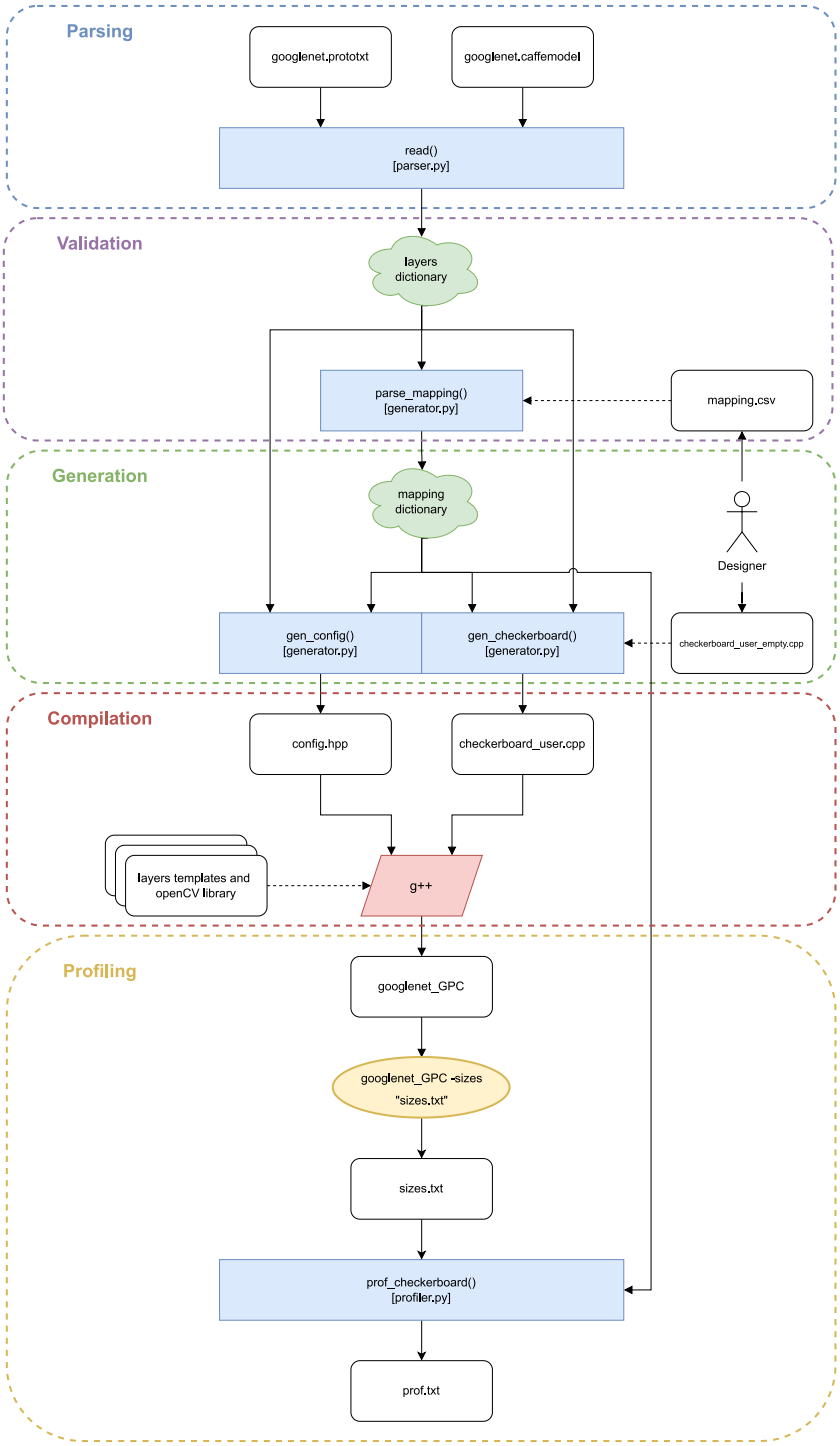

Figure 2.7. GoogLeNet CNN on GPC (version 1) complete implementation steps.

2.2.2 Mapping Validation

In the validation phase the layers dictionary, generated during parsing, is compared with the application mapping provided by the user.

Using Microsoft Excel as a CAD

Due to the fact that we are working with a grid-based structure, it was decided to use the .csv file format to describe the user mapping. The main advantage of this choice is that it is possible to visualize and/or modify .csv files inside spreadsheet editors like Microsoft Excel. Figure 2.8 shows how the first inception block can be described on the spreadsheet.

The red squares are the cores and must contain the name of the modules associated with them, while the blue squares are the memories and must contain their own id. The white squares are used to define the channels and the direction of the "arrows" represent the direction of the data stream. Finally, the grey squares should not contain anything.

The Validation Algorithm

The .cvs file is passed to the *parse_mapping()* function to validate the designed structure.

The algorithm iterates between the *output* channels of each core until the name of one of the connected modules correspond to the target name. The target is one of the output names written inside the .prototxt file. When the search finishes without finding the requested name, a "missing connection" error occurs. When a fork or a merge module is connected to the core, the algorithm checks its outputs recursively. If the name is found then the complete searching path is included inside the *mapping dictionary* along with the size of the exchanged data. Figure 2.9 shows a simplified version of the validation algorithm.

Also, a warning is printed on the terminal in case of extra input or output connections. This allows the identification of subtle mistakes that sometimes do not interfere with the application execution but increments memory occupation.

2.2.3 Code Generation

The main advantage of this framework is that it automatically generates parts of the application, in particular the `checkerboard_user.cpp` and the `config.hpp` files. The first one is used to describe the cores' behavior and the communication between them. The last one contains macros for each layer parameter and exchanged data size. This separation allows the user to easily modify parameters or sizes without the need to re-generate the entire structure, which most of the time remains the same.

The information required to complete the `checkerboard_user.cpp` are filled inside a

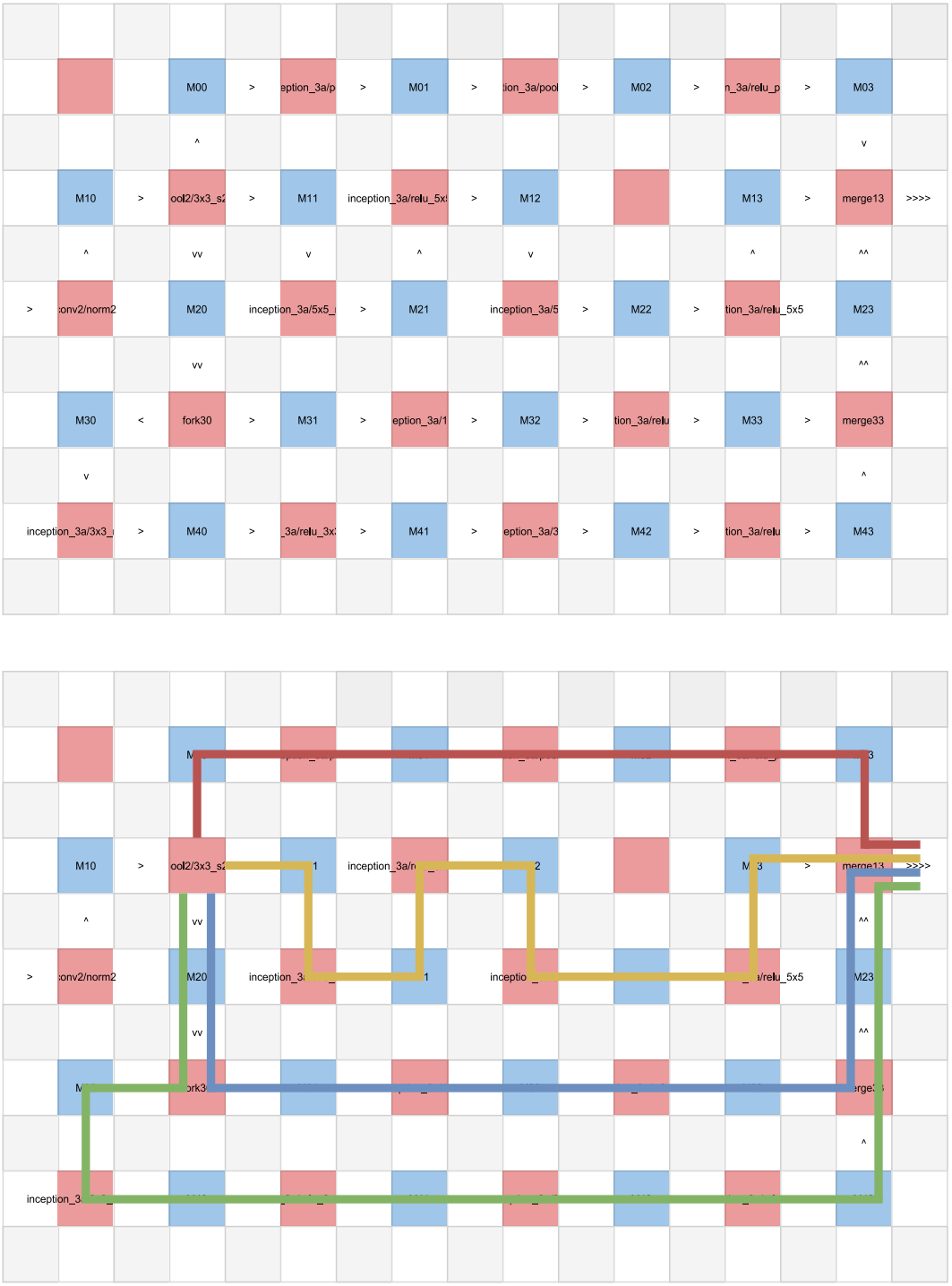

Figure 2.8. GoogLeNet CNN on GPC (version 1) spreadsheet description of the first inception block. The four paths are highlighted in the bottom image.

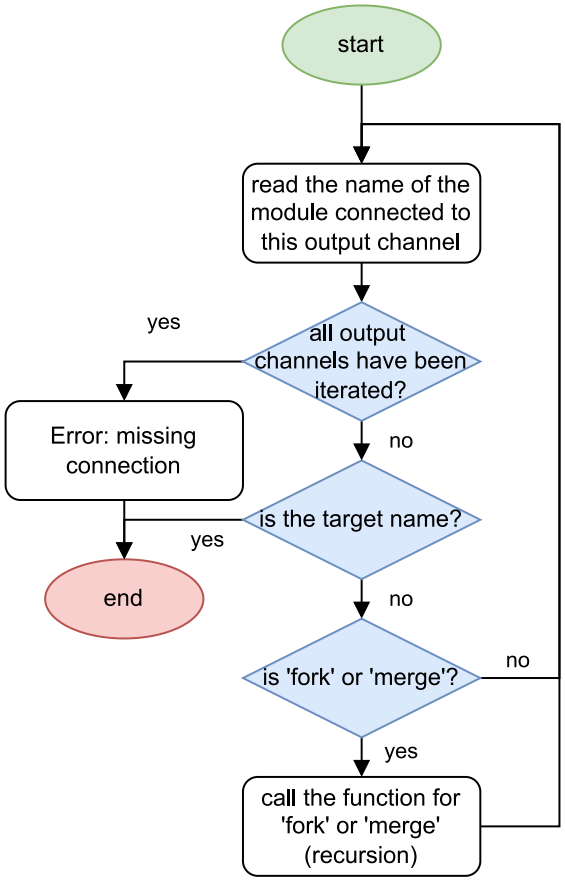

Figure 2.9. Validation algorithm.

reference file provided by user called `checkerboard_user_empty.cpp`. This gives the user a little bit more control over the generated file. The generation algorithm searches inside the `checkerboard_user_empty.cpp` file the *main()* method definitions and the constructors of all the used cores. The file is scanned line-by-line and Regular Expressions are used to identify the correct pattern.

Inside the constructor, the layer module will be instantiated and the thread stack size will be modified accordingly using the *set_stack_size()* function. While inside the *main()* method definition, the MemTools channels will be instantiated and the layer object will call the *run()* method, which will read the input data from the channels and generate the output results.

The layers are declared as template classes into separate header files and make use of the OpenCV library. The generator automatically includes all the layers classes into the `checkerboard_user.cpp` file. As an example, Listing 2.2 shows the implementation of

the first layer inside the `checkerboard_user.cpp` after being generated.

```
1  void Core00::main(){
2
3      // conv1_7x7_s2
4
5      MemTools memIn_up(CoreBus, sig_up, id_to_memAddress(-1, 0),
       OFF_CHIP_MEMORY_SIZE, 1, CONV1_7X7_S2_MEM_INP0_SLOT_SIZE, 1);
6      memIn_up.InitMem();
7
8      MemTools memOut_right(CoreBus, sig_right, id_to_memAddress(0, 0),
       ON_CHIP_MEMORY_SIZE, 1, CONV1_7X7_S2_MEM_OUT0_SLOT_SIZE, 1);
9      memOut_right.InitMem();
10
11     memIn_up.PopData(conv1_7x7_s2.inpVec[0].data,
       CONV1_7X7_S2_MEM_INP0_SLOT_SIZE, 0);
12
13     if(verbosity_level > 0) {
14         printf("[conv1/7x7_s2 (Core00)] %lu bytes popped\n", conv1_7x7_s2.
       inpVec[0].total()*conv1_7x7_s2.inpVec[0].elemSize());
15     }
16
17     if (dump_data) {
18         dumpData(STR(DUMP_DATA_FOLDER) "conv1_7x7_s2_in0.bin", conv1_7x7_s2.
       inpVec[0].data, CONV1_7X7_S2_MEM_INP0_SLOT_SIZE);
19     }
20
21     conv1_7x7_s2.run();
22
23     if(core_size) {
24         std::string str = "(Core00) conv1_7x7_s2 " + std::to_string(
       conv1_7x7_s2.size());
25         storeData(size_file_name.c_str(), str);
26     }
27
28     memOut_right.PushData(conv1_7x7_s2.outVec[0].data,
       CONV1_7X7_S2_MEM_OUT0_SLOT_SIZE, 0);
29
30     if(verbosity_level > 0) {
31         printf("[conv1/7x7_s2 (Core00)] %lu bytes pushed\n", conv1_7x7_s2.
       outVec[0].total()*conv1_7x7_s2.outVec[0].elemSize());
32     }
33
34     if (dump_data) {
35         dumpData(STR(DUMP_DATA_FOLDER) "conv1_7x7_s2_out0.bin", conv1_7x7_s2.
       outVec[0].data, CONV1_7X7_S2_MEM_OUT0_SLOT_SIZE);
36     }
37
38 }
```

Listing 2.2. Implementation of the conv1_7x7_s2 layer inside `checkerboard_user.cpp` file.

Looking at Listing 2.2 we can see that the generator produces some extra code needed to debug and/or profile the application. In particular, “verbose_lev”, “dump_data” and “core_size” are variables that can be set by the user launching the application with the options -v/-vv, -dump and -size. All the options and their meaning are explained in the next paragraph.

2.2.4 Compilation

The model was compiled by using the GCC C++ Compiler 9.4.0 and the provided Makefile. The model makes use of three external libraries: OpenCV 3.4.18, SystemC 2.3.3 language and TLM 2.0.5 (included inside the SystemC language).

Some options can be passed to the executable file by using the command line. The complete list is shown below:

- **-v** and **-vv** activate level 1 and level 2 verbosity. This allows the printing of debugging information on the terminal.
- **-img <path-to-img-file>** let the user choose the input image of the GoogLeNet CNN. By default, it is the ./img/space_shuttle.jpg
- **-dump** lets the application generate binary files containing the cores inputs and outputs FIFOs content during the simulation.
- **-sizes <path-to-size-file>** lets the application generate a human-readable file containing the intrinsic memory usage of the layer during the simulation. This file can be used to improve the profiling results by passing it to the *profiler.py*.

2.2.5 Profiling

The profiler aims to generate a report containing useful information about memory usage.

It requires the mapping dictionary generated in the validation phase and optionally the file generated by passing the *-sizes* option to the application before launching it. The first one contains information about the inputs and outputs channels and the exchanged data size, everything related to the *shared* memory usage. The last one contains the memory occupation of every single layer, also called *local* usage because this data is stored in the memory inside the same cell. The *total* memory usage is the sum of the *shared* and the *local* usages.

The results of the profiling will be discussed in Chapter 6.

2.3 Limits of the model

The first GoogLeNet CNN on GPC model shows that it is feasible to design large applications on the GPC architecture. However, this implementation is far from perfect. During the design process compromising was necessary more than once. In this section, we will describe the problems faced in the first model, which will be solved in the second version of the model, as explained in Chapter 5.

MemTools, Fork and Merge

MemTools limitations have been already discussed in Subsection 2.1.1. Avoiding partitions led to the use of Fork and Merge modules throughout the structure. Considering that the inception block requires 5 extra cores due to these modules and that inside the CNN 9 inception blocks are present, we could reduce by 45 the required number of cores. In percentage, we have a reduction of about 23% of the all grid area occupation by improving the multi-channel implementation.

Hard profiling

The data structures created by parsing the CNN information simplify the code generation, not the profiling. The mapping dictionary does not let the designer quickly obtain essential information, such as the grid size. This data needs to be searched inside the generated code, increasing the profiling complexity. The complete memory usage report requires the simulation of the application, which could be avoided by using an improved data structure. Without these difficulties, other types of reports would have been designed, such as one about application delays.

Microsoft Excel is not a CAD

Describing the structure using spreadsheet editors works well for simple designs, but when a core has multiple inputs or outputs is much more complex.

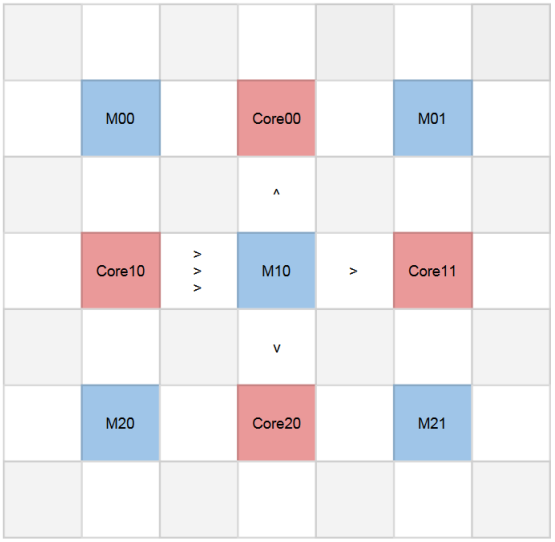

Figure 2.10. Spreadsheet unclear description.

Figure 2.10 shows a case in which the format used becomes unclear. In the example, we have three FIFOs: the first connects Core10 to Core00, the second Core10 to Core11 and

the third Core10 to Core20. Since Core10 has three outputs, we cannot know to which one the correct FIFO is attached.
A possible solution is to impose an output order from top to bottom, but we have to adjust Core10 accordingly every time, which becomes cumbersome.
For our model, this was possible because, during the validation phase, the algorithm automatically ordered every output. Without the network information provided by the Caffe model, this would be much more tricky.

Other than that, the spreadsheets allow us to visualize the data flow of the model, but it is not possible to edit the channels or modules parameters from there. For this reason, spreadsheets could be used during the first steps of the design process but not for debugging or fine-tuning, which makes their usage limited.

Chapter 3

MARI: Memory Access Resources and Interfaces library

MARI library is a novel alternative to the MemTools library described in Subsection 2.1.1.

The GPC architecture aims to solve the shared memory bottleneck problem [2], still present in modern multi-processor structures, using local shared memories. Lowering the congestion and the need for caches allows designers to increase the number of cores inside the chip, which leads to an increase in memory accesses.
Considering all that, formal communication protocols between cores are crucial for large applications like the GoogLeNet model.

MARI library aims to simplify interactions between cores by providing a set of high-level communication channel interfaces, which emulate the behavior of commonly used channels throughout a memory. In this thesis work, we will only present the FIFO, but other types of channels will be implemented in the future, like Stacks (LIFO).
This library also introduces the *Memory Interface* that virtually attaches the selected core to one of its surrounding memories.

The relation between channel interfaces and channels is comparable to memory interfaces and memories. Interfaces allows the user to alter the state of specific data structures through operations like *push()*, *pop()*, *read()* or *write()*. On the other hand, data structures like channels or memories are passive containers that store data in a precise format.

Channel interfaces require a memory interface by which they can communicate with the memory. When the user instantiates a new channel interface inside the core, a portion of

the shared memory is initialized to contain the channel. For the communication to work, all the cores in play must know the channel's exact length and location (offset) inside the memory.
MARI library distinguishes between input and output channel interfaces. For a FIFO channel, the input interface communicates with the FIFO head, while the output with its tail.

Figure 3.2 and Figure 3.1 show two views of a simple example of communication between cores through the use of FIFOs.

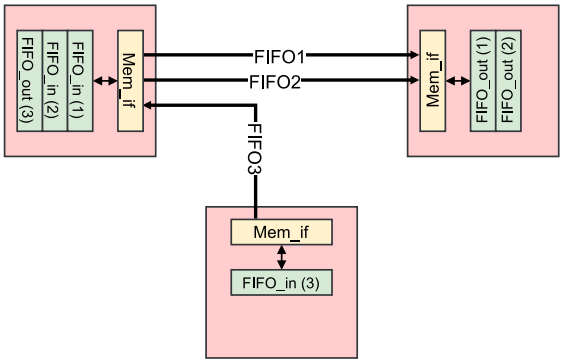

Figure 3.1. Example of communication between cores using MARI library, Channels View.

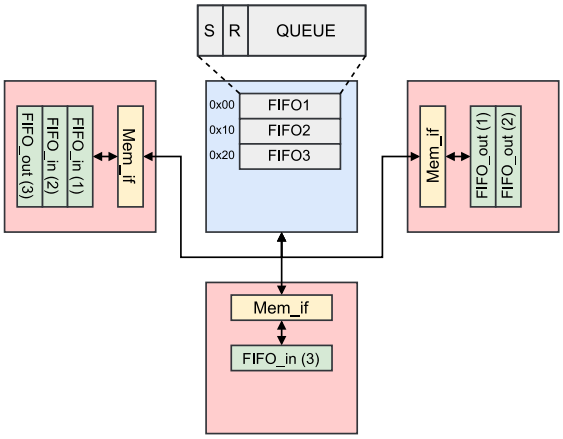

Figure 3.2. Example of communication between cores using MARI library, Memories View.

3.1 Memory Interfaces

The most important difference from its predecessor (MemTools) is that MARI introduces the concept of *Memory Inteface*.

In the MemTools library, creating more than one channel using the same memory is complex[1]. Partitions are sub-channels inside the object that "embed" the memory interface. MARI's solution is to *separate the channel from the memory interface*.

First, the user creates a memory interface of type *Mem_if* for each memory he wants to use. Then the memory interface is passed by reference to every instantiated channel interface that uses a portion of that memory for its channel. This process works well because, in reality, channels are an abstraction; there are no hardware FIFOs, just one memory. Moreover, creating memory interfaces take away from the user the tedious task of initializing every channel with memory-related parameters except the offset. However, in the MARI library, the offsets of the channels are simpler to investigate because they are relative to the start address of the memory interface to which they are connected. Listing 3.1 shows the parameters required to construct a Mem_if object.

```
1 Mem_if(tlm::tlm_initiator_socket<>& socket, sc_core::sc_event &int_sig,
      uint32_t startAddr);
2 void write(uint32_t id, uint32_t mem_offset, void *data, uint32_t len);
3 void write(uint32_t mem_offset, void *data, uint32_t len);
4 void read(uint32_t id, uint32_t mem_offset, void *data, uint32_t len);
5 void read(uint32_t mem_offset, void *data, uint32_t len);
```

Listing 3.1. Methods declarations of the Mem_if class inside the MARI library.

The first argument references the TLM2.0 initiator socket through which the interface communicates with the memory.
The second argument is a reference to the SystemC event called every time a channel interface changes the content of the memory, as an interrupt signal.
The last argument is the physical starting address assigned to the shared memory.

It is possible to read and write from memory through the use of the channels interfaces or by simply calling the *read()* and *write()* methods declared inside the Mem_if class, as shown in Listing 3.1.
Referencing a Mem_if object allows accessing a specific memory without passing all these parameters again. The reasons why *read()* and *write()* methods have multiple implementations will be explained in Section 3.3.

[1]It should be possible to create different channels in the same memory by adjusting their starting address accordingly. However, MemTools describe the *startAdress* parameter used to construct a channel as "the starting address of the memory unit" suggesting that its value should not change when referring to the same memory.

3.2 FIFO Interfaces

MARI library contains 3 type of FIFO interfaces: *FIFO_in*, *FIFO_out* and *FIFO_inout*.

- FIFO_in connects to the input of a FIFO. The user can push data through it and check if it is full.
- FIFO_out connects to the output of a FIFO. The user can pop data through it and check if it is empty.
- FIFO_inout connects to one side of a channel composed of two FIFOs in opposite directions. The user can push and pop data through this channel and check whether it is full or empty. Basically, it is a FIFO_in combined with a FIFO_out.

The way all FIFOs work is much simpler compared to MemTools.
MARI keeps the sent and received counters, but there are no partitions and slots. MARI's FIFOs store data in *bytes*, so it is up to the user to decide the size of each element inside the queue[2]. Listing 3.2 shows the parameters required to construct the three types of FIFO interface objects.

```
1  FIFO_in(Mem_if &mem_if, uint32_t mem_offset, uint32_t size);
2  FIFO_in(uint32_t id, Mem_if &mem_if, uint32_t mem_offset, uint32_t size);
3  void push(void* data, uint32_t len = 1);
4  bool full();
5
6  FIFO_out(Mem_if &mem_if, uint32_t mem_offset, uint32_t size);
7  FIFO_out(uint32_t id, Mem_if &mem_if, uint32_t mem_offset, uint32_t size);
8  void pop(void* data, uint32_t len = 1);
9  bool empty();
10
11 FIFO_inout(Mem_if &mem_if, uint32_t mem_offset, uint32_t size);
12 FIFO_inout(uint32_t id, Mem_if &mem_if, uint32_t mem_offset, uint32_t size);
```

Listing 3.2. MARI library FIFO interfaces constructors.

All the FIFO interface constructors require the reference to a memory interface, the location inside the memory expressed as the number of bytes from the starting address (offset)[3] and the size of the queue in bytes.

[2]If the length of the FIFO and the element size are multiple of 4 bytes there is a reduction of memory accesses, due to the 32-bit interface implementation of GPC memories.

[3]Zero is the first available offset in each memory.

The reasons why the constructors have multiple implementations will be explained in Section 3.3.

3.2.1 Ring Buffering

The mechanism in which data is pushed or popped inside the queue recall the one used for ring (or circular) buffers.
When the queue is empty, the FIFO interface writes/reads from the queue's first to last byte. Once it reaches the end, the writing/reading starts over from the first byte. When some bytes are pushed while the queue is full, the FIFO interface raises the interrupt event to inform the other cores that the queue is full and **waits** until new space is available.
On the other hand, if some bytes are popped while the queue is empty, the FIFO interface raises the interrupt event to inform the other cores that the queue is empty and **waits** until some data is written. Figure 3.3 shows an example of the ring buffering mechanism while pushing 10 bytes inside an 8-byte long FIFO.

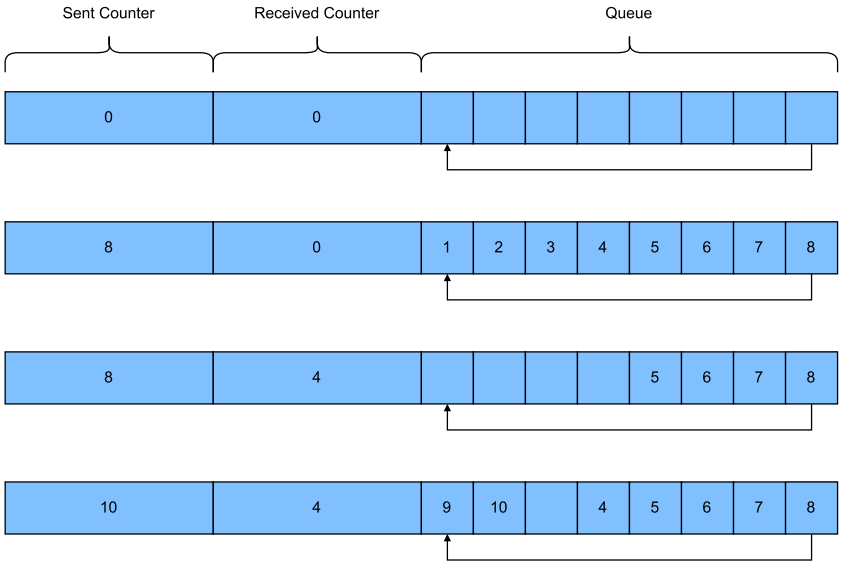

Figure 3.3. Example of ring buffering inside an 8-byte long FIFO.

Looking at Figure 3.3, we can see that the FIFO waits for the 4 bytes to be popped before pushing the 2 remaining bytes.

3.3 Profiling

MARI library allows the user to extract data about the memory accesses that can improve the profiling quality of the model.

Compiling the library with the "DEBUG_TIMING" macro defined enables the generation (during the simulation) of a binary file called *mari.log*, which contains the record of all the memory accesses.

Since, in many cases, the number of interactions with the memories could be high, the file has been encoded to reduce its size. Figure 3.4 shows how every access has been encoded.

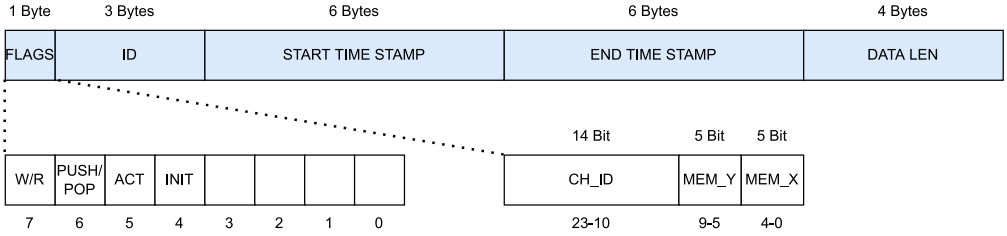

Figure 3.4. Memory access encoding inside the *mari.log* binary file.

Operation	Encoding	Operation Meaning
READ	0b0000XXXX	Reading through a memory interface.
WRITE	0b1000XXXX	Writing through a memory interface.
POP_READ	0b0000XXXX	Reading performed by a FIFO interface during a pop operation.
POP_WRITE	0b1000XXXX	Writing performed by a FIFO interface during a pop operation.
PUSH_READ	0b0100XXXX	Reading performed by a FIFO interface during a push operation.
PUSH_WRITE	0b1100XXXX	Writing performed by a FIFO interface during a push operation.
ACT_POP	0b0010XXXX	The operation that actually reads the queue performed by a FIFO interface during a pop.
ACT_PUSH	0b1110XXXX	The operation that actually writes the queue performed by a FIFO interface during a push.
INIT_POP	0b1001XXXX	Writing performed by a FIFO_out interface during the initialization.
INIT_PUSH	0b1101XXXX	Writing performed by a FIFO_in interface during the initialization.

Table 3.1. FLAGS field encoding used inside the mari.log file.

All the values stored inside the file use the big-endian ordering notation.
Starting from the right, the first 4 bytes are the length in bytes of the data written or read inside the memory.
Then there are the two 6-byte-long simulation time stamps in picoseconds taken at the end and the beginning of the memory access. Only the SystemC simulator scheduler advances

the simulation time, so these stamps are platform-independent.
The last 4 bytes are used for identification purposes. Table 3.1 shows the encoding used for the FLAGS field, which identifies the operation performed.

For the ID field, the user can choose the encoding he prefers. In the GoogLeNet on GPC model (version 2), the first 10 bits are used to identify the grid coordinates of the memory that has been written/read. Each coordinate requires 5 bits instead of just 4 because it accepts -1 and 17 values to identify external memories.
The last 14 bits are a unique number associated with each channel inside the memory.
The ID values can be passed as the first argument of the *read()* and *write()* methods or of the FIFO interfaces constructors, as shown in Listing 3.1 and Listing 3.2. The flags cannot be modified by the user. By default, the ID value is 0xFFFFFF, which stands for "NO ID".

The *mari.log* file will be an essential ingredient for the MapGL timing reports generation, as discussed in Chapter 4.

Chapter 4

MapGL

MapGL (Map Grid-based Layouts) is a CAD software written using the PyQt5 library [17]. It lets the user design custom applications for the GPC architecture. MapGL can auto-generate the SystemC model and create reports that give the user important information about memories usage and execution delays. The software structure was designed with modularity in mind so that it can be extended to other architectures in the future.

MapGL lets the user focus on the application design without caring about the specific programming language description, making the project highly reusable and portable. The entire MapGL design can be saved within a single JSON file.

Inside the MapGL editor, every design is defined by using *Modules* and *Channels*. A module is a component that can be attached to a core of the GPC to describe its behavior. In order to use it, the user must drag and drop the module on one of the cores. On the other hand, a channel allows two or more cores to communicate. The SystemC code generator fully uses the MARI library to implement channels.

4.1 Memories and Channels views

On launch, MapGL is set on Memories View by default as shown in Figure 4.1. The user can then switch between views using the two buttons on top.

In *Memories View*, the user can look at a more "realistic" representation of the structure. The blue squares represent the memories, while the red squares the cores. Both memories and cores contain parameters that the user can adjust. To do this, the component that has to be modified must be selected by clicking on it. By doing that, its parameters list will appear on the Parameters tab on the right and the user will be able to modify it. On

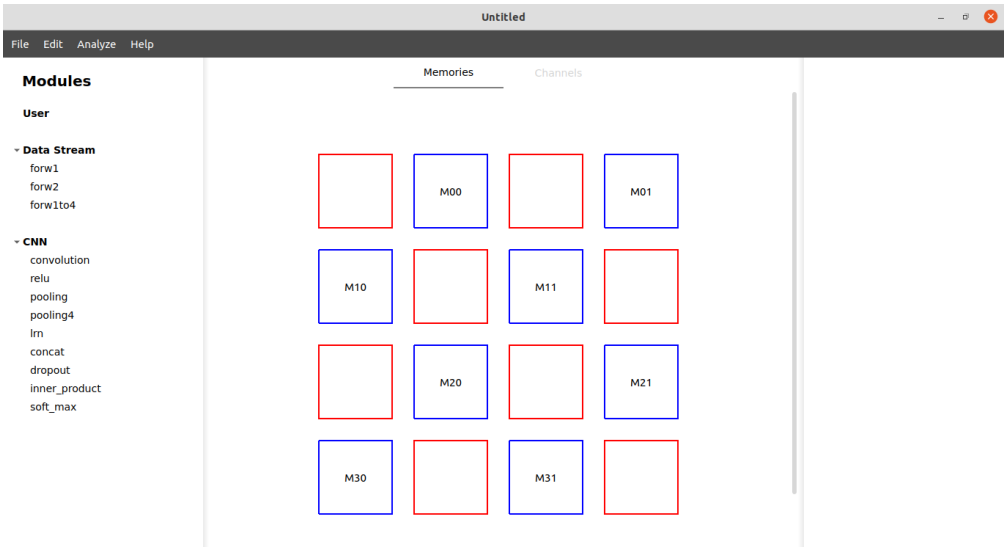

Figure 4.1. MapGL startup window.

the other hand, by clicking on the white background, it is possible to edit the parameters of the GPC itself, like grid width and height.

In *Channels View*, the user can visualize all the underlying interconnections between cores. Here, new channels can be created by clicking on one of the four memory interfaces that surround each core and then clicking again on a different memory interface. This action will render a black arrow between the two interfaces, representing the newly created channel. This operation will abort if the two interfaces are not connected to the same memory.
A channel can be deleted by simply clicking on it and pressing "Delete" on the keyboard. As for the Memory view, the user can modify core or channel parameters through the Params tab after clicking on one of these components.

In both views, it is possible to copy and paste core or channel parameters. To do that, the user has to first select a core or a channel by clicking on it and then press "Ctrl+C" on the keyboard to copy its parameters into the internal clipboard. Then, the user has to select another existing core or channel and press "Ctrl+V" to update all its parameters with the ones copied.

On both views, zooming is possible by using the mouse wheel or pressing "Ctrl+Plus" or "Ctrl+Minus" on the keyboard. The architecture can be auto-fitted into the view by pressing "Ctrl+F".

4.2 Modules

A *module* in MapGL describes the behavior of a core.
Every core can have just one module assigned to it, but it is possible to use the same module for different cores. The three main characteristics that every module should have are:

- A *name*, required for identification.
- Some *parameters*, which increase customization and reusability of the Module.
- Some *dependencies*, the files that contain the Module's declaration and definition.

How the module is defined depends on the exported project language. At the moment, MapGL only allows generating SystemC projects[1], so modules must be defined using C++ function templates.

Listing 4.1 shows one of the possible definitions of an adder as a MapGL module:

```
1  template<uint8_t bytes_size>
2  void adder(FIFO_out& in_a, FIFO_out& in_b, FIFO_in& out_sum) {
3
4      // getting the inputs
5      uint64_t a, b;
6      in_a.pop(&a, bytes_size);
7      in_b.pop(&b, bytes_size);
8
9      // adding together a and b
10     uint64_t sum = a + b;
11
12     // generating the output
13     out_sum.push(&sum, bytes_size);
14
15 }
```

Listing 4.1. Example of an adder MapGL module implementation.

For SystemC-based projects, the module is the code a core executes during the application simulation. In this specific example, Listing 4.1 makes the processor behave like an adder. At this stage of development, the designer does not need to know the hardware implementation of the core's processor. This simplification reduces the simulation time and speeds up the design process.

[1] In the future MapGL may support VHDL export.

4.2.1 How to create a module

In this paragraph, the adder MapGL module will be used as a reference to explain the process of creating a module from scratch.

The first step is to describe in C/C++ the main algorithm that our core will execute. Then, we need to put it inside the definition of a function template or a regular function, as shown in Listing 4.1. *The function's name will be our module's name.*

Exported SystemC project uses MARI library FIFOs to handle cores' communications. Our newly created module has to use the FIFOs interfaces (instantiated inside its core) to exchange data with the surrounding cores. To do that, we must pass our function the references to those interfaces. The main advantage of using references is that it detaches the FIFO interface from the module implementation. In other words, we can freely adjust the channel parameters without touching the module to which it is bound.

The *FIFO_out* interfaces are the inputs of the module, while the *FIFO_in* interfaces are the outputs[2]. Looking at Listing 4.1 we can see that the *pop()* and *push()* methods are called in the beginning and end of our function to collect and transmit data.

The final step is to increase the customization of our module by adding extra parameters[3]. We can pass that to our function, as we did for FIFO interfaces, or we can use the template parameter list. MapGL editor always treats module parameters like **constants**, so it is recommended to use template parameters when this is possible. For example, in Listing 4.1, the integer type of the "bit_size" parameter allows it to be a template parameter. On the other hand, FIFOs interfaces cannot be template parameters because they are references, so they are passed as regular function arguments.

To use our custom module, we need to import it inside the MapGL editor. However, MapGL cannot import .hpp or .cpp files directly[4]. We first need to create a JSON file containing our module's name and parameters, plus the *relative* paths to the .hpp and .cpp files in which our module is defined.
Listing 4.2 shows the JSON file implementation of the adder module in Listing 4.1.

```
1 {
2     "header_files" : ["./adder.hpp"],
3     "source_files" : ["./adder.cpp"],
4     "modules" : {
```

[2]For those familiar with VHDL, we are basically describing the entity of our module.

[3]For those familiar with VHDL, we are implementing generics in our module.

[4]In the future, a parser will be created to import these types of files.

```
5           "adder" : {
6               "in_a" : {
7                   "val" : null,
8                   "type" : "str"
9               },
10              "in_b" : {
11                  "val" : null,
12                  "type" : "str"
13              },
14              "out_sum" : {
15                  "val" : null,
16                  "type" : "str"
17              },
18              "bit_size" : {
19                  "val" : 1,
20                  "min" : 1,
21                  "max" : 8,
22                  "template" : true
23              }
24          }
25      }
26  }
```

Listing 4.2. Example of the JSON file used to import the adder module.

The key "modules" contains the parameters of all the modules defined inside the dependency files. However, in this example, it is just the adder module.
We can add attributes to each parameter, enabling unique behaviors inside the MapGL editor. For example, in Listing 4.2, the "bit_size" value is restricted between 1 and 8 by using "min" and "max" attributes. So inside the MapGL editor, the user cannot modify the "bit_size" value outside this range.

The "val" attribute is the parameter default value and must always be present. At the moment, its value can only be one of three types: integer (int), floating-point (float) or string (str).
The "val" attribute can also be null, but MapGL will raise an error if the user exports the project while at least one parameter has its value null. This solution prevents the user from leaving critical parameters unset, like names of input or output channels, which would cause compiling errors.
The MapGL editor can detect a parameter type by reading its value. However, if its default value is null, the detection will not work, so the user must specify a type for that parameter by using the "type" attribute.

The complete list of all attributes is shown below:

- **val** sets the parameter's default value and must always be present. Its value can be null, an integer, a floating-point number or a string.
- **type** is only required when the *val* attribute is null. The accepted values are "int", "float" and "str".

- **min** sets the lower limit of the parameter value. By default, this attribute is equal to minus infinite.
- **max** sets the upper limit of the parameter value. By default, this attribute is equal to plus infinite.
- **readonly** does not allow the user to change the parameter's value when set to true. Even if it is not modifiable, the value still appears in the MapGL editor. By default, this attribute is equal to false.
- **template** tells the code generator to put the parameter's value inside the template parameter list when set to true. Otherwise, it will be passed to the function as an argument. By default, this attribute is equal to false.
- **size** contains the memory occupation in bytes of the module. Its value will be used during the generation of the memories usage report. By default, this attribute is equal to zero.
- **label** is the identification name that the MapGL editor shows for this specific module instance. The original name of the module will be used when its value is not set. By default, this attribute is empty.

After the JSON file importation, our custom module would appear on the Modules tab on the left, under the "User" section. To use it, the designer has to drag and drop it inside one of the cores blocks in red.

4.3 Memory Structure

The MapGL editor automatically handles the structure and order in which the channels are stored inside the memories. This automatization prevents memory segmentation, but sometimes, it is necessary to visualize and manually change the memory mapping.

The memory structure window allows the designer to visualize and change the order in which the channels are stored without generating unwanted "holes" inside the memory.

The window can be opened by double-clicking on a memory or a channel. Figure 4.2 shows a possible layout of one of the GPC memories displayed through the help of the Memory Structure window.

Looking at Figure 4.2, we notice how the table represents the memory while the rows are the continuous channels sorted by offset. When the size of one channel is modified, its relative memory addresses shrink accordingly. Then, the offset of all the other channels inside the same memory is adjusted to avoid segmentation.
All the rows can be moved inside the table by drag-and-drop, allowing the designer to change channel ordering.

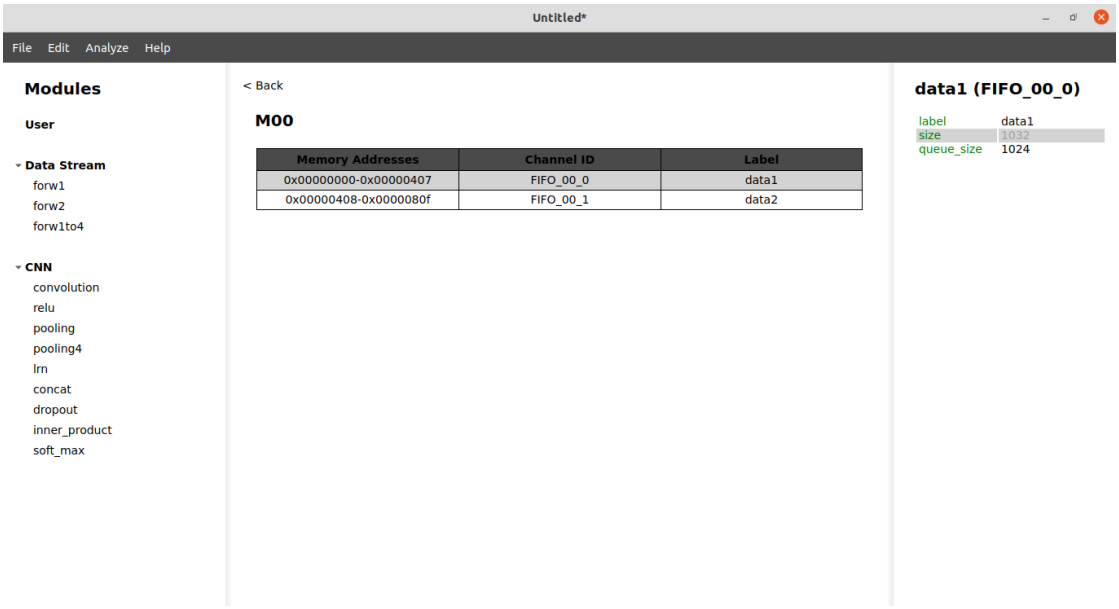

Figure 4.2. MapGL Memory Structure window.

4.4 Project Export

The MapGL editor allows designers to convert their mapping into actual models by automatically generating all the necessary files for the project. In this thesis work, we will analyze the SystemC export process because it is the only one that MapGL supports at the moment.

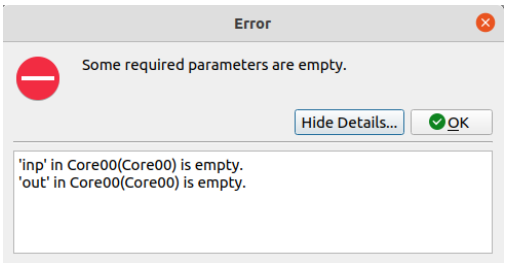

Figure 4.3. Error dialog that appears when at least one parameter has still the "val" attribute null.

Before generating the code, a check is performed to control that all the parameters have values different from null. If not, the dialog shown in Figure 4.3 pops up and the export stops. In the dialog, MapGL tells the designer which parameters must be set. If the control does not find any errors, another window allows the user to choose the output folder in which the SystemC project will be generated.

The list below shows the steps that the generator follows to construct the project:

1. Create the `./src`, `./inc` and `./obj` folders.
2. Create copies of the `checkerboard_arch.cpp`, `mari.cpp` and `std_modules.cpp` files inside the `./src` folder.
3. Create copies of the `checkerboard_arch.h`, `mari.hpp`, `std_modules.hpp` and `opencv.hpp` files inside the `./inc` folder.
4. Create a copy of the `top.cpp` file inside the output folder and rename the file has `tb_<project_name>.cpp`.
5. Create a copy of the dependencies source files inside the `./src` folder and of the dependencies header files inside the `./inc` folder.
6. Generate the `config.hpp` file inside the `./inc` folder.
7. Generate the `checkerboard_user.hpp` file inside the `./inc` folder.
8. Generate the `checkerboard_user.cpp` file inside the `./src` folder.
9. Generate the Makefile inside the output folder.

The `checkerboard_arch.cpp` and `checkerboard_arch.h` files contains the SystemC model description of the GPC architecture.

The `mari.cpp` and `mari.h` files contains the MARI library interfaces used for the channels implementation.

The `std_modules.cpp` and `std_modules.hpp` files contain the SystemC MapGL standard modules description. These modules are the ones that appear on the Modules tab by default on MapGL startup. At the moment, they are all embedded into every exported project automatically.

The `opencv.hpp` file contains all the functions and includes related to the OpenCV library. Currently, it is embedded in every exported project automatically because the CNN MapGL standard modules use it.

The `top.cpp` file contains the definition of the top model, a stimulus module, a monitor module, the *sc_main()* and a *DumpMemory()* function useful for debugging purposes. This file aims to be a bare-bone testbench implementation that allows the designer to start simulating the application easily.

After the export, the paths to the SystemC and OpenCV libraries must be adjusted manually inside the `Makefile` to make it work properly. Then, the SystemC project can be compiled using the command "make all", which will generate the application's executable.

4.4.1 Dependencies

The MapGL editor automatically handles the dependency files passed during the importation of custom modules. However, sometimes can be useful to visualize and manually add or remove dependencies to the project.

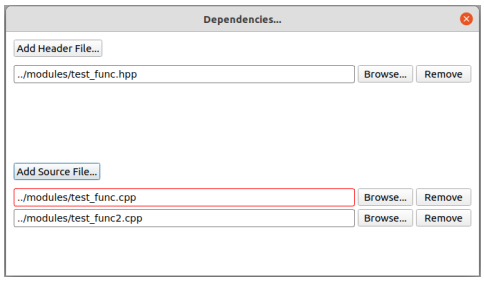

Figure 4.4. Dependencies window.

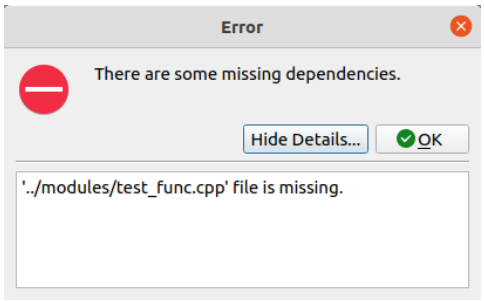

Figure 4.5. Error dialog that appears when at least one dependency file cannot be found.

The Dependencies window shown in Figure 4.4 allows the designer to list and manage third-party files.
Since, at the moment, only SystemC export is allowed, the dependency files can be of two types: C/C++ header or source files. The designer must press the respective "Add" button to add one of the two. A window will allow the user to select the dependency file. Then, a new row will appear, containing the path to the chosen file relative to the project location.
By pressing the "Remove" and "Browse..." buttons, it is possible to remove or change the file's location.
When the box's border is red, MapGL could not find the file. In this case, the solution is to modify the path using the "Browse..." button or to remove it. If neither of the two things can be done, the user will be unable to export the project as the error message shown in Figure 4.5 will appear.

4.5 Profiling

Once the mapping has been completed, MapGL allows designers to perform two types of analysis. The *memories usage analysis* requires just the information provided during the design process. On the other hand, the *timing analysis* needs the SystemC project to be exported and compiled. In other words, the first does not require the model to be simulated, while the second does.

4.5.1 Memories Usage Analysis

The *memories usage analysis* evaluates how much each memory is occupied by channels and cores (modules) data. It can also be used to check if memory is overflowing, meaning that the stored data size is larger than the memory itself.

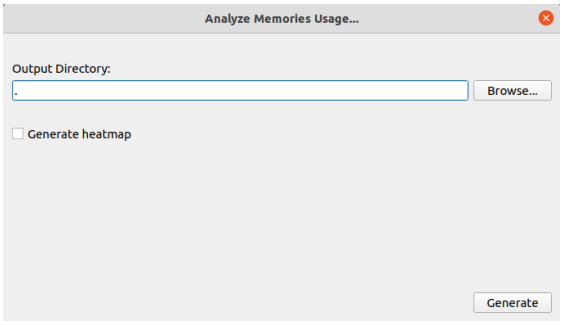

Figure 4.6. Memories Usage Analysis window.

The Memories Usage Analysis window, shown in Figure 4.6, lets the user configure some of the analysis parameters.
The "Output Directory" field contains the relative path to the folder in which all the files generated during the analysis will be placed. In the example, the output directory corresponds to the project folder.
The checkbox "Generate heatmap" lets the user decide whether to generate the model's heatmap image or not. The Memories usage heatmap allows the user to roughly estimate how much data is contained inside each memory.

The analysis will start immediately after pressing the "Generate" button on the bottom-right of the window. It usually takes only a few milliseconds to be performed. Then, a report file containing all the detailed information about the memories usages will be generated in the output directory, along with the heatmap image if the checkbox was checked.

Memories Usage Report

An example of a memories usage report is shown in Listing 4.3.
The preamble presents general information about the project.
The Summary section shows the analysis results performed on the overall structure.
The Memories usage section shows the list of memories sorted by the most to least occupied. The "Core usage" represents the portion of memory required by the core on the same cell. On the other hand, the "Challels usage" represents the combination of all the parts of memory used by its channels. Finally, the "Total" is the sum of the core and channels usages. When the "Total" value is higher than the actual memory size, the term "OVF" will appear next to it and an error message will be shown on the first line of the file. In this way, the user could quickly identify memory overflows.
Two other sections list the same data from the core's and channels' perspectives sorted by memory occupation.

```
1  ******************************************************
2  * MEMORIES USAGE REPORT
3  *
4  * Project : canny_GPC.json
5  * Version : 1.0.0
6  * Date    : sab ott 01 12:51:05 2022
7  *
8  * Grid Width        : 2
9  * Grid Height       : 4
10 * Memories Size     : 134217728 bytes
11 ******************************************************
12
13 ******************************************************
14 * Summary
15 ******************************************************
16 Total Cores Memories Usage      : 2819563 bytes
17 Total Channels Memories Usage   : 691456 bytes
18 Total Memories Usage            : 3511019 bytes
19 Most Used Memory by Cores       : Mem30 (Core Memory Usage: 614417 bytes)
20 Most Used Memory by Channels    : Mem20 (Channels Memory Usage: 153616 bytes)
21
22 ******************************************************
23 * Memories Usage
24 ******************************************************
25 Memory ID:       Core Usage:      Channels Usage:  Total:
26 Mem30            614417           153616           768033 bytes
27 Mem20            460812           153616           614428 bytes
28 Mem31            438378           153616           591994 bytes
29 Mem10            460912           76808            537720 bytes
30 Mem21            460820           76808            537628 bytes
31 Mem01            384112           0                384112 bytes
32 Mem11            0                76808            76808 bytes
33 Mem00            112              184              296 bytes
34
35 ******************************************************
36 * Cores Memories Usage
37 ******************************************************
38 Core ID:         Module:                Memory Usage:
39 Core30           non_max_supp           614417 bytes (Mem30)
40 Core10           blur_y                 460912 bytes (Mem10)
41 Core21           magnitude_x_y          460820 bytes (Mem21)
42 Core20           derivative_x_y         460812 bytes (Mem20)
43 Core31           apply_hysteresis       438378 bytes (Mem31)
44 Core01           blur_x                 384112 bytes (Mem01)
45 Core00           make_gaussian_kernel   112 bytes (Mem00)
46 Core11                                  0 bytes (Mem11)
47
48 ******************************************************
49 * Channels Memories Usage
50 ******************************************************
51 Channel ID:              Memory Usage:
52 smoothedim (FIFO_10_0)   76808 bytes (Mem10)
53 tempim (FIFO_11_0)       76808 bytes (Mem11)
54 delta_x1 (FIFO_20_0)     76808 bytes (Mem20)
55 delta_y1 (FIFO_20_1)     76808 bytes (Mem20)
56 magnitude1 (FIFO_21_0)   76808 bytes (Mem21)
57 delta_x2 (FIFO_30_0)     76808 bytes (Mem30)
58 delta_y2 (FIFO_30_1)     76808 bytes (Mem30)
59 magnitude2 (FIFO_31_0)   76808 bytes (Mem31)
60 nms (FIFO_31_1)          76808 bytes (Mem31)
61 kernel2 (FIFO_00_0)      92 bytes (Mem00)
62 kernel1 (FIFO_00_1)      92 bytes (Mem00)
```

Listing 4.3. Example of Memories usage analysis report.

4.5.2 Timing Analysis

The *timing analysis* allows the designer to estimate channels' and modules' delays. This analysis gives a rough idea of the application performance, but the results are only as accurate as the delay values that are provided to the simulator since MapGL works with high-level models.

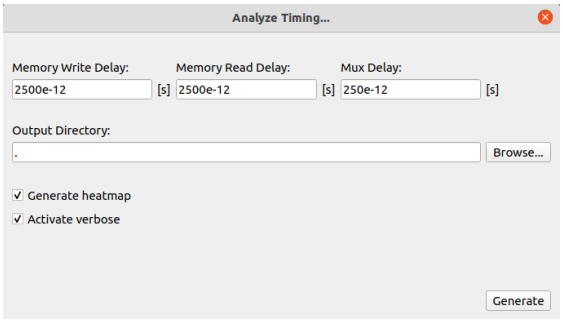

Figure 4.7. Timing Analysis window.

The Timing Analysis window, shown in Figure 4.7, lets the user configure some of the analysis parameters.
The first three fields that can be adjusted are the local memories read and write delays and the multiplexers propagation delay. These values can be chosen based on real hardware components, giving more accurate results or arbitrarily chosen ones.
The "Output Directory" field contains the relative path to the folder in which all the generated files will be located.
The "Generate heatmap" checkbox lets the designer decide whether to generate the model delays heatmaps. In this case, the analysis will create three heatmaps that show the contribution of the channels, cores and idles delays.
Finally, the "Activate verbose" checkbox lets the designer decide whether to print debugging messages to the terminal.

When the "Generate" button is clicked, a dialog will pop up, allowing the designer to choose the exported SystemC project location. The reason is that the timing analysis uses the SystemC project simulation to get the data required to calculate the final results. After selecting the folder, MapGL will make a copy of the project. Inside the copied folder, the application will be compiled and executed in debug mode, which generates the *mari.log* file described in Section 3.3. For this reason, before performing the timing analysis, the designer must ensure that the project compiles and executes without errors.

Using the time stamps of all the memory accesses inside the *mari.log* file, MapGL can reconstruct the delays of channels and then modules.

During a FIFO communication the channel can enter into two different phases: *idle* or *execution*. The channel is in the idle phase when it is waiting for cores to push or pop data, while it is in the execution phase when the channel reads or writes the memory. The times spent into these phases are called *idle delay* and *real execution delay*.
The sum of these two delays is called *execution delay* and, by definition, is the time interval from the start of the first memory access to the end of the last one for that specific channel.

Finally, we have the *latency*, which is the span between the start of the first memory access and the start of the *actual* memory read or write operation (defined in Section 3.3 as act_pop and act_push).
Figure 4.8 shows the life span of the channel communication with the time spent in each of the previously mentioned phases.

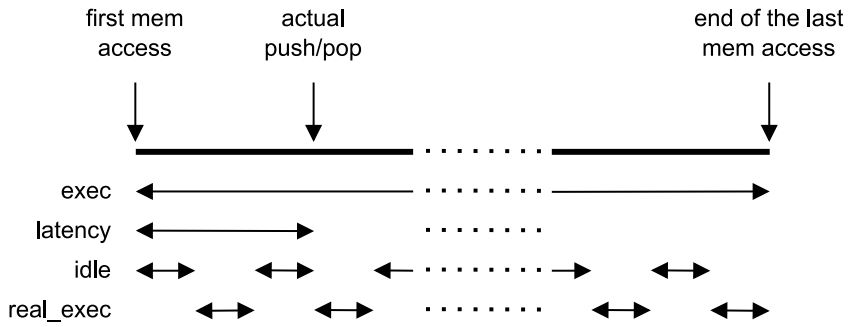

Figure 4.8. Channel communication life span with delays contributions.

On the other hand, modules have three phases: *idle*, *execution* or *communication*. The module is in the idle phase when at least one of its channels is in the idle phase and it is in the communication phase when at least one of its channels is in the execution phase. For the modules, the *idle delay* is the sum of the idle delays of its channels and the *channels delay* is the sum of the real execution delays of its channels.
When the module is not in one of these two phases, it is in the execution phase, in which it executes its algorithm. The time spent in this phase is called *real execution phase*.
The modules' *latency* is defined as the interval from the start of the simulation to the first pop operation performed by one of its channels.
The *execution delay* is the span between the first pop and the last push operations or it can also be defined as the sum of the idle, channels and real execution delays.
Figure 4.9 shows the life span of the module with the time spent in each of the previously mentioned phases.

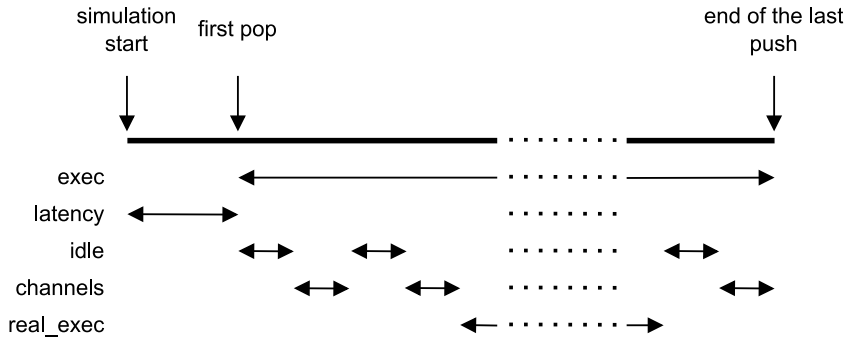

Figure 4.9. Module life span with delays contributions.

Timing Report

An example of a timing report is shown in Listing 4.4.
The initial part shows general information about the project and its parameters.
The Cores Delays section shows the previously described delays' values calculated for every core. The list is sorted by using the real execution delay.
On the other hand, the Channels Delays section shows the channels' communication delays. There are two lists, one for the push and one for the pop operation and they are sorted by using the real execution delay.

```
1  ********************************************************
2  * TIMING REPORT
3  *
4  * Project : canny_GPC.json
5  * Version : 1.0.0
6  * Date    : sab ott 01 12:55:43 2022
7  *
8  * Grid Width            : 2
9  * Grid Height           : 4
10 * Memory Write Delay    : 2.50e-09 s
11 * Memory Read Delay     : 2.50e-09 s
12 * Mux Delay             : 2.50e-10 s
13 ********************************************************
14
15 ********************************************************
16 * Cores Delays
17 ********************************************************
18 Core ID:  Module:               Latency:   Exec Delay:  Idle Delay:   Channels Delay:  Real Exec Delay:
19 Core10    blur_y                1.000e+06  8.174e+07    4.445e+07     2.881e+05        3.700e+07 ns
20 Core01    blur_x                1.000e+06  4.454e+07    9.307e+06     2.401e+05        3.500e+07 ns
21 Core30    non_max_supp          9.198e+07  3.467e+07    1.734e+07     3.361e+05        1.700e+07 ns
22 Core21    magnitude_x_y         9.184e+07  1.772e+07    3.360e+05     3.841e+05        1.700e+07 ns
23 Core31    apply_hysteresis      1.093e+08  3.050e+07    1.731e+07     1.921e+05        1.300e+07 ns
24 Core20    derivative_x_y        8.264e+07  9.720e+06    2.400e+05     4.801e+05        9.000e+06 ns
25 Core00    make_gaussian_kernel  1.000e+06  2.985e+02    1.425e+02     1.480e+02        8.000e+00 ns
26
27 ********************************************************
28 * Channels Delays
29 ********************************************************
30
31 PUSH:
32
33 Channel ID:              Latency:         Exec Delay:      Idle Delay:      Real Exec Delay:
34 tempim (FIFO_11_0)       4.521124e+07     3.360605e+05     1.440225e+05     1.920380e+05 ns
35 magnitude2 (FIFO_31_0)   1.094114e+08     1.440335e+05     4.801450e+04     9.601900e+04 ns
36 smoothedim (FIFO_10_0)   8.259530e+07     1.440265e+05     4.800750e+04     9.601900e+04 ns
37 delta_x1 (FIFO_20_0)     9.178734e+07     1.440265e+05     4.800750e+04     9.601900e+04 ns
38 delta_y1 (FIFO_20_1)     9.207539e+07     1.440265e+05     4.800750e+04     9.601900e+04 ns
39 magnitude1 (FIFO_21_0)   1.092674e+08     1.440265e+05     4.800750e+04     9.601900e+04 ns
40 delta_x2 (FIFO_30_0)     9.193136e+07     1.440265e+05     4.800750e+04     9.601900e+04 ns
41 delta_y2 (FIFO_30_1)     9.221942e+07     1.440265e+05     4.800750e+04     9.601900e+04 ns
42 imageout (FIFO_down_0)   1.396995e+08     1.152180e+05     6.720850e+04     4.800950e+04 ns
43 nms (FIFO_31_1)          1.266035e+08     4.800950e+04     0.000000e+00     4.800950e+04 ns
44 kernel1 (FIFO_00_1)      1.000008e+06     1.485000e+02     7.450000e+01     7.400000e+01 ns
45 kernel2 (FIFO_00_0)      1.000156e+06     1.420000e+02     6.800000e+01     7.400000e+01 ns
46
47
48 POP:
49
50 Channel ID:              Latency:         Exec Delay:      Idle Delay:      Real Exec Delay:
51 tempim (FIFO_11_0)       4.525924e+07     3.360635e+05     1.440225e+05     1.920410e+05 ns
52 magnitude2 (FIFO_31_0)   1.094595e+08     1.440385e+05     4.801650e+04     9.602200e+04 ns
53 delta_x2 (FIFO_30_0)     9.197937e+07     1.440295e+05     4.800750e+04     9.602200e+04 ns
54 delta_x1 (FIFO_20_0)     9.183534e+07     1.440295e+05     4.800750e+04     9.602200e+04 ns
55 delta_y2 (FIFO_30_1)     9.226742e+07     1.440295e+05     4.800750e+04     9.602200e+04 ns
56 magnitude1 (FIFO_21_0)   1.093154e+08     1.440295e+05     4.800750e+04     9.602200e+04 ns
57 delta_y1 (FIFO_20_1)     9.212339e+07     1.440295e+05     4.800750e+04     9.602200e+04 ns
58 smoothedim (FIFO_10_0)   8.264331e+07     1.440295e+05     4.800750e+04     9.602200e+04 ns
59 imagein (FIFO_up_0)      1.011521e+07     9.602350e+04     4.801100e+04     4.801250e+04 ns
60 nms (FIFO_31_1)          1.266515e+08     4.803600e+04     1.450000e+01     4.802150e+04 ns
61 kernel1 (FIFO_00_1)      1.000018e+06     2.065000e+02     1.295000e+02     7.700000e+01 ns
62 kernel2 (FIFO_00_0)      1.000205e+06     1.025000e+02     1.650000e+01     8.600000e+01 ns
```

Listing 4.4. Example of Timing analysis report.

Chapter 5

GoogLeNet CNN on GPC (Version 2)

This chapter aims to describe the design process that led to an improved version of the GoogLeNet on GPC model presented in Chapter 2. The MARI library and the MapGL editor, introduced in Chapter 3 and Chapter 4, cover a central role in the development.

The comparison of the experimental results obtained by the two versions will be analyzed in Chapter 6.

5.1 Inception Block

The inception block structure is the first significant improvement over the old model. By using MARI instead of the MapGL library, it is possible to remove the Fork and Merge Modules presented in Subsection 2.1.2. The simplified implementation of the inception block is shown in Figure 5.1.

This new version contains 15 cores without leaving any unused ones. According to the analysis made in Section 2.1, this is theoretically the most packed obtainable structure. In comparison, the first implementation used 20 cores, which, in percentage, are 25% more cores.

The last layer is placed close to the first core of the next inception block to increase its repeatability, like in the first version.

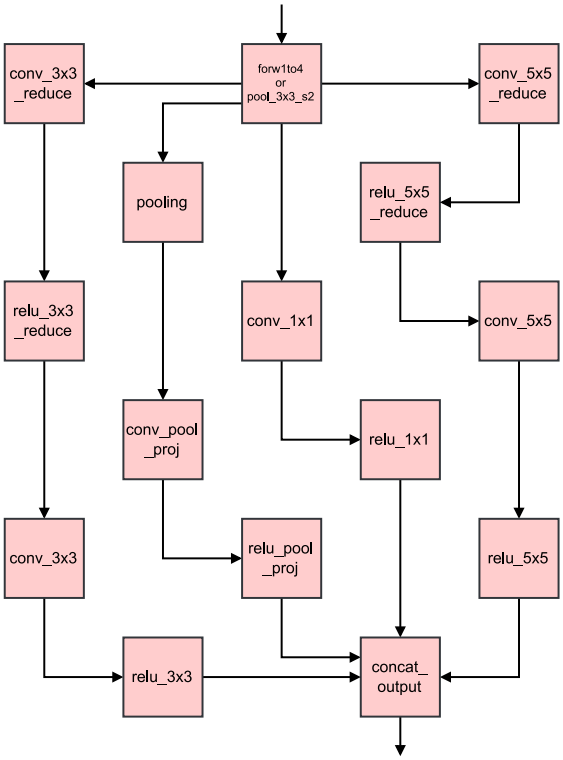

Figure 5.1. GoogLeNet CNN on GPC (version 2) inception block design.

5.2 Complete Structure

After completing the new inception block design, we started mapping the application using MapGL.
The mapping process lasted longer than the old spreadsheet implementation since, in this case, the designer must manually write the value of every module and channel parameter. The previous GoogLeNet on GPC version automatically filled the parameters, parsed from the .prototxt file, during the generation phase.
Unfortunately, MapGL does not include such a parser at the moment. However, the possibility of copying all the parameters of other completed modules came in handy.

The CNN layers used for the previous GoogLeNet model were converted into MapGL modules by following the steps described in Subsection 4.2.1. The process required squeezing the C++ template class definitions inside template functions.
Then, these modules were included in the standard modules package of the MapGL editor to be reused in future CNN-related applications.

Figure 5.2 shows the final MapGL mapping of the GoogLeNet on GPC (version 2) model.

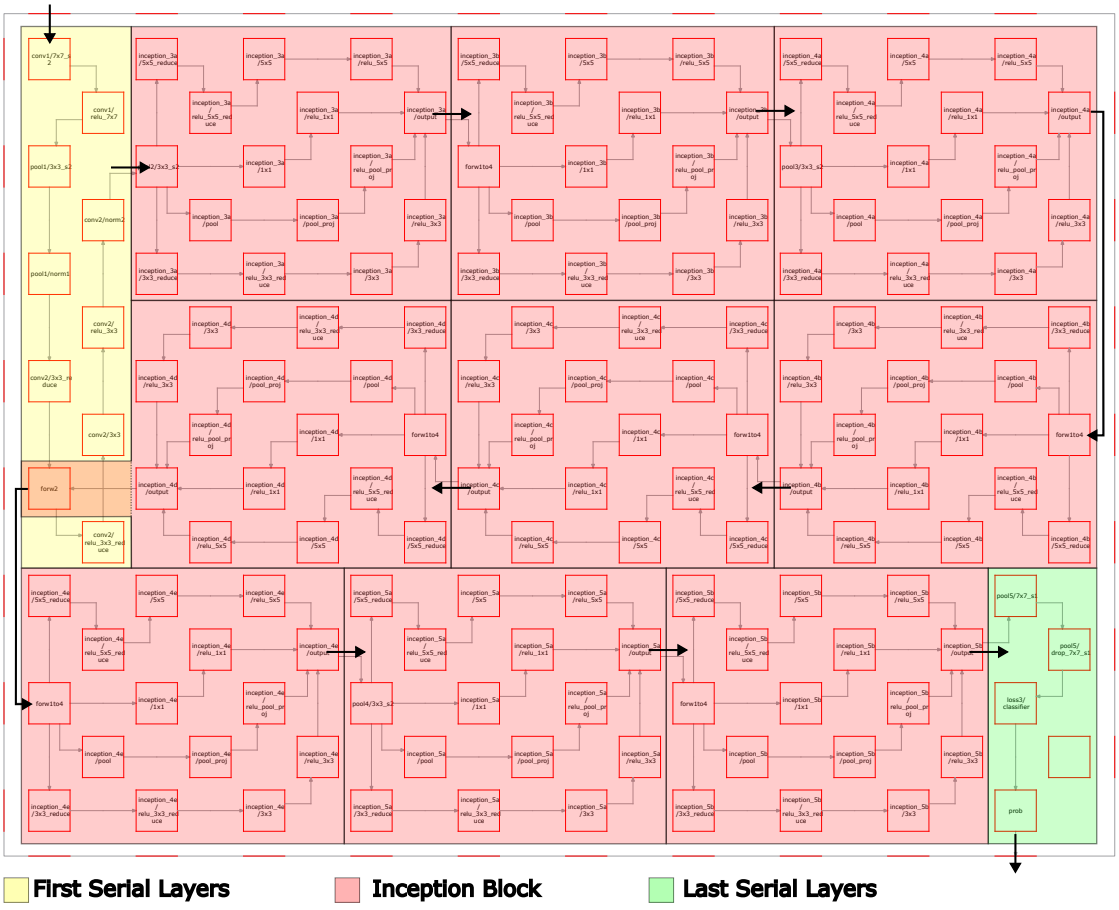

Figure 5.2. GoogLeNet CNN on GPC (version 2) structure.

This version makes use of a 15 by 10 grid. Of the 150 available cores, only one was not used. Overall, this solution contains only 7 cores more than the optimal implementation of 142 cores, as described in Section 2.1.
These extra cores were mainly used to allow all the inception blocks to have the same structure, which keeps the mapping simple.

As for the first GoogLeNet on GPC model, some colors were used to identify the three main parts of the structure. The first serial layers are in yellow, the last serial layers are in green, and the parallel layers inside the inception blocks are in red. In this version, we also used the top external memory to receive the input image, the two memories on the sides to shorten the path between cores and the bottom memory to store the output data.

It is worth mentioning that, in the left part, we use one core to "forward" the data from both the first layers and the sixth inception block. This solution is possible thanks to the versatility of the MARI library channels.

5.3 Profiling

The application's profiling was performed using the *memories usage* and *timing* analyses provided by the MapGL editor. These two analyses require the mapping generated in the previous paragraph.

In the first version of the GoogLeNet on GPC model, the FIFOs sizes were set equal to the input data size. In other words, every MemTools FIFO has just one slot.
This solution works well when the designer wants to maximize the application speed without caring about the occupation of memories.

The new GoogLeNet model can easily modify all the channel parameters, thanks to the use of MapGL. For this reason, it was decided to make two implementations of the same model: one *high-speed* and one *low-speed*. The high-speed implementation reduces the application delay by increasing memories usages. The low-speed implementation reduces memories usages and increases the application delay.
However, the first version of the model will be compared only to the high-speed implementation because both maximize the FIFOs sizes.

For all the modules used until now, it was assumed that their execution delay was equal to zero. At this level of abstraction, it is impossible to know all the timing specifications of the processor used in the final architecture.

The closest we can get to obtaining some rough values was by running the layers on a known processor and calculating the required execution delays.
The processor used is an Intel Xeon E3-1240 with a clock frequency of 3.4 GHz. The installed operating system in which the analyses were performed is CentOS 7.6 and simulation times were measured using */usr/bin/time* [16].
Delays were calculated 500 times using the same image to obtain more reliable results.

The results of all the analyses will be discussed in Chapter 6.

Chapter 6

Experiments and Results

In this chapter, we will analyze and compare the results of the two model versions.
All the results are just rough estimations since the application has been designed using a high-level of abstraction.

6.1 Version 1

The profiling of the first version of the GoogLeNet on GPC model generates a report containing all the memory occupations divided by *shared*, *local* and *total* usage. The shared usage is the memory's channels occupation, the local usage is the data occupation of the core inside the same cell and the total usage is the sum of the previous two.

Listing 6.1 shows an extract of the GoogLeNet CNN on GPC (version 1) report.

```
1  ************************************************
2  module: checkerboard
3  date: Thu Sep 22 18:44:49 2022
4  ************************************************
5
6  Grid Height : 15
7  Grid Width  : 13
8
9  ON-Chip Mem Size   : 0x00800000 (8388608) bytes
10 OFF-Chip Mem Size  : 0x20000000 (536870912) bytes
11
12 Total Cores Memories Usage      : 140212032 bytes
13 Total Channels Memories Usage   : 71774144 bytes
14 Total Memories Usage            : 211986176 bytes
15
16 The memory that has the higher shared usage : M00 3211264 bytes
17 The memory that has the higher local usage  : M10 6422528 bytes
18 The most used memory : M10 9633792 bytes
```

```
19
20 The core that uses local memory the most : conv1_relu_7x7(Core10) 6422528 bytes
21
22 Memories usage:      Shared          Local           Total
23 M10                  3211264         6422528         9633792          bytes
24 M00                  3211264         3851264         7062528          bytes
25 M20                  2408448         4014080         6422528          bytes
26 M21                  1204224         4816896         6021120          bytes
27 M30                  802816          4816896         5619712          bytes
```

Listing 6.1. Extract of the GoogLeNet CNN on GPC (version 1) profiling report.

Looking at Listing 6.1, we notice that our chosen mapping lets memory M10 overflow. To solve this problem, we can try to find a different mapping or manually modify the generated `checkerboard_user.cpp` file. In this case, we manually adjusted the FIFO (shared memory) size by dividing its length in half. The *conv1/relu_7x7* module needs to send the two halves of the data one by one, slightly increasing the overall delay but avoiding the memory overflow.
In the end, M10 shared memory occupation becomes 1605632 bytes, leading to a total memory occupation of 8028160 bytes.

This solution dedicates lots of memory to the channels to maximize the model's speed.

For this model, the timing profiling was too complex to implement.
Nevertheless, we can obtain the application delay by simply printing the time stamp of the SystemC simulator at the end of the simulation. Before doing that, the memories and multiplexer delays must be adjusted manually inside the generated code through the macros present inside the `checkerboard_user.cpp` file. The used values are discussed in Subsection 6.2.2.
When the simulation ended, the printed application delay was 158784755 ns.

6.2 Version 2

For the new model, we analyzed both the memory usage and the timing of the two implementations: *high-speed* and *low-speed*.
For the high-speed case, we set the length of the FIFOs equal to the size of the data to send/receive so that the operation takes just one transfer. For the low-speed case, we set the length of all the FIFOs to an arbitrary value of 64 bytes, except the first and the last, which use the top and bottom external memories. The reason is that the size of these memories is not related to the architecture.

Finally, another timing profiling was performed for both implementations, where each layer had an execution delay calculated as described in Section 5.3.

6.2.1 Memories Usage

Figure 6.1 shows the heatmaps generated after the memories usage analysis of the high-speed and low-speed implementations.

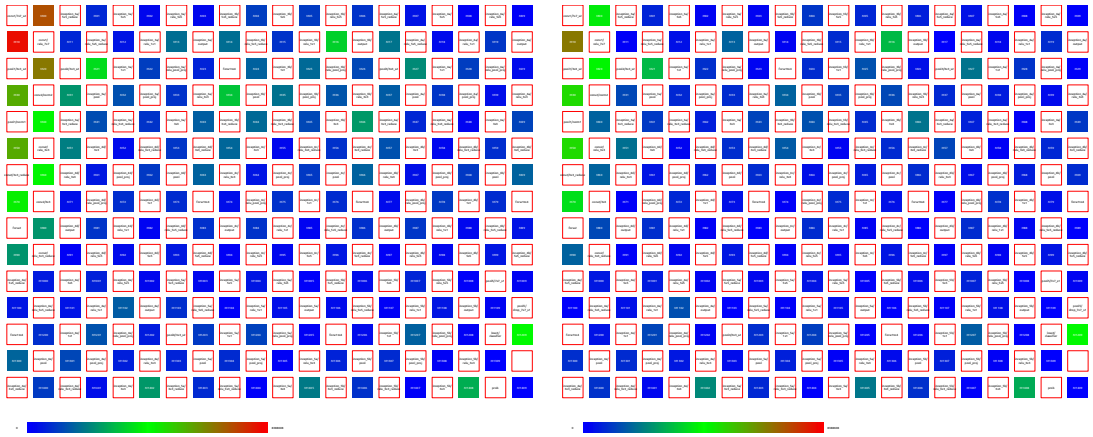

Figure 6.1. Memories usage analysis heatmaps of the high-speed (on the left) and low-speed (on the right) implementations of the GoogLeNet on GPC (version 2).

Looking at the top-left part of the two heatmaps, we notice that, as aspected, in the high-speed implementation, the memories are redder than in the low-speed one, meaning they are more used. We can also notice that the first layers of the GoogLeNet CNN are the ones that require bigger FIFOs or more space for the core's data. However, both implementations were designed without memory overflow, as confirmed by the two extracts of the memories usage reports shown in Listing 6.2 and Listing 6.3.

```
1  ******************************************************
2  * MEMORIES USAGE REPORT
3  *
4  * Project    : googlenet_GPC_high_speed.json
5  * Version    : 1.0.0
6  * Date       : sab ott 01 12:43:30 2022
7  *
8  * Grid Width         : 10
9  * Grid Height        : 15
10 * Memories Size      : 8388608 bytes
11 ******************************************************
12
13 ******************************************************
14 * Summary
15 ******************************************************
16 Total Cores Memories Usage       : 122867776 bytes
17 Total Channels Memories Usage    : 52152724 bytes
18 Total Memories Usage             : 175020500 bytes
19 Most Used Memory by Cores        : Mem10 (Core Memory Usage: 6422528 bytes)
20 Most Used Memory by Channels     : Mem00 (Channels Memory Usage: 3211272 bytes)
21
22 ******************************************************
23 * Memories Usage
```

```
24 ******************************************************
25 Memory ID:       Core Usage:      Channels Usage:  Total:
26 Mem10            6422528          1605644          8028172 bytes
27 Mem00            3851264          3211272          7062536 bytes
28 Mem20            4014080          2408456          6422536 bytes
29 Mem30            4816896          802824           5619720 bytes
30 Mem50            4816896          802824           5619720 bytes
```

Listing 6.2. Extract of the GoogLeNet CNN on GPC (version 2) memories usage report for the high-speed implementation.

```
1  ******************************************************
2  * MEMORIES USAGE REPORT
3  *
4  * Project   : googlenet_GPC_low_speed.json
5  * Version   : 1.0.0
6  * Date      : sab ott 01 12:40:23 2022
7  *
8  * Grid Width        : 10
9  * Grid Height       : 15
10 * Memories Size     : 8388608 bytes
11 ******************************************************
12
13 ******************************************************
14 * Summary
15 ******************************************************
16 Total Cores Memories Usage      : 122867776 bytes
17 Total Channels Memories Usage   : 12528 bytes
18 Total Memories Usage            : 122880304 bytes
19 Most Used Memory by Cores       : Mem10 (Core Memory Usage: 6422528 bytes)
20 Most Used Memory by Channels    : Mem31 (Channels Memory Usage: 144 bytes)
21
22 ******************************************************
23 * Memories Usage
24 ******************************************************
25 Memory ID:       Core Usage:      Channels Usage:  Total:
26 Mem10            6422528          72               6422600 bytes
27 Mem30            4816896          72               4816968 bytes
28 Mem50            4816896          72               4816968 bytes
29 Mem1209          4108096          72               4108168 bytes
30 Mem20            4014080          72               4014152 bytes
```

Listing 6.3. Extract of the GoogLeNet CNN on GPC (version 2) memories usage report for the low-speed implementation.

6.2.2 Timing

MapGL timing analysis evaluates channels' delays using three parameters: the memory read and write delays and the multiplexer propagation delay. The designer can set their values before starting the simulation.

The memory read and write delays were referred from [18]. In this case, we assumed that

the OFF-chip memories are DRAM, while the ON-chip memories are SRAM.
The OFF-chip memory read and write delay values were set inside the testbench manually.

The propagation delay of the multiplexer was arbitrarily chosen to be one-tenth of the ON-chip memory read/write delay.

The chosen communication delays are shown in Table 6.1.

Delay Type	Delay [ns]
OFF-chip memory read (DRAM)	50
OFF-chip memory write (DRAM)	50
ON-chip memory read (SRAM)	2.5
ON-chip memory write (SRAM)	2.5
Multiplexer propagation	0.25

Table 6.1. Communication delays used for the timing profiling [11].

Figure 6.2 shows the channel delays' heatmaps generated after the timing analysis of the high-speed and low-speed implementations.

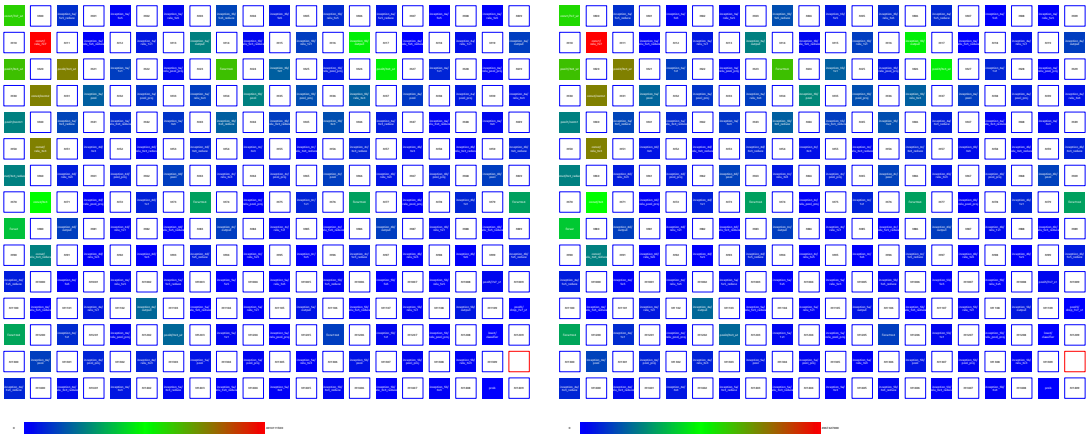

Figure 6.2. Timing analysis channel delays' heatmaps of the high-speed (on the left) and low-speed (on the right) implementations of the GoogLeNet on GPC (version 2).

Looking at Figure 6.2 seems that the two heatmaps look identical even if the channels should be faster in one implementation and slower in the other.
The reason is that the delays are normalized to the highest value. On the contrary, in the memories usage heatmaps in Figure 6.1, we notice a color difference because values are normalized to the memory size. The memory dimensions depend on the GPC grid size, so it remains the same in the two implementations.

We can also notice a correlation between the timing heatmaps in Figure 6.2 and the

memories usage heatmaps in Figure 6.1. The similarity in behavior is present because, by increasing the quantity of data to send, we increase the channel delay.

Listing 6.4 and Listing 6.5 shows the extracts of the timing reports of the two implementations.

```
1  ********************************************************
2  * TIMING REPORT
3  *
4  * Project    : googlenet_GPC_high_speed.json
5  * Version    : 1.0.0
6  * Date       : sab ott 01 12:43:59 2022
7  *
8  * Grid Width            : 10
9  * Grid Height           : 15
10 * Memory Write Delay    : 2.50e-09 s
11 * Memory Read Delay     : 2.50e-09 s
12 * Mux Delay             : 2.50e-10 s
13 ********************************************************
14
15 ********************************************************
16 * Cores Delays
17 ********************************************************
18 Core ID:  Module:                              Latency:   Exec:      Idle:      Channels:  Real Exec:
19 Core81    inception_4d/output (concat)         7.463e+07  1.474e+06  9.565e+05  5.175e+05  0.000e+00 ns
20 Core71    inception_4d/relu_pool_proj (relu)   7.575e+07  6.274e+04  0.000e+00  6.274e+04  0.000e+00 ns
21 Core50    conv2/relu_3x3 (relu)                3.960e+07  3.011e+06  0.000e+00  3.011e+06  0.000e+00 ns
22 Core70    conv2/3x3 (convolution)              3.760e+07  2.007e+06  2.000e+00  2.007e+06  0.000e+00 ns
23 Core1408  inception_5b/3x3 (convolution)       9.094e+07  7.058e+04  0.000e+00  7.058e+04  0.000e+00 ns
24 Core1407  inception_5b/relu_3x3_reduce (relu)  9.090e+07  4.706e+04  0.000e+00  4.706e+04  0.000e+00 ns
25 Core84    inception_4c/output (concat)         7.227e+07  1.443e+06  9.408e+05  5.018e+05  0.000e+00 ns
```

Listing 6.4. Extract of the GoogLeNet CNN on GPC (version 2) timing report for the high-speed implementation.

```
1  ********************************************************
2  * TIMING REPORT
3  *
4  * Project    : googlenet_GPC_low_speed.json
5  * Version    : 1.0.0
6  * Date       : sab ott 01 12:33:49 2022
7  *
8  * Grid Width            : 10
9  * Grid Height           : 15
10 * Memory Write Delay    : 2.50e-09 s
11 * Memory Read Delay     : 2.50e-09 s
12 * Mux Delay             : 2.50e-10 s
13 ********************************************************
14
15 ********************************************************
16 * Cores Delays
17 ********************************************************
18 Core ID:  Module:                                Latency:   Exec:      Idle:      Channels:  Real Exec:
19 Core86    inception_4c/relu_5x5_reduce (relu)    8.798e+07  5.694e+04  2.784e+04  2.911e+04  0.000e+00 ns
20 Core96    inception_4c/5x5_reduce (convolution)  8.733e+07  6.744e+05  3.118e+05  3.627e+05  0.000e+00 ns
21 Core19    inception_4a/output (concat)           6.808e+07  1.416e+07  1.354e+07  6.209e+05  0.000e+00 ns
22 Core09    inception_4a/relu_5x5 (relu)           6.906e+07  1.710e+05  1.128e+05  5.822e+04  0.000e+00 ns
23 Core84    inception_4c/output (concat)           8.688e+07  3.117e+06  2.496e+06  6.209e+05  0.000e+00 ns
24 Core85    inception_4c/relu_1x1 (relu)           8.673e+07  3.041e+05  1.489e+05  1.552e+05  0.000e+00 ns
25 Core28    inception_4a/1x1 (convolution)         6.728e+07  7.984e+05  3.909e+05  4.075e+05  0.000e+00 ns
```

Listing 6.5. Extract of the GoogLeNet CNN on GPC (version 2) timing report for the low-speed implementation.

The *latency* is the time from the start of the simulation to the first value popped. The

execution delay is the time from the first pop to the last push. The *idle delay* is the span in which the core waits for the other cores to push or pop the data. The *channel delay* is the time spent for communication purposes. Finally, the *real execution delay* is the time spent elaborating the data, which in this case is zero since we did not specify any delay inside our modules.

The values are sorted using the real execution delay, but in this case, they are basically randomly ordered.

The overall application delay of the high-speed implementation was 81708403 ns, while for the low-speed was 104018256 ns. The delay is reduced by 21% in the high-speed implementation but memories usage increases by 42%.

Adding delays to the layers

The last timing profiling was performed by adding the execution delays calculated as described in Section 5.3 to each layer. These values are only used to get a rough estimation of the overall application behavior. They are not to be intended as accurate performance results.

Figure 6.3 shows the real execution delays' heatmaps generated after analyzing the high-speed and low-speed implementations.

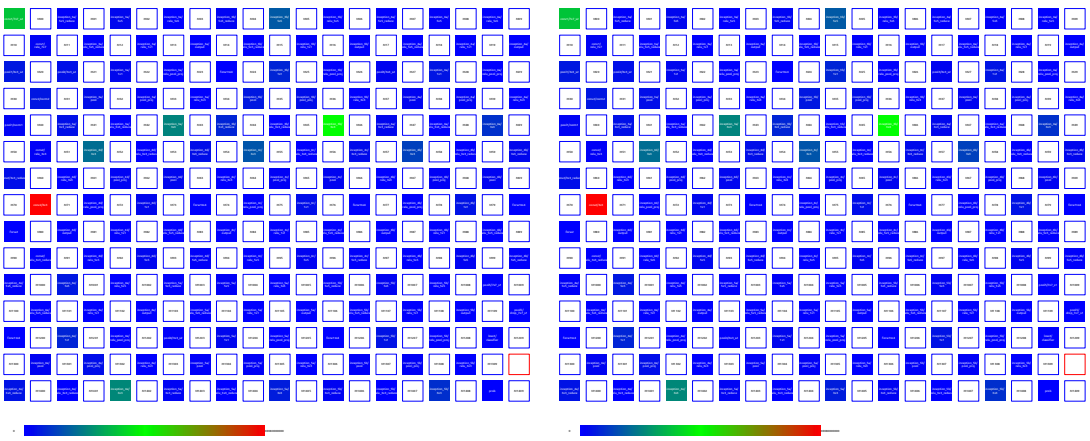

Figure 6.3. Timing analysis real execution delays' heatmaps of the high-speed (on the left) and low-speed (on the right) implementations of the GoogLeNet on GPC (version 2).

In this case, the two heatmaps are the same by definition. We notice that the *conv2/3x3* module has a higher execution delay than the other modules.
In future models, we could increase the size of the FIFOs for that particular core to

mitigate the bottleneck. Another solution could be splitting that module's operations into two cores.

This time, the reports shown in Listing 6.4 and Listing 6.5 are sorted correctly, thanks to the non-zero real execution delays. This way, we can have a rough idea of which cores are the slowest inside the structure for future improvements.

```
1  ******************************************************
2  * TIMING REPORT
3  *
4  * Project    : googlenet_GPC_high_speed.json
5  * Version    : 1.0.0
6  * Date       : sab ott 01 12:55:21 2022
7  *
8  * Grid Width            : 10
9  * Grid Height           : 15
10 * Memory Write Delay    : 2.50e-09 s
11 * Memory Read Delay     : 2.50e-09 s
12 * Mux Delay             : 2.50e-10 s
13 ******************************************************
14
15 ******************************************************
16 * Cores Delays
17 ******************************************************
18 Core ID:  Module:                          Latency:   Exec:      Idle:      Channels:  Real Exec:
19 Core70    conv2/3x3 (convolution)         7.486e+07  7.109e+07  2.000e+00  2.007e+06  6.908e+07 ns
20 Core46    inception_3b/3x3 (convolution)  1.951e+08  3.529e+07  0.000e+00  6.272e+05  3.466e+07 ns
21 Core00    conv1/7x7_s2 (convolution)      1.753e+07  3.602e+07  7.150e+06  2.383e+06  2.648e+07 ns
22 Core1402  inception_4e/3x3 (convolution)  3.230e+08  1.842e+07  0.000e+00  2.352e+05  1.818e+07 ns
23 Core43    inception_3a/3x3 (convolution)  1.667e+08  1.810e+07  0.000e+00  4.391e+05  1.766e+07 ns
24 Core51    inception_4d/3x3 (convolution)  2.918e+08  1.499e+07  0.000e+00  2.117e+05  1.478e+07 ns
25 Core05    inception_3b/5x5 (convolution)  1.908e+08  1.245e+07  0.000e+00  2.509e+05  1.220e+07 ns
```

Listing 6.6. Extract of the GoogLeNet CNN on GPC (version 2) timing report with layers delays for the high-speed implementation.

```
1  ******************************************************
2  * TIMING REPORT
3  *
4  * Project    : googlenet_GPC_low_speed.json
5  * Version    : 1.0.0
6  * Date       : sab ott 01 12:58:38 2022
7  *
8  * Grid Width            : 10
9  * Grid Height           : 15
10 * Memory Write Delay    : 2.50e-09 s
11 * Memory Read Delay     : 2.50e-09 s
12 * Mux Delay             : 2.50e-10 s
13 ******************************************************
14
15 ******************************************************
16 * Cores Delays
17 ******************************************************
18 Core ID:  Module:                          Latency:   Exec:      Idle:      Channels:  Real Exec:
19 Core70    conv2/3x3 (convolution)         7.691e+07  7.401e+07  2.446e+06  2.484e+06  6.908e+07 ns
20 Core46    inception_3b/3x3 (convolution)  2.023e+08  3.618e+07  7.447e+05  7.762e+05  3.466e+07 ns
21 Core00    conv1/7x7_s2 (convolution)      1.753e+07  3.887e+07  9.534e+06  2.860e+06  2.648e+07 ns
22 Core1402  inception_4e/3x3 (convolution)  3.396e+08  1.875e+07  2.792e+05  2.911e+05  1.818e+07 ns
23 Core43    inception_3a/3x3 (convolution)  1.721e+08  1.872e+07  5.213e+05  5.433e+05  1.766e+07 ns
24 Core51    inception_4d/3x3 (convolution)  3.055e+08  1.529e+07  2.513e+05  2.620e+05  1.478e+07 ns
25 Core05    inception_3b/5x5 (convolution)  1.988e+08  1.281e+07  2.978e+05  3.105e+05  1.220e+07 ns
```

Listing 6.7. Extract of the GoogLeNet CNN on GPC (version 2) timing report with layers delays for the low-speed implementation.

In this case, the overall application delay of the high-speed implementation was 351072036 ns, while for the low-speed was 369080897 ns. These results are much higher than the previous case in which the layer execution delay was equal to zero. The channels' contributions to the application delay are minimal using these delay values. Modifying the layers' execution delays can drastically change the results.

The aim of this last profiling was just to show how to use the timing profiling of MapGL at its highest potential, not to get any accurate results.

6.3 Comparison

In this section, we will compare the first version of the model with the high-speed and low-speed implementations of the second one.
For an even comparison, we will consider only the profiling results in which the layers' execution delays are equal to zero.

Table 6.2 shows the results of the three models, while Table 6.3 shows how the high-speed and low-speed implementations perform compared to the first version of the model.

	Grid size	**Total memories usage [MB]**	**Total delay [ms]**
1st version	15×13	210	159
2st version (high-speed)	15×10	175	82
2st version (low-speed)	15×10	123	104

Table 6.2. Results of the first and second versions of the GoogLeNet on GPC model. The second version has high-speed and low-speed implementations.

	Grid size	**Total memories usage**	**Total delay**
high-speed	−23.1%	−16.8%	−48.5%
low-speed	−23.1%	−41.6%	−34.5%

Table 6.3. Comparison between the first model and the high-speed and low-speed implementations.

The results in Table 6.2 are not to be considered accurate, they are just rough estimations.

More interesting are the percentages in Table 6.3, which give an idea of the improvements from the first to the second model.

One of the main reasons for the superiority of the second version over the first is the reduction of almost one-fourth of the grid size. By reducing the number of cores used, the application becomes faster and uses fewer memories.

Another critical factor is the number of accesses to external memories. Both models use the two side memories to shorten the path between inception blocks, which ideally is a good solution. However, Table 6.1 shows that DRAMs are much slower than SRAMs. The first version of the model pushes the same data through the side memories four times, while the second version pushes it just one time, thanks to the use of the *forw1to4* module. This lack drastically increases the first version application delay. However, this effect could be mitigated by reducing access delays to external memories.

Chapter 7

Conclusion

In this thesis, we described the design process and the profiling of two versions of the GoogLeNet on GPC model. We also introduced the MARI library and the MapGL editor that aids the analysis and mapping of new GPC applications.

The GoogLeNet model allowed us to exploit the scalability of the GPC architecture by designing and making simulations on a grid of up to 195 cores.

Using the MapGL editor, we created, visualized and modified our mapping without using any specific programming language. It also allowed us to save our project and automatically generate the SystemC model. Ultimately, we profiled our application using MapGL integrated tool for the memories usage and timing analysis.

7.1 Future works

Even using MapGL, mapping large applications on the GPC still requires lots of effort. Sometimes, a minor detail can lead the designer to redesign the entire structure.
A possible solution could be the implementation of an *auto-mapper*, which exploits all the possible mappings and chooses the best according to some constraints imposed by the designer. For each mapping, the auto-mapper could check the results of the application profiling, searching for critical paths or memory overflows. All the information extracted from the profile reports could be used to find the best mapping possible.
For CNN applications, the Caffe model file format that describes the network structure represents the perfect input. The auto-mapper can also decide whether a core should use one or more memories to store data or whether to use external or local memories.

At the moment, MapGL supports only SystemC export. However, it could be possible

in the future to export projects using other languages, such as VHDL or SystemVerilog. The editor works at a high level of abstraction and is modular, so it should not be too complex to implement. The limit could still come from the language itself.

It could also be possible to include inside MapGL the possibility to map application using other many-processor architectures. For example, one of them could be a 3D version of the GPC introduced in [5].

Bibliography

[1] L. Azriel, A. Mendelson, and U. Weiser, "Peripheral memory: a technique for fighting memory bandwidth bottleneck," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 54–57, 2015.

[2] S. A. McKee, "Reflections on the Memory Wall," in *Proceedings of the 1st Conference on Computing Frontiers*, CF '04, (New York, NY, USA), p. 162, Association for Computing Machinery, 2004.

[3] A. Gerstlauer, C. Haubelt, A. D. Pimentel, T. P. Stefanov, D. D. Gajski, and J. Teich, "Electronic system-level synthesis methodologies," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1517–1530, 2009.

[4] D. C. Black, J. Donovan, B. Bunton, and A. Keist, *SystemC: From the Ground Up*. Springer US, 2010.

[5] R. Dömer, "A Grid of Processing Cells (GPC) with Local Memories," tech. rep., CECS, University of California, Irvine, 2022.

[6] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, 2015.

[7] W. Müller, W. Rosenstiel, and J. Ruf, eds., *SystemC: Methodologies and Applications*. USA: Kluwer Academic Publishers, 2003.

[8] "IEEE Standard for Standard SystemC Language Reference Manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–638, 2012.

[9] J. Aynsley, "OSCI TLM-2.0 Language Reference Manual," *Open SystemC Initiative (OSCI)*, 2009.

[10] A. Daroui, "A Loosely-Timed TLM-2.0 Model of a JPEG Encoder on a Checkerboard GPC," *UC Irvine*, 2022.

[11] V. B. Govindasamy and R. Dömer, "Mapping of an APNG Encoder to the Grid of Processing Cells Architecture," *UC Irvine*, 2022.

[12] Y. Wang and R. Dömer, "A Scalable SystemC Model of a Checkerboard Grid of Processing Cells," *UC Irvine*, 2022.

[13] G. Bortolan, I. Christov, and I. Simova, "Potential of rule-based methods and deep learning architectures for ecg diagnostics," *Diagnostics*, vol. 11, no. 9, 2021.

[14] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[15] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov, "Realtime computer vision with opencv: Mobile computer-vision technology will soon become as ubiquitous as touch interfaces.," *Queue*, vol. 10, p. 40–56, apr 2012.

[16] E. M. Arasteh and R. Dömer, "Systematic Evaluation of Six Models of GoogLeNet using PDES," *UC Irvine*, 2021.

[17] J. M. Willman, *Beginning PyQt*. Apress, 2020.

[18] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware Software Interface ARM Edition*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2016.