POLITECNICO DI TORINO

Master degree course in Electronic Engineering

Master Degree Thesis

HLS techniques for high performance parallel codes in Logic-in-Memory systems

Supervisors
Prof.ssa Mariagrazia Graziano
Prof. Maurizio Zamboni
Prof.ssa Giovanna Turvani

Candidate
Alessio Naclerio
ID: 270065

Academic year 2021 – 2022

Summary

Recently, several researches have been conducted at the *VLSI Laboratory* of Politecnico di Torino on the *Logic-in-Memory (LiM)* model. It is an innovative architectural paradigm that aims at tackling the set of limitations showed by the *von-Neumann model*, usually referred to as *memory wall* or *von-Neumann bottleneck*. The key idea behind the concept of LiM is the implementation of memory devices in which *standard cells are equipped with simple computational units*. As a result, memory accesses are largely reduced, as well as power consumption and data-fetching latency. Moreover, high timing performance can be achieved, as the regular structure of LiM systems offers the possibility to perform *parallel processing*.

Two tools have been developed at the *VLSI Laboratory* with the purpose of creating a framework that allows a designer to easily devise and characterize LiM architectures. Firstly, *DEXiMA* was born as a simulator for LiM systems, providing the user with information about *timing performance*, *space occupation* and *static and dynamic power consumption*. Then, *Octantis* has been proposed, presenting itself as a *High-Level Synthesizer* that handles a C program and generates a LiM architecture for its execution. In order to accomplish this translation process, it considers the *LLVM Compiler Framework*, whose front-end (*Clang*) and *Optimization Passes* have been exploited. The Octantis back-end receives the *LLVM IR* code generated by the previous steps and produces the description of the LiM architecture by means of configuration files for DEXiMA. For this purpose, the typical four-stage structure of HLS tools have been implemented, encompassing *allocation*, *scheduling*, *binding* and *code emission*.

Although being able to synthesize parts of several LiM architectures already proposed at the *VLSI Laboratory*, Octantis only provided solutions for the handling of simple code constructs. The main objective of this thesis has been **the enlargement of the set of *data structures* and *C constructs***

available for the description of the input program while considering the benefits that they could derive from the parallelization opportunities offered by a LiM implementation. Hence, the work mainly focused on the identification of strategies for the management of **for-loop nests** and the synthesis of operations involving **linear and two-dimensional arrays**, whose elements can be visited by means of well-defined *Array Access Patterns*. The handling of these complex structures along with the implementation of loop unrolling allows Octantis to synthesize *highly parallel structures* capable of executing algorithms that require non-trivial access the elements of arrays.

A new Pass called **InfoCollector** has been developed in order to deal with the increased complexity of information characterizing the input algorithm. It performs an accurate analysis of the LLVM IR code with two main purposes: *speeding up the scheduling phase* and, most importantly, *gather information regarding nested loops and array accesses* inside the LLVM IR code. As regards the former, InfoCollector implements techniques that allow the overall execution time of the mentioned stage not to be affected by the larger amount of instructions to be taken into account. As a matter of fact, more elaborated algorithms can result in a huge number of LLVM IR operations to be considered by the scheduler. The latter task is crucial for the correct mapping and parallelization of operations involving vectors or matrices onto the final LiM architecture. Several concepts belonging to the mathematical framework referred to as *Polyhedral Model* have been employed to let InfoCollector provide the binder with data structures that allow the generation of an optimal LiM system, in which its intrinsic parallel capabilities are fully exploited.

As a consequence, *the scheduling and binding phases have undergone some modifications* in order to benefit from the introduction of the new pass. Specifically for the binding phase, an important *target-dependent optimization strategy* has been developed aimed at limiting the amount of needed hardware resources and the overall area as much as possible, while ensuring high timing performance. The implementation of parallel processing indeed offers the advantage of drastically reducing the total execution time but, at the same time, it causes an increase of space occupation.

Moreover, the code emission stage has also been expanded with the insertion of two new modules. The former aims at the production of configuration files for the new **DEXiMA-CAD**, which is a tool for DEXiMA that enables the

visual representation of the LiM architecture. The latter deals with the generation of a *VHDL* description of the system devised by Octantis synthesis process, as well as an associated VHDL *testbench* in order to check its correct behaviour by carrying out simulations with commercial EDA tools, such as *Modelsim*.

Finally, several *Image Processing* algorithms have been chosen for the synthesis on LiM devices. Since they are usually data-intensive, they are accelerated through parallel computing systems, such as GPUs. Hence, the LiM model may represent an alternative way to address their efficient implementation. Results have shown that Octantis is now able to synthesize *more complex structures with the possibility of parallelizing the execution of elaborated operations*, which can also require sophisticated access patterns to be considered for the visit of both C vectors and matrices. Furthermore, the introduced optimization strategy has been proved to enable a great saving of area and hardware resources.

In conclusion, the handling of loop nests and array accesses represents the starting point for the *synthesis of acknowledged HLS benchmarks* on LiM architectures, thus enabling the performance comparison with other already available implementations. Moreover, it is important to envision a more complex system where a LiM device and a conventional processing unit can coexist, in order to allow the implementation of even more complex operations. As a consequence, Octantis would have to devise several strategies to allocate the implementation of each instruction to one of the two units.

Contents

List of Tables 8

List of Figures 9

Introduction 11

I Octantis, a tool for Logic-in-Memory exploration **13**

1 Motivation and background 15
1.1 An introduction to the Logic-in-Memory model 15
1.2 DExIMA: a simulation tool for LiM systems 17

2 The Octantis project 19
2.1 Introduction 19
2.2 The LLVM Project 20
2.2.1 The LLVM Intermediate Representation 21
2.3 The structure of Octantis 24
2.3.1 From the input C algorithm to the optimized LLVM IR 24
2.3.2 The Back-End 26

II The expansion of Octantis **31**

Introduction and Motivations 33

3 Polyhedral Model: a powerful mathematical framework 37
3.1 Introduction to the Polyhedral Model 37
3.1.1 Definitions and concepts 38
3.1.2 The Loop Array Dependence graph 41

4 InfoCollector: a preliminary analysis pass 45
4.1 Introduction . . . 45
4.2 The collection of information . . . 46
4.2.1 The importance of alias analysis . . . 46
4.2.2 The handling of loops . . . 50
4.2.3 The handling of pointers . . . 56
4.3 The construction of Access Pattern Matrices . . . 63
4.4 The identification of *valid* Basic Blocks . . . 65

5 The evolution of Octantis structure 67
5.1 Introduction . . . 67
5.2 The scheduling phase: leveraging InfoCollector . . . 68
5.3 The binding phase: facing higher complexities . . . 70
5.3.1 Handling array access patterns . . . 72
5.3.2 A new target-dependent optimization . . . 83

6 The expansion of the code emission phase 89
6.1 The generation of VHDL files . . . 90
6.2 The generation of DExIMA-CAD configuration files . . . 92

7 Tests 95
7.1 Image Processing algorithms . . . 95
7.1.1 Synthesis of the Integral Image algorithm . . . 96
7.1.2 Synthesis of a multi-image encryption algorithm . . . 99
7.1.3 Synthesis of an approximated Arithmetic Mean Filter . 103

8 Conclusions and future works 107

Bibliography 109

List of Tables

7.1 Results regarding the types of rows present in the LiM system that implements the generation of the *Integral Image*. Data are provided for each optimization level. 98
7.2 Results obtained from the synthesis of the algorithm for the generation of the Integral Image. Along with the optimization level, different data are provided. 99
7.3 Aggregated results of the synthesis by Octantis of the algorithm for the generation of both the XOR-Image and the XOR-Keys. 102
7.4 Results concerning the types of rows inside the LiM system that implements the application of the approximated Arithmetic Mean Filter. Different values are provided along with the optimization level. 105
7.5 Overall results obtained from the synthesis of the algorithm implements the application of the approximated Arithmetic Mean Filter. Along with the optimization level, different data are provided. 105

List of Figures

2.1 The retargetability principle is highlighted in figure. It allows a compiler to handle multiple source programming languages and target machines. 21
2.2 Analysis and Transform Passes in the LLVM Compiler structure. 23
2.3 Block diagram of the Octantis structure. 24
3.1 A SCoP with the possible representations of the related Iteration Domain. Iteration Domain *a* is the set of all iteration vectors, while *b* shows the linear inequalities that, in turn, form the 2-dimensional polyhedron. 39
3.2 A simple C code with 3 nested loops is provided along with the associated LAD graph and the APMs and APMCs of each array. 42
4.1 Structure of an LLVM natural loop 52
4.2 Structure of a loop nest composed of two loops after the loop-simplify Pass has been issued 53
4.3 A simplified example program composed of 3 nested loops is considered. *LoopInfoTable* and the related organization of data inside *loopInfoMap* and *nestedLoopMap* are detailed. . . . 56
4.4 Visual representation of how the parameters of a *getelementptr* instruction are used to obtain the final pointer in the first example provided. 59
4.5 Code snippet related to the second example examined. The use of loop iterator in *getelementptr* instructions is highlighted. 60
4.6 Internal structure of *PointerInfoTable* for a simplified code. . . 62
4.7 A simplified code with 4 nested loops where the access to arrays *A* and *B* is provided at the top. The organization of information by means of *PointerInfoTable* and *LoopInfoTable* allows the creation of a *LAD graph*, from which the APMs of the two arrays are extracted. 64
4.8 Scheme showing the filter-like behaviour of InfoCollector. . . . 65

5.1 Movement of information and data structures among InfoCollector, the scheduler and the binder, with the introduction of parameters regarding pointers in Instruction Table. 69
5.2 Mapping of the accumulation operation on an set of LiM rows implemented following a *reduction-tree* strategy. 71
5.3 Identification of a set inside an array. 76
5.4 New internal structure of Octantis binder. 81
5.5 Two overlapping accumulation sets inside an array. 86
5.6 Two overlapping accumulation sets. 87
6.1 Interface of the entity generated by means of PrintVHDLFiles. 90

Introduction

The work carried out within this thesis aimed at the *expansion of the Octantis project*, a High-Level Synthesis tool for the exploration of Logic-in-Memory systems developed at the *VLSI Laboratory* of Politecnico di Torino. Given the intrinsic parallel capabilities of LiM systems, the Octantis synthesis process has been designed to be well-suited for applications that could benefit from a parallelized implementation. Hence, in this thesis, the set of data structures and constructs available for the development of the input algorithm has been enlarged, with particular interest shown towards the ones that are mostly employed for the description of parallel codes. Therefore, as deeply discussed in the second part of this document, the major focus has been put into the handling of *nested loops* and *one/two-dimensional arrays*. The combination of the loop unrolling technique and the study of array access patterns, namely the order in which their elements are visited, allowed the effective parallelization of more complex algorithms, suitable for the mapping on a LiM architecture.

In the first part of the document, the motivations behind the development of Octantis are presented, and the description of the internal structure of the tool is discussed. An overview of the research works carried out at the *VLSI Laboratory* of Politecnico di Torino on LiM architecture is provided, along with a brief introduction to DEXiMA. After that, the various stages that contribute to the Octantis synthesis process are analyzed.

In the second part, the introduction of *InfoCollector Pass* is deepened along with the consequent modification of the other Octantis HLS phases. The dissertation starts with the presentation of the *Polyhedral Model*, a mathematical framework whose main concepts have been exploited for the organization of information and the analyses implemented by InfoCollector. After that, the main tasks carried out by this new Pass are addressed, highlighting the complex code structures that it allows handling and the strategies

implemented towards their synthesis. Successively, the *evolution of both the scheduling and the binding phases* is discussed. As regards the former, the benefits derived from several techniques performed by InfoCollector are presented. As for the latter, the discussion is focused on how the structure of the binder has been modified to drive the synthesis process according to the data structures produced by the new Pass. Moreover, the insertion of an additional *target-dependent optimization technique* within the binding phase is addressed too, showing the advantages that it can bring in terms of saving of area occupation and hardware resources.

While InfoCollector enables the enhancement of Octantis input capabilities, the expansion of the code emission stage allows for a wider set of output description formats of the final LiM system to be available. Two new modules respectively aimed at the generation of *VHDL* and *DEXiMA-CAD* files are presented.

Finally, tests have been carried out addressing data-intensive algorithms belonging to the *Image Processing* field. Their C implementation has been given to Octantis for the synthesis on LiM systems. The results gathered about area occupation, hardware resources and timing performance are reported and discussed. Hence, the correctness of the novelties introduced and the improvements with respect to the previous version of the tool have been checked, along with the considerations on possible future expansions.

Part I

Octantis, a tool for Logic-in-Memory exploration

Chapter 1

Motivation and background

1.1 An introduction to the Logic-in-Memory model

During recent years, several researches have been carried out at *VLSI Laboratory* of Politecnico di Torino concerning the *Logic-in-Memory* paradigm. It represents a promising architectural solution that aims at overcoming the drawbacks of the *von-Neumann model* which are being experienced in the last decades.

As a matter of fact, the architecture of digital processing systems has been conceived according to this model, which requires the separation between the device responsible for the storage of data, the *memory*, and the one that performs computations, the *CPU*. Within this scheme, the CPU has to access the memory in order to retrieve data useful for subsequent elaboration. However, while the former has become more and more powerful over time due to CMOS technology scaling, the latter has not undergone the same improvement. This has resulted in a substantial difference between the computational speed that characterizes modern CPUs and the time required to perform memory accesses. As a consequence, a larger amount of time and energy is wasted while fetching data rather than elaborating them. Moreover, especially for data-intensive applications, the huge need for memory access highly affects the total power consumption. The set of issues that stem from the separation between memory and CPU as well as their performance gap

is generally referred to as *memory wall* or *von-Neumann bottleneck*.

The presented drawbacks gave energy to the exploration of the Logic-in-Memory model, whose main objective is to integrate computational units directly inside the memory. This choice can result in a drastic reduction of data-fetching latency and the related power consumption, also enabling faster elaboration of data with lower energy waste. From a structural point of view, LiM architectures have been conceived as arrays of *memory cells that can be equipped with relatively simple operational units*. In this way, all data elaborations are enclosed inside the LiM array, without the need of transferring information outside of it. Moreover, the high degree of regularity that characterizes this kind of structure enables the possibility to perform *parallel computing*, which represents a key feature that many implementations could take benefit from.

The research works carried out at *VLSI Laboratory* led to the definition of *CLiMA* [1], a *Configurable Logic-in-Memory Architecture*. The key feature of CLiMA is the availability of arrays whose basic block is represented by the so-called *CLiM cells*. The choice of this name derives directly from their internal structure, which consists of a normal memory cell equipped with logic that is configurable in order to allow different operations to be carried out. CLiMA has been exploited for the implementation of a *Quantized Convolutional Neural Network*[1] and the *Bitmap Indexing Algorithm*[2], providing promising performance results. Furthermore, studies regarding the possible benefits that LiM architectural solutions may derive from beyond-CMOS technologies have been carried out. For instance, always in [1] the Perpendicular Nano Magnetic Logic (*pNML*) technology has been considered for the implementation of CLiMA.

In conclusion, many researches have addressed the adoption of the LiM model for specific architectures. However, the main objective would be the creation of a framework that allows the exploration and characterization of LiM implementations, thus having a more comprehensive view that takes into account multiple abstraction levels. This is the main reason behind the development of both DEXiMA, which is briefly presented in the next section, and Octantis, whose description is deepened in Chapter 2.

1.2 DExIMA: a simulation tool for LiM systems

DEXiMA is a C++ based tool developed at *VLSI Laboratory of Politecnico di Torino* that performs the characterization of a system composed of a LiM component and an out-of-memory *CMOS* circuit, which are connected through a bus. By means of the implementation of both static and dynamic analysis, it can provide important information regarding *timing performance*, *area occupation* and *static* and *dynamic power consumption*.

A purely structural description of the entire system must be devised and fed to DEXiMA through proper configuration files. The *front-end* of the tool is responsible for producing the correct data structures required for the actual simulation starting from these files. As regards the LiM unit, a rectangular array of LiM cells must be defined. Its description has to contain information about the types of LiM cells employed, the modules that carry out intra-row or intra-column operations and how they are connected. Different kinds of cells can be exploited, each one characterized by different integrated logic. The out-of-memory system has to be defined in a similar way, firstly specifying the hardware modules, chosen from a pre-defined set of available components, that are required for the implementation and, after that, their interconnections.

The DEXiMA *back-end* is the one in charge of performing the simulation of the circuit. In order to do that, suitable models have been considered for the characterization of both the traditional logic and the memory cells. As regards the first, the information provided by *TAMTAMS*[3] has been taken into account. It is a web-based framework developed at *Politecnico di Torino* that can be employed for the analysis of CMOS circuits starting from their characteristic parameters. On the other hand, the memory performances are estimated by *CACTI*[4], an open-source simulator of memories developed by *Hewlett-Packard Laboratories*. However, solutions dealing with beyond-CMOS technologies are being considered, as the mentioned model turned out to be not suitable for the proper description of LiM cells.

As already mentioned, the simulation implemented by DEXiMA addresses both static and dynamic analysis. The former allows obtaining important static parameters like space occupation, maximum delay and static power

consumption. However, the latter enables the definition of dynamic power consumption, which is much more relevant with respect to the static one. For this purpose, the description of the algorithm that the system performs is needed, and it has to be provided in input to DEXiMA following a specific syntax.

In conclusion, DEXiMA represents a promising tool for the characterization of LiM systems, in constant evolution along with the research around this new innovative model. After the development of this tool, the introduction of the Octantis project has been taken into account in order to broaden the scope of the entire framework that was being designed. As a matter of fact, it has been decided to express the LiM array devised by the Octantis synthesis process by means of DEXiMA configuration files, thus enabling its simulation and connecting the two software.

Chapter 2

The Octantis project

2.1 Introduction

During the previous chapter, the main elements that have characterized the research on LiM systems carried out at *VLSI Laboratory* of Politecnico di Torino have been discussed. Octantis emerged within this lively environment, with the purpose of further exploring the potential of the LiM paradigm. It was born in 2020 thanks to the thesis work carried out by *A. Marchesin*[5], and the promising results that have been obtained led to the development of a scientific paper[6].

Octantis is a *High-Level Synthesizer*, developed in C++, whose main objective is the generation of an optimal LiM architecture, starting from an input algorithm described using standard C language. As it will be better clarified throughout the following sections, Octantis highly exploits the *LLVM Compiler Framework* in order to accomplish the mentioned "translation". LLVM not only offers a fully-developed open-source compiler infrastructure but also a set of related projects as well as libraries allowing developers to perform code analysis, transformation and optimization.

Octantis presents itself as an *agile tool for the exploration of LiM architectures*, enabling designers easy access to this innovative model, without being concerned by the implementation details. As a matter of fact, Octantis guides users through the examination of LiM solutions while letting them focus on the development of the input algorithm. Furthermore, a *modular approach* has been adopted for the internal structure of the tool in order to help future developers in the expansion of the code.

2.2 The LLVM Project

LLVM was born as a research project at the *University of Illinois* in 2000, under the responsibility of *V. Adve* and *C. Lattner*, addressing the implementation of modern strategies for supporting static and dynamic compilation of multiple programming languages. Since then, LLVM has evolved into a collection of *modular and reusable compiler technologies*, involving several subprojects, some of which are also used in production by several open source and commercial tools, as well as being largely exploited in academic research.

As highlighted by the same C. Lattner in its chapter of *The Architecture of Open Source Applications*[7], the LLVM compiler follows the traditional three-stage compiler structure that consists of:

- a **Front-End**: it parses the input code and checks for its overall correctness by conducting lexical, syntax and semantic analysis. As regards the LLVM compiler, these operations are carried out by *Clang*[8], a front-end specifically designed for C/C++. The last step of this stage is responsible for the generation of the *Intermediate Representation (IR)* of the source code. It consists in a semantically-equivalent description of the input algorithm expressed by means of a low-level, machine-like language. LLVM has its own intermediate representation, which will be examined in the following section.
- a **Middle-End**: during this stage, the IR undergoes a set of *target-independent optimizations*, which are achieved through the analysis of the code and the subsequent transformation.
- a **Back-end**: the main objective of this final step is the translation of the IR into the desired output description format, with the possibility of employing machine-dependent code optimizations, too.

The discussed architecture presents a crucial advantage that can be referred to as **retargetability**. The main aim of *retargetable compilers* consists in the exploitation of the common IR, which can be considered as a midpoint where different back-ends share the same comprehension of the source program. By means of this representation, also visualized in Figure 2.1, it is possible to *share the set of target-independent optimizations among multiple back-ends*. Moreover, this design allows the easy handling of a new source or target language, as it is unnecessary to redevelop the entire compiler structure. For instance, porting the compiler to manage a new source language

only requires the implementation of a new dedicated front-end, while the IR optimizer and the back-end can remain the same.

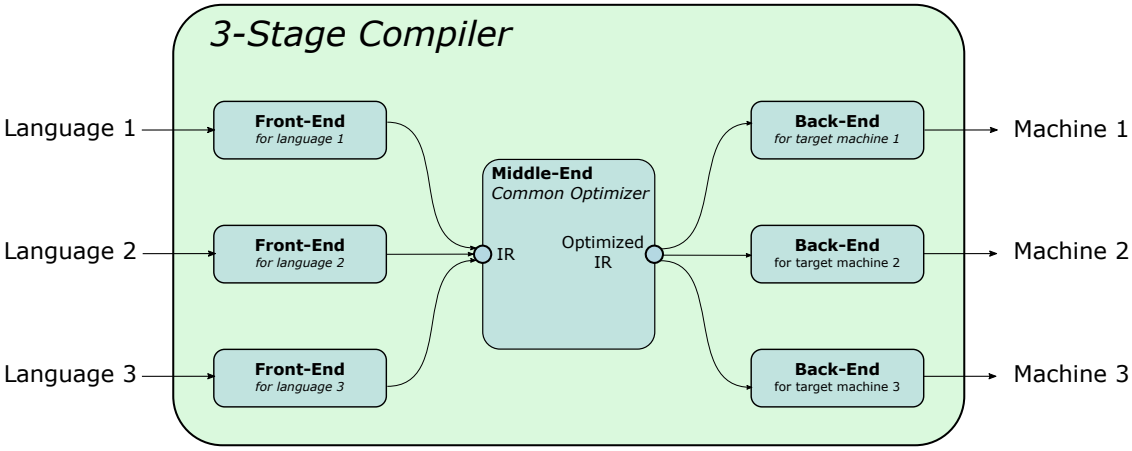

Figure 2.1. The retargetability principle is highlighted in figure. It allows a compiler to handle multiple source programming languages and target machines.

Another main advantage of this design, which also follows directly from retargetability, is that the compiler can serve a *larger set of programmers* than it would if it only provided solutions for the handling of one source and target language. For an open-source project, this results in the presence of a larger community of potential contributors to draw from, which naturally leads to more and faster improvements to the compiler.

Finally, one of the key aspects that favoured the choice of LLVM for the development of Octantis relies on its free licence, which allows deriving also commercial products from the original LLVM project.

2.2.1 The LLVM Intermediate Representation

As already highlighted in the previous section, the IR represents the *backbone* of the compiler structure, being the connection point between the front-end and the back-end. The LLVM project has its own intermediate representation, the *LLVM IR*, which consists of a *low-level language*, with a *RISC-like instruction set*. It has three equivalent forms:

- An in-memory representation.
- An on-disk representation encoded by means of the bitcode files.

- An on-disk representation provided in a human-readable form (the LLVM assembly files).

The internal structure of LLVM IR can be compared to the one that characterizes *Chinese Boxes*, a collection of several boxes one contained inside another. Similarly, the top-level entities of LLVM IR are represented by *modules*, which are composed of different *functions*. The latter, in turn, are formed by a sequence of *basic blocks*, whose fundamental units are single *instructions*. A module also contains *peripheral* entities, such as global variables, the target data layout, external function prototypes and data structure declarations.

The *control flow* of the program is determined by basic blocks, which are characterized by the presence of a single *entry point* and a single *exit point*. When a basic block is entered, all of its internal instructions are executed. Then, the terminator instruction is the one responsible for jumping to another block or returning from the function. Moreover, the first basic block of a function is a special one, as it must not be the target of any branch instruction.

Furthermore, the LLVM IR presents the following fundamental properties:

- It employs the **Static Single Assignment (SSA)** form. Each value only has a single instruction that defines it, and all of its successive uses can be immediately retraced to that specific instruction. This design choice allows the creation of *use-def chains* that highly simplifies the implementation of optimizations, avoiding the need of a separate data flow analysis to compute these chains.

- The structure of instructions follows the **three-address code** form. The operations that elaborate data act on two source operands and place the result in a distinct destination one.

- It has an **infinite number of registers**. LLVM local identifiers can assume any name that starts with the % symbol, including numbers that start from zero, such as %0 and %1, with no limitations on the maximum number.

As shown in Figure 2.2, the IR is the point where *target-independent optimizations* take place. As regards LLVM IR, they are implemented as

Passes[9], which parse portions of the program in order to either collect useful information or transform the code. Three main types of Passes are available in LLVM, each with different purposes:

1. *Analysis Passes* gather the information that other passes can effectively exploit to properly carry out their tasks. They efficiently recognize useful code proprieties and optimization opportunities.

2. *Transform Passes* remodel the code aiming at its optimization, eventually using the data structure generated by previously issued analysis passes.

3. *Utility Passes* provide useful utilities that do not otherwise fit categorization. For example, passes that write a module to bitcode are neither analysis nor transform passes.

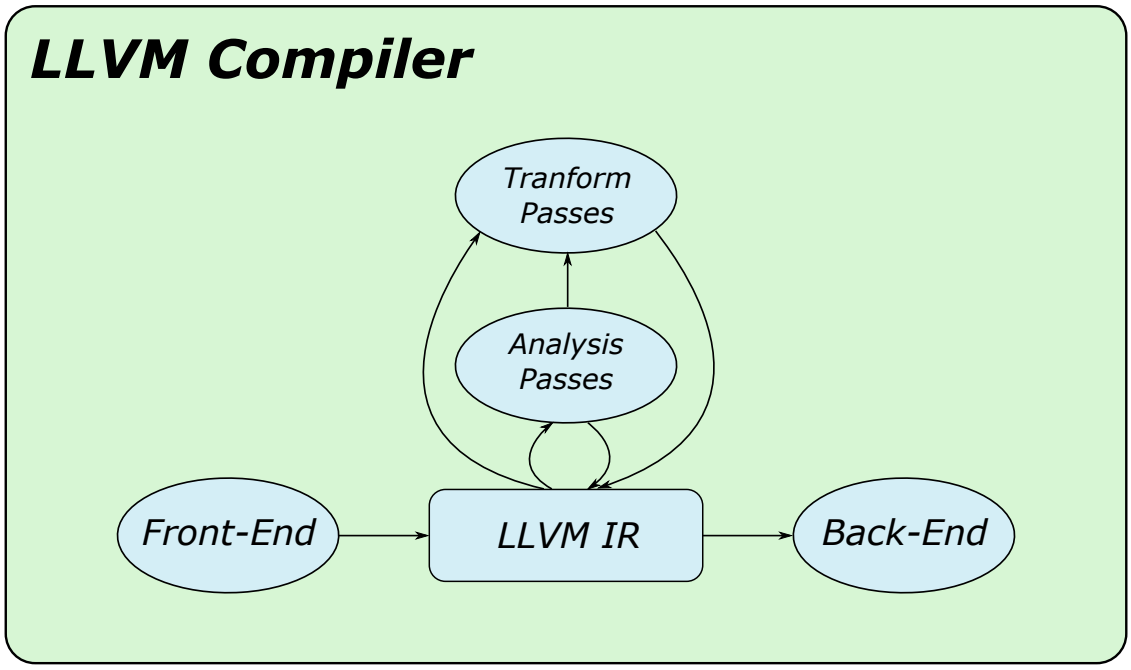

Figure 2.2. Analysis and Transform Passes in the LLVM Compiler structure.

In conclusion, further details about the LLVM structure and its IR can be found in [10].

2.3 The structure of Octantis

The Octantis project highly exploits the *LLVM Framework* to achieve the translation from C code to DExIMA configuration files, which are used for the description of the final LiM architecture.

First of all, in order to properly handle an input C algorithm, it takes advantage of the native LLVM front-end, *Clang*. Then, some already available *LLVM Passes* have been employed, as well as the one specifically designed for the implementation of loop unrolling, in order to perform the desired optimizations on the LLVM Intermediate Representation. From this step forward, the back-end of Octantis takes place, whose internal organization follows the typical four-stage structure common to all modern High-Level Synthesis tools, which consists of *allocation*, *scheduling*, *binding* and *code emission*. These phases are implemented by means of several C++ classes that are coordinated by a unique new Pass named *OctantisPass*. In Figure 2.3, a schematic overview of the blocks that contribute to the structure of Octantis is shown.

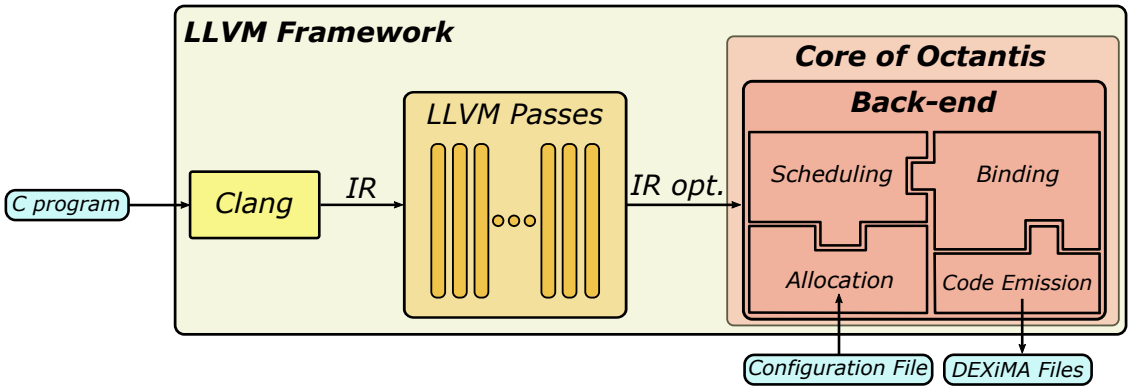

Figure 2.3. Block diagram of the Octantis structure.

2.3.1 From the input C algorithm to the optimized LLVM IR

As already mentioned, the first step of Octantis compilation requires the input C program to be handed over to Clang, which is in charge of translating it into LLVM Intermediate Representation. However, several *constraints*

must be considered while developing the algorithm. As a matter of fact, even though the C language offers the possibility to exploit high-level programming constructs, the users shall keep in mind that the design of a hardware component is being performed. Hence, the following things should be strictly avoided:

- **Dynamic allocation of memory**: a LiM system is basically a memory, hence this feature is in direct contrast with the very concept of LiM.
- **Recursive function calls**: they represent a highly complex C structure difficult to handle for whatever hardware implementation.
- **Multiplication and division**: they are known to consume a large number of hardware resources. As for the LiM system, the complexity is required to be rather limited in order to enable fast parallel computations with low power consumption. However, shift operations represent an alternative option, although only approximated results can be obtained.

Hence, all of these operations would be either meaningless or inconvenient to integrate into a LiM architecture. Furthermore, it is compulsory to *declare variables as integers*, due to the fact that the LiM system only handles arithmetic operations on this type of data. Finally, the users can also exploit *bit-wise logic operators*, even the negative ones (i.e. nand, xnor, nor), which are not included in standard C.

In addition to the input C program, Octantis also requires a *configuration file*. Its structure has been designed to gather all the possible constraints that can be exploited in order to explore the design space throughout the synthesis process. As regards Octantis first release, only the handling of the information that determines the dimension of a LiM row has been effectively implemented. However, the organization of the configuration file is such to manage other types of parameters during future expansions of the program.

Once Clang has generated the LLVM IR, several optimizations are run, mainly performed by already available *LLVM Passes*, such as *mem2reg* and *simplifycfg*. The former promotes memory references with register ones, strengthening the SSA form of the code. The latter performs dead code elimination and merging of basic blocks when possible, in order to produce a more efficient code. Along with these two, passes that address loop analysis and transformation have been taken into account. The introduced LLVM Passes dealing with loops are:

- **licm Pass**: it tries to reduce the size of the body of the loop by removing as many instructions as possible. This operation can help the next phases reduce the number of resources used.
- **loop-deletion Pass**: it prunes the input IR code in order to delete all the loops that do not participate in the computation of the final results.
- **loop-reduce Pass**: it reduces the number of array references inside the loops and, in particular, the ones regarding the management of the variable used as the index.
- **loop-simplify Pass**: it is responsible to transform loops in simpler forms whenever possible.

While all these passes mostly deal with the generation of a more compact and efficient code, a **loop unrolling** pass has been developed as it represents a relevant optimization technique to be exploited in relation to LiM architectures. As a matter of fact, the intrinsic parallel capabilities of this kind of system are well-suited for the concurrent execution of multiple loop iterations. This results in an increasing amount of hardware resources as well as area occupation, but it has the great advantage of largely reducing the overall execution time.

2.3.2 The Back-End

The back-end is the core of the Octantis project, as it is responsible for the effective generation of the final optimized LiM architecture starting from the IR code coming from the previous steps. As already mentioned, its internal organization follows the structure of all modern High-Level Synthesis tools, which is composed of allocation, scheduling, binding and code emission phases. During the next paragraphs, an overview of the tasks carried out by each of the four stages is presented, motivating the design choices that have been made.

Allocation

The allocation phase is the one responsible for the identification of the constraints specified by the designer inside the configuration file. The collection of this information is crucial in order to properly drive the whole synthesis process. As regards Octantis, the allocator actually handle only the parameter that defines the memory word size. As already discussed, the structure

of the configuration file has been devised with the purpose of being easily expandable, thus allowing the introduction of new constraints that could be useful in future versions of the tool.

Scheduling

The main aim of the scheduling phase is to assign each instruction that must be mapped on hardware with an execution time, thus creating a sort of *finite state machine* of the program. The scheduler of Octantis implements an *As Soon As Possible* (ASAP) algorithm, whose objective is to determine the lowest possible time instant in which a specific operation can be issued. In order to properly carry out this task, the presence of data dependencies must be checked, as they have to be respected for the correct execution of the algorithm devised by the designer.

As regards Octantis, the structure of the LLVM IR code must be carefully considered in order to keep track of the evolution of the input program. As a matter of fact, a well-defined and fixed approach is implemented in the LLVM IR, from the definition of variables to their elaboration. Once this "pattern" is known, it can be followed to identify all the operations performed in the algorithm. First of all, each variable has to be *allocated* in the stack region of the memory. When an arithmetic or logic operation between two operands has to be performed, a *load* must be executed to store them inside local registers. Then, the actual operation is issued, followed by a *store*, which is useful to save the obtained result again into the stack.

Since the main goal of Octantis is the generation of a memory, load and store instructions that take into account a stack region are meaningless for the mapping on such a device. However, they are highly exploited by the scheduler respectively in order to know which variable must be allocated inside the LiM array, and when a result is available. Hence, they are essential for the detection of *data dependencies*. Regarding the effective operations that can be carried out for the elaboration of two source operands, Octantis supports *sum* and *bit-wise logic operators*, such as and, xor, or, along with their negated form (nand, xnor, nor). Their detection is performed by the scheduler, too.

At the end of the described process, the instructions are stored in a dedicated data structure, called *Instruction Table*, which can only contain:

- **load** operations, specifying the operand to be allocated inside the LiM array.
- **arithmetic** and **logic** operations that are performed between two source operands, which have been previously allocated.

Binding

The binding phase is responsible for the correct mapping of instructions onto the desired output hardware structure, which, in the case of Octantis, is represented by a LiM architecture. The C++ class responsible for the mentioned task is called *LiMCompiler*, and it is invoked right after the scheduler by OctantisPass. It generates a LiM unit that is capable of performing the operations required by the input algorithm, and a related *finite state machine* based on the control information gathered during the scheduling phase.

Before starting the discussion about how the binder carries out its task, it is useful to have an overview of the LiM architecture taken as a reference by Octantis. The main characteristic of the memory array is its *regularity*. Each row has the same size and the cells belonging to a single row are uniform and equipped with the same internal logic. As regards interfaces, rows can handle *two input connections*, one in input to the memory cell and the other to the internal logic, and *one output connection*. In order to carry out an operation, two source operands are needed. They are stored inside two separate memory rows, one of which must include the needed computational unit useful for the required elaboration. Finally, an additional row is exploited to store the result. Moreover, also configurable cells can be defined, referring to the structure of the CLiM-Architectures briefly described before.

When LiMCompiler is launched, the parsing of the Instruction Table begins. As already pointed out in the previous section, two possible types of instructions can be found inside this data structure, load and arithmetic/-logic operations. When considering the former, the binder inserts a new memory row with the same size specified in the configuration file inside the array, without the need of equipping its cells with any logic. On the other hand, the latter requires more complex handling, described as follows. Once the two source operands memory rows are identified, three cases can occur:

- *both or one of the source rows are not equipped with any logic*: one of the two is chosen to be enhanced with the needed operators, connecting

their other input to the second operand.

- *one of the source rows has the same logic required for the current operation*: if the number of input connections previously present in the mentioned row was 1, an additional connection is added coming from the other source row. This choice has been made in order to reduce complexity.

- *both source rows have different logic operators from the one required by the current operation*: this obviously represents the worst case, having to duplicate one of the two source rows in a new one with the right LiM cells. The other row is set as the other input for the logic.

Special cases are the ones represented by operations inside loops and accumulations. As regards the former, the scheduler collects the needed information for the binder in order to properly map the unrolled loop onto the LiM array. Hence, the number of inserted source or result LiM rows is given by the number of loop iterations. On the other hand, the latter represents a peculiar case effectively detected by the scheduler, in which one of the two source operands is also the destination one. In this situation, the binder identifies the set of LiM rows that contain the data to be accumulated, and it inserts rows with intermediate results following a *reduction tree* strategy, in order to compute the final result.

Along with the LiM array described above, LiMCompiler is also responsible for the generation of a data structure handling the control flow of the mentioned architecture. Its main aim is to keep track of the active time of each LiM row. This type of information is fundamental in order to properly simulate the behaviour of the overall structure, but also to estimate its dynamic power consumption thanks to DExIMA.

Code Emission

The code emission phase, which is handled by *PrintDexFile* class, is in charge of generating the output configuration file for DExIMA, starting from the data structures that describe the final LiM system produced by the Octantis synthesis process. This file is crucial to provide DEXiMA with both the synthesized LiM architecture and the control flow of instructions that are meant to be executed, which is useful for the dynamic simulation.

The description of the array takes place as a traditional RTL circuit. First of all, the memory dimension has to be specified. Then, LiM cells composing the structure have to be declared specifying their internal logic. In parallel, the definition of *interconnections* has to be provided, which can be either *inter* or *intra cells*. The former refers to the connection between two different LiM rows, thus having a "vertical" flow of information. The latter indicates connections inside a single row. This case occurs when an addition operation is mapped, having to propagate the carry among the cells of the same row. This type of connection is not made explicit by Octantis, however, PrintDexFile recognizes it and properly defines it in the configuration file.

During Octantis first release, it has been decided not to print the section regarding control signals due to the fact that DExIMA simulation details were under study. However, Octantis could provide in output the full list of memory rows with their respective active time, in order to perform a quick debug on the generated structure.

Part II

The expansion of Octantis

Introduction and Motivations

During the previous dissertation, an overview of the Octantis project has been provided. As already highlighted, the expressiveness of the input C algorithm is highly limited, mainly due to the fact that certain C structures are not easy or even feasible to map on an hardware component. Furthermore, being Octantis in its early stages, it has been decided to focus on providing the internal structure of the program with a *modular* organization rather than handling complex code constructs, which has been left to future expansions of the tool. Hence, the main aim of this thesis work is to **broaden the set of C structures to be considered by the Octantis synthesis process**. The choice of the algorithmic constructs to be taken into account has been made examining the possible benefits that they could derive from parallel processing on LiM systems.

Switch and *for loops* represent the main basic C statements that the first version of Octantis could manage. While the former does not take much advantage from a LiM implementation, the latter is certainly more likely to profit from the intrinsic *parallel capabilities* of the Logic-in-Memory paradigm. As a matter of fact, a *loop unrolling* pass has been specifically designed in order to fully exploit the mentioned characteristic. However, Octantis synthesis could only handle:

- Simple for loops with *no nesting structures*.
- *Trivial access* only to *one-dimensional arrays* inside loops, namely the common row major order.

The innovations introduced in the scope of this work mainly aim at overcoming these limitations, giving Octantis the possibility to manage **loop**

nests and the synthesis of instructions involving **one/two-dimensional arrays**, whose elements can be visited following well-defined **Array Access Patterns**. At the same time, the extent of use of loop unrolling has been enlarged in order to cope with the new structures. As a matter of fact, if data dependencies are avoided, the LiM system can hugely benefit from the application of the mentioned technique, especially in terms of execution time. Moreover, this expansion of the tool represents a step forward in the analysis of LiM capabilities in relation to *data-intensive applications*, which are highly affected by the drawbacks of the *Von Neumann bottleneck*, as already pointed out in the introduction of the thesis.

As will be better clarified during the next chapters, a new Pass called **InfoCollector** has been introduced at the beginning of Octantis synthesis process, with the purpose of handling the *increased amount of information* coming from the input C algorithm. It organizes it exploiting dedicated data structures that are also useful for the subsequent phases. The introduction of this preliminary pass has indeed led to the **expansion of the previous structure of Octantis**. Furthermore, **target-dependent optimizations** have been explored in order to minimize the amount of needed hardware resources used for the final LiM architecture, whose area occupation may become huge when considering more complex data-intensive algorithms.

Finally, the *code emission phase has been enlarged* with the addition of two modules. The former addresses the generation of a **VHDL description** of the LiM structure, including both the declaration of the datapath and the control unit useful to properly drive the execution of the algorithm. A *VHDL testbench* is also provided in order to test the correct behaviour of the system. The latter is responsible for the production of the needed configuration files for the new **DExIMA-CAD** tool, which allows a visual representation of the LiM architecture.

The innovations carried out towards the expansion of Octantis are presented and discussed during the next chapters. First of all, an introduction to the *Polyhedral Model* is provided in Chapter 3, since InfoCollector Pass and the rest of the synthesis process exploit several data structures that are originated from the main concepts and definitions of this model. After that, the actual innovations are highlighted. The introduction of *InfoCollector* is presented in Chapter 4. The aims and strategies adopted by the pass regarding the handling of information are addressed, as well as the employed data

structures. In Chapter 5 the expansion of Octantis structure is addressed, highlighting the modifications to the HLS stages and the benefits they have derived from the introduction of the new pass are described. Successively, the expansion of the code emission phase is discussed in Chapter 6. Finally, Chapter 7 is dedicated to the *test cases* that have been devised in order to verify the behaviour of the implemented innovations and gather meaningful results.

Chapter 3

Polyhedral Model: a powerful mathematical framework

3.1 Introduction to the Polyhedral Model

InfoCollector takes advantage of some of the mathematical definitions that belong to the *polyhedral model* in order to effectively organize and successively use the information regarding *nested loops* and *array accesses*.

The *polyhedral model* is a *mathematical framework* whose theoretical foundations can be traced back to the work carried out by *Karp*, *Miller*, and *Winograd* in 1968 within their seminal contribution called *The Organization of Computations for Uniform Recurrence Equations*[11]. After that, several studies[12] were conducted addressing the possibility of *representing programs by means of a set of linear equations*, with the purpose of restructuring them for *parallel execution*. Moreover, a great contribution was given by the researches on *loop transformations*[13] and *systolic arrays design*[14] that were carried out between the 70s and the 80s. During the 90s, all the mentioned theoretical bases led to the actual development of the first tools that enabled the use of *polyhedral optimizations*, such as *PIP*[15] and *PolyLib*[16].

In the last decades, the polyhedral model has been considered in the field of **compilation tools** as it is capable of providing an innovative way of

representing programs that can be exploited for **optimized code generation**. Nowadays, the large adoption of parallel hardware accelerators, such as Graphic Processing Units (GPUs), requires compilers to implement proper code analyses and transformations to let the algorithms benefit from the mapping on such kinds of devices. The polyhedral model ensures a *fine-grained representation* of the program, which is achieved by means of the construction of mathematical relationships among several variables of interest present in the program itself. This is particularly useful in applications characterized by a large number of operations, such as algorithms with loop nests. This model allows addressing the entire set of loop iterations at once, thus devising restructuring opportunities that enable advanced optimization techniques aimed at the parallelization of the program.

The progress made in the exploration of the capabilities of the polyhedral model led to the birth of the so-called **polyhedral compilation**[17], which includes the set of compilation analysis and optimization techniques that rely on the representation provided by the model. It has been already employed in many compiler tools, such as *Polly*[18] for what concerns the LLVM framework. Its application has been also proposed to be employed in other contexts, including *memory usage optimizations*[19], *code generation for high-level synthesis*[20]. Moreover, the polyhedral community has a strong academic background and it is a lively research community, always exploring possible further applications of the model.

3.1.1 Definitions and concepts

The mathematical objects around which the theory of the polyhedral framework revolves are referred to as **Polyhedra**. They consist of *sets of points in a $\mathbb{Z}$ vector space whose borders are originated by a set of* ***linear inequalities***. The main aim of this model is to represent a program by means of polyhedra, which can be manipulated using mathematical transformations in order to optimize the code that originated them. The exploitation of this representation is well-suited for algorithms characterized by the presence of *loops*.

Each loop in a program is characterized by *boundaries* and, eventually, *conditionals*, which are identified by *if statements* that can be present in the loop itself. The operands that are involved in the definition of both loop boundaries and conditionals must only depend on *loop iterators* and *constants* in

order to produce the mentioned inequalities that generate a polyhedron. The set of consecutive statements in the code where loop bounds and conditionals satisfy these requirements is defined as **Static Control Part** (SCoP).

The discussed features can be also expressed by means of a proper mathematical formalism. To begin with, in programming languages like *C*, an n-deep loop nest can be represented through an n-entry vector called **Iteration Vector**:

$$\vec{x} = (i_1, i_2, ..., i_{n-1}, i_n) \tag{3.1}$$

Each element i_k indicates the iterator of the k^{th} loop, where k indicates the depth of the loop itself, n being the innermost one. Since iterators can assume multiple values depending on the characteristics of the related loop, many iteration vectors are originated, and they identify the **Iteration Domain**. It can also be represented by the set of linear inequalities that stem from loop bounds and conditionals mentioned above.

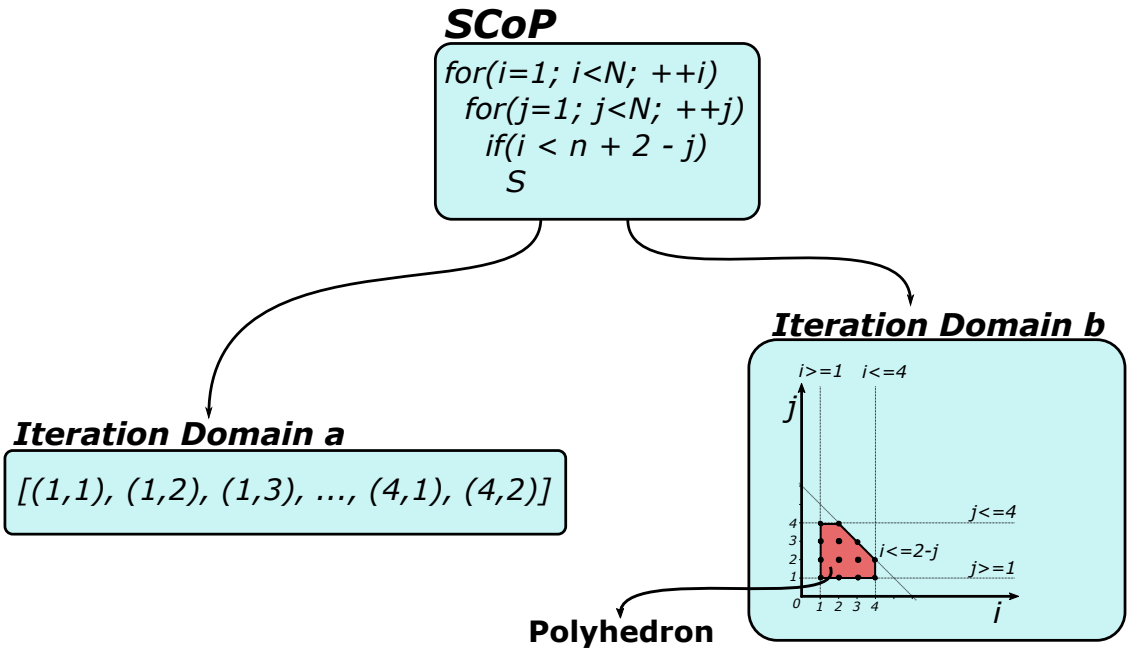

Figure 3.1. A SCoP with the possible representations of the related Iteration Domain. Iteration Domain *a* is the set of all iteration vectors, while *b* shows the linear inequalities that, in turn, form the 2-dimensional polyhedron.

Figure 3.1 shows an example SCoP and the related visualizations of the

iteration domain. As it can be noticed in the referenced Figure, a *two-dimensional polyhedron* has been generated by representing the iteration domain on the *ij*-plane. The number of dimensions that characterize this mathematical object is equal to the maximum depth of the loop nest.

At this point in code analysis, the generation of optimized code relies on the proper *scanning of polyhedra* that have been produced. However, this thesis work addresses the handling of nested loops towards the efficient parallelization and mapping of operations involving arrays onto a LiM architecture. Hence, other concepts that belong to the same model and deal with the management of array accesses have been considered, thus leaving the optimization of the algorithm to eventual improved future versions of Octantis.

In the polyhedral model, each array access is assigned with an **Array Access Function**. In order to allow its comprehension, the concept of **Access Vector** must be introduced. Given a m-dimensional array $A[a_1][a_2]...[a_{M-1}][a_M]$, it is defined as:

$$R_A = (a_1, a_2, ..., a_{M-1}, a_M) \tag{3.2}$$

Each of the possible values that R_A can assume enables access to a specific element of the array A, and the set of all these values forms the **Array Domain**. The array access function is defined as:

$$F : D_I \longrightarrow D_A, \tag{3.3}$$

where D_I is the iteration domain and D_A is the array domain, and its main purpose is to *provide information about the element of the array A that is visited during a specific iteration of the loop nest.* The function can be expressed by means of the following matrix equation:

$$F \begin{pmatrix} i_1 \\ i_2 \\ . \\ . \\ . \\ i_{n-1} \\ i_n \end{pmatrix} = APM_A \begin{pmatrix} i_1 \\ i_2 \\ . \\ . \\ . \\ i_{n-1} \\ i_n \end{pmatrix} + APMC_A \tag{3.4}$$

where APM is an MxN matrix that will be hereinafter referred to as **Access Pattern Matrix (APM)** that multiplies the Nx1 iteration vector, and the result is added to the so-called **Access Pattern Matrix Constant (APMC)**, which is Mx1. *APM* and *APMC* can identify a well-defined *Array Access Pattern* for the array A that indicates the order in which the elements of the array are visited in the loop nest.

3.1.2 The Loop Array Dependence graph

As already mentioned, one of the main tasks that the InfoCollector module must accomplish consists in the collection of information about nested loops and array accesses. This would allow obtaining the access pattern for each array instance in the algorithm, and consequently performing the mapping of the related operations onto a LiM system. Hence, InfoCollector needs to implement a proper solution to enable the *efficient organization of data regarding loops nests and arrays in order to create all the needed APMs and APMCs*, which will be exploited for the *correct mapping of operations during the binding phase*.

The concept of **Loop Array Dependence Graph (LAD)** has been developed in [21] the context of design space exploration for HLS tools. It is capable of representing the relationships among loops belonging to the same nest but also between the loops and arrays accesses. The formal definition of the LAD graph is the following:

- Given:
  1. a **loop nest** L composed of N loops: $L = L_1, L_2, ..., L_{N-1}, L_N$
  2. a set A of M **arrays** accessed inside the nest: $A = A_1, A_2, ..., A_{M-1}, A_M$
- The LAD is the the *directed graph* $G(V, E)$ where:
  1. $V = L \cup A$
  2. E is made up of the following edges:
     - (L_i, L_{i-1}), if elements of L appear in order from the outermost to the innermost, as $L = L_1, L_2, ..., L_{N-1}, L_N$.
     - (L_i, A_j), if array A_j is accessed using the iterator that belongs to loop L_i.

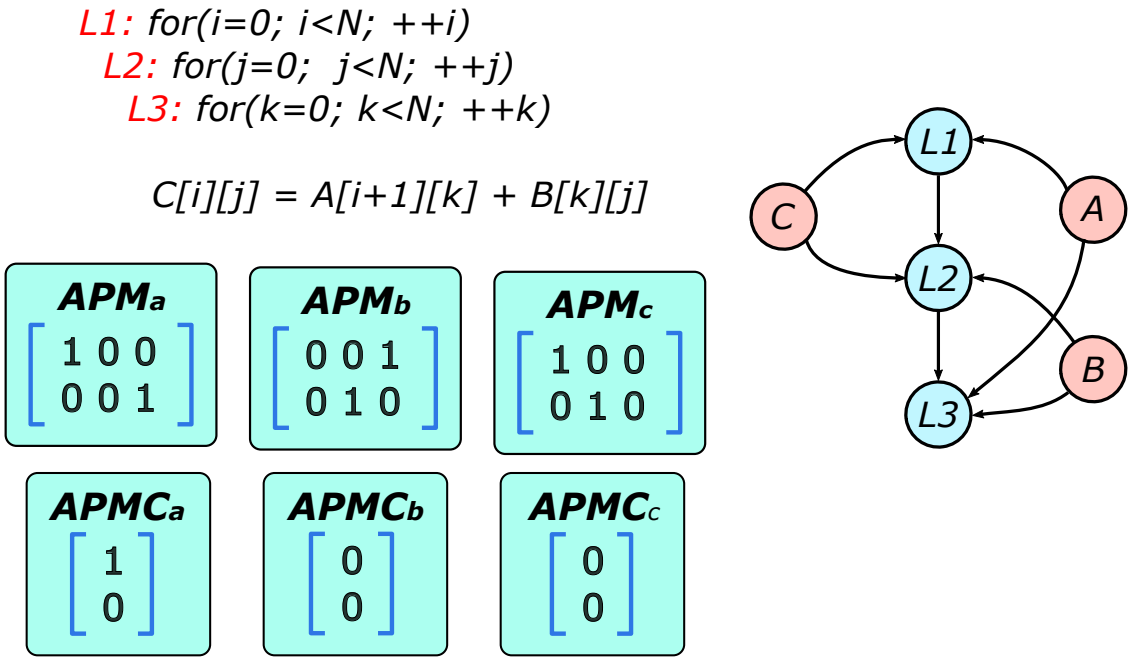

Figure 3.2. A simple C code with 3 nested loops is provided along with the associated LAD graph and the APMs and APMCs of each array.

A visual representation of the LAD graph is provided along with the C code that generates it in Figure 3.2. APMs can be easily retrieved by combining the information regarding the depth of each loop inside its nest and the iterators that contribute to the formation of the array indexes. As a matter of fact, the ordered organization of loops allows obtaining the iteration vector. The edges between an array and a loop node can be exploited in order to properly fill the related APM as follows:

- Given that array A is MxN and L loops are present:
  - If the edge (L_i, A_j) is present and the iterator of L_i contributes to index k of A, the element at position (k, i) of the APM will be the integer n that multiplies the iterator in its occurrence within the array index.
  - If the edge (L_i, A_j) is present but the iterator of L_i does not contribute to index k of A, the element at position (k, i) of the APM will be zero.
  - If the edge (L_i, A_j) is not present, the element at position (i) in each row of the APM will be zero.

On the other hand, the elements of APMCs can easily be retrieved by checking for the presence of constants in the index of the array instance under consideration.

Figure 3.2 shows an example C code with the related LAD graph and the APMs and APMCs belonging to each of the three arrays instances inside the loop nest. As will be better explained in the following chapters, InfoCollector implements a strategy that takes great inspiration from how the LAD graph organizes these types of information. The main aim is to generate the APMs and APMCs, which are further analyzed to retrieve the access pattern of each array in order to perform the correct parallelization of operations while mapping them onto the final LiM system.

Chapter 4

InfoCollector: a preliminary analysis pass

4.1 Introduction

Due to the necessity to handle more complex C constructs and data structures that can be employed for the description of the input algorithm, a new pass called **InfoCollector** has been developed. It has been introduced as it performs an appropriate *analysis of the LLVM IR code* to allow the upcoming stages, binding and scheduling, to efficiently *manage the access to 1/2-dimensional arrays inside nested loops*.

Several concepts presented in Chapter 3 have been exploited. As a matter of fact, gathering useful information for the creation of *Access Pattern Matrices* is one of the key tasks addressed by InfoCollector, as they represent the central objects around which the mapping and parallelization of operations on the LiM system revolve.

The set of data structures and the techniques implemented by InfoCollector are discussed in detail during the dissertation of the following sections, as they constitute a fundamental *preliminary step* that lays the foundation for the evolution of the scheduling and binding phases.

4.2 The collection of information

As it can be easily noticed by its name, InfoCollector mainly focuses on the *collection of information from the LLVM IR code*. As a consequence, the structure, the organization of the control flow and the meaning of each instruction of LLVM IR must be clear to carry out this task in an efficient way. For this reason, during the dissertation about the various analysis performed by InfoCollector, several aspects of LLVM IR will be deepened, thus allowing a better understanding of the strategies adopted. Three main operations that address the gathering of information are implemented by the new pass:

1. **Alias Analysis** aims at the identification of multiple local registers that are used as aliases of a variable that has been allocated at the beginning of the IR program. A proper data structure is exploited to keep track of the relationship between aliases. It is extremely useful not only for other analysis operations carried out by InfoCollector itself but also to let the scheduling phase properly accomplish its task by quickly detecting data dependencies between instructions.

2. **Loop Analysis** is aimed at recognizing loop iterators, operands that identify loop bounds and, most importantly, the order of loops inside a nest. It highly exploits the fixed form that characterizes LLVM *natural loops*, presented in 4.2.2, in order to detect the mentioned parameters. The information gathered is fundamental for the creation of a structure inspired by the LAD graph.

3. **Pointer Analysis** explores the body of a loop searching for pointers, which are used in the LLVM IR code to perform accesses to arrays. They are easily recognizable as they are the result of the *getelementptr* instruction. Once identified, they are analyzed in order to contribute to the completion of the LAD graph.

In the next sections, the techniques adopted to carry out these three kinds of analysis are detailed.

4.2.1 The importance of alias analysis

Here, the strategy that has been devised to execute *alias analysis* is considered. Before that, further details regarding the LLVM IR code structure

are presented, as they are important for the proper comprehension of the technique implemented.

According to the LLVM IR conventions, any variable belonging to the input program has to be reserved its relative space inside the stack region of the memory using the *alloca* instruction. It returns a pointer to the mentioned location, which is stored in a local register. When an operation is performed, the required variable is fetched from the memory by means of a *load*, and it is put inside an internal register. As regards arrays, the required section in the stack frame has to be allocated as well at the beginning of the program. When an operation requires access to an array, the *getelementptr* instruction is used, as it returns a pointer that can be stored in a register. Then, it is used by a load operation as an address in order to retrieve the correct element of the array from the memory.

The objective of the alias analysis performed by InfoCollector is to keep track of the relationship between the values returned by alloca instructions and:

- For every **single variable**, the list of all *local registers where the same variable has been stored*.
- For each **array**, the list of *pointers used to access it*.

The data structure that has been exploited to specify the mentioned relations is a C++ *map*, named **aliasInfoMap**. It consists of an *associative array* composed of elements called *pairs*, each one having a unique *key* identifier and a related *value* field. This arrangement perfectly suits the modelling of information required by this task, as the creation of a univocal connection between the two parts mentioned above can be implemented. The strategy adopted to populate the data structure is reported in the following:

- When an **alloca** is detected, a new pair is created using its destination register as the key field, and it is inserted in aliasInfoMap. The associated value is left empty, waiting to be filled.
- When a **load** is found, if the operand of the instruction corresponds to a key inside aliasInfoMap, its destination register is added to the list of aliases in the related value field. In Listings 4.1 and 4.2, an example C code is provided along with the related LLVM IR, in which local identifiers %4 and %5 are the aliases respectively of %1 and %2.

Example of alias analysis for *load* instructions

```
void foo(){
    int A, B, C;

    C = A + B;
}
```

Listing 4.1. C code

```
define dso_local void @foo() #0 {

    %1 = alloca i32, align 4
    %2 = alloca i32, align 4
    %3 = alloca i32, align 4
    %4 = load i32, i32* %1, align 4
    %5 = load i32, i32* %2, align 4
    %6 = add nsw i32 %4, %5
    store i32 %6, i32* %3, align 4
    ret void
}
```

Listing 4.2. LLVM IR code.

- When a **getelementptr** is identified, its operand is searched among the various keys already present in aliasInfoMap, as it must refer to a previously allocated array. If it is found, the destination register of the getelementptr instruction is inserted in the list of aliases contained in the related value field. The example C code and the related LLVM IR are provided below in Listings 4.3 and 4.4. Registers %11 and %17 contain the pointer used for the access of arrays A and C, respectively identified in the LLVM IR with %1 and %3.

alias analysis for *GEP* instructions

```
void foo(){
    int A[10], B, C[10];

    for(int i = 0; i < 10; ++i){
        C[i] = A[i] + B;
    }
}
```

Listing 4.3. C code

alias analysis for *GEP* instructions

```
define dso_local void @foo() #0 {
  %1 = alloca [10 x i32], align 16
  %2 = alloca i32, align 4
  %3 = alloca [10 x i32], align 16
  %4 = alloca i32, align 4
  store i32 0, i32* %4, align 4
  br label %5

5:
  ; Loop Header

;Loop Body
8:
  %9 = load i32, i32* %4, align 4
  %10 = sext i32 %9 to i64
  %11 = getelementptr inbounds [10 x i32], [10
    x i32]* %1, i64 0, i64 %10
  %12 = load i32, i32* %11, align 4
  %13 = load i32, i32* %2, align 4
  %14 = add nsw i32 %12, %13
  %15 = load i32, i32* %4, align 4
  %16 = sext i32 %15 to i64
  %17 = getelementptr inbounds [10 x i32], [10
    x i32]* %3, i64 0, i64 %16
  store i32 %14, i32* %17, align 4
  br label %18

18:
  ; Loop Latch

21:
  ret void
}
```

Listing 4.4. LLVM IR code

- A **sext** instruction may be present right after a load in order to perform a sign extension of the value fetched from the memory. This is done only if the operation that successively elaborates the operand requires it. In these cases, the destination register of the sext instruction is considered as an alias of the variable retrieved by means of the previous load.

The data structure created by means of the described process is a *support object* to the other operations performed by InfoCollector, as well as to the scheduling phase. As a matter of fact, each time an operation has to be carried out, alias registers are created either due to load or getelementptr instructions. Hence, the discussed analysis is fundamental to quickly identify which variable or array is addressed.

InfoCollector highly exploits aliasInfoMap during its exploration of loops and pointers, mainly in order to efficiently recognize the loop iterators that contribute to the formation of pointers and the arrays visited by means of the latter. On the other hand, the scheduler needs this type of information to properly construct the list of instructions to be mapped on the final LiM system.

4.2.2 The handling of loops

As already pointed out, Octantis highly focuses on *for-loops*, as their proper management can allow for the implementation of parallel execution of operations on a LiM architecture. As a matter of fact, the tool performs loop unrolling whenever such structure is identified in the code, supposing *no correlations are present among different iterations due to data dependencies*. They indeed constitute the main threat to concurrent execution, and they must be avoided as much as possible in the development of the algorithm.

However, if loop unrolling is effectively carried out, *multiple operations sharing the same time frame can be mapped on a LiM system*, thus fully exploiting its intrinsic parallel capabilities. While the overall area occupation of the final circuit may become quite large, growing with the complexity of the input program, timing performance benefits significantly, remaining rather low even with more elaborated algorithms.

The main objective of the strategies and data structures introduced by InfoCollector is to broaden the scope of loop unrolling by enabling the handling of *loop nests*. The support of these advanced constructs allows for further exploration of the potential of LiM systems. In the following paragraphs, the general structure of LLVM loops is examined, in order to help understand the implemented techniques regarding the management of nested loops.

The loop structure in the LLVM IR

The official LLVM documentation provides a deep description of the structure of loops in LLVM at[22]. LLVM internally represents the input program by means of a *control-flow graph (CFG)*, where nodes and edges identify the collection of paths that might be traversed during the execution of the program itself. As regards LLVM, each node corresponds to a different basic block in the IR, and a loop can be identified as a subset of nodes from the CFG with the following properties:

1. the sub-graph that contains all the edges from the CFG within the loop is *strongly connected*, which means that every node is reachable from all others.

2. All edges coming from outside the sub-graph point to the same node called the *header*. As a consequence, every execution path to any of the other nodes of the loop have to pass through the header.

3. The loop is the maximum subset with these properties. Hence, no additional nodes from the CFG can be added such that the induced sub-graph would still be strongly connected and the header would remain the same.

The loop structure defined above is usually referred to as **natural loop**. Each of its internal basic blocks has specific properties and contains different types of useful information that can be exploited for both analysis and optimization. Furthermore, the *loop-simplify* Pass, which is already taken into account in the first version of Octantis, performs a normalization of the natural loop, thus creating a fixed loop structure. The availability of a constant arrangement represents a great advantage for developers, as an algorithm can be easily identified leveraging this well-defined organization. Hence, the classification of the nodes and edges that characterize a natural loop is fundamental, and it is provided in the following:

- **Header**: as already mentioned, it is the loop entry node.

- **Pre-header**: it is the *only predecessor* of the header node. It is always executed before entering the loop.

- **Latch**: it is the node from which the edge (*back-edge*) that returns to the header starts. It is unique and it is always executed before starting a new iteration.

- **Back-edge**: it is the edge from the latch to the header, and it is *unique*.
- **Body**: it is the set of nodes that can be traversed to reach the latch. The basic blocks that compose the loop body are the ones that contain the operations effectively carried out inside the loop itself.
- **Exit-block**: it is the target node of the header, reached when the loop is exited. Hence, it is always executed after exiting the loop.

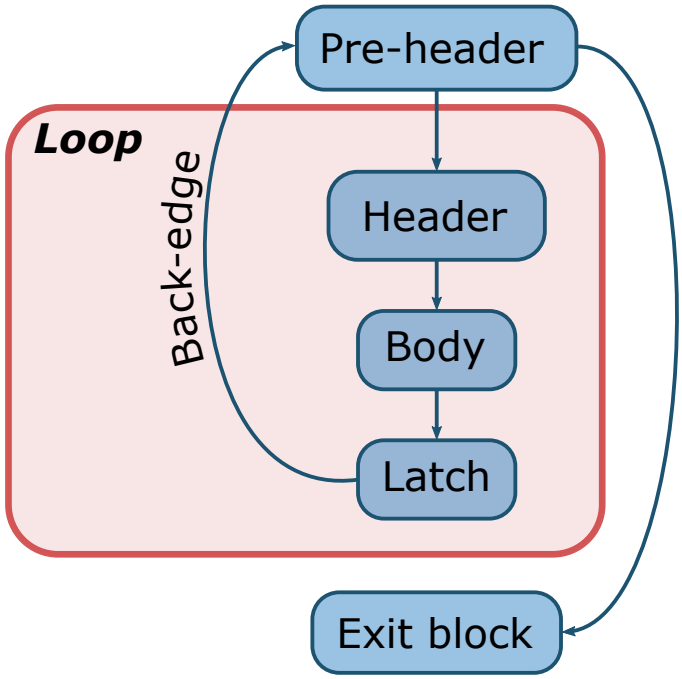

Figure 4.1. Structure of an LLVM natural loop

Figure 4.1 shows the structure of the natural loop described above. The use of the presented nodes can be extended for the characterization of nested loops. In this case, the loop-simplify Pass ensures the creation of a *Chinese-box* structure, as depicted in Figure 4.2, where each loop preserves its well-defined nodes. The comprehension of this regular organization has been fundamental for the definition of the implemented strategies aimed at the collection of information about loops.

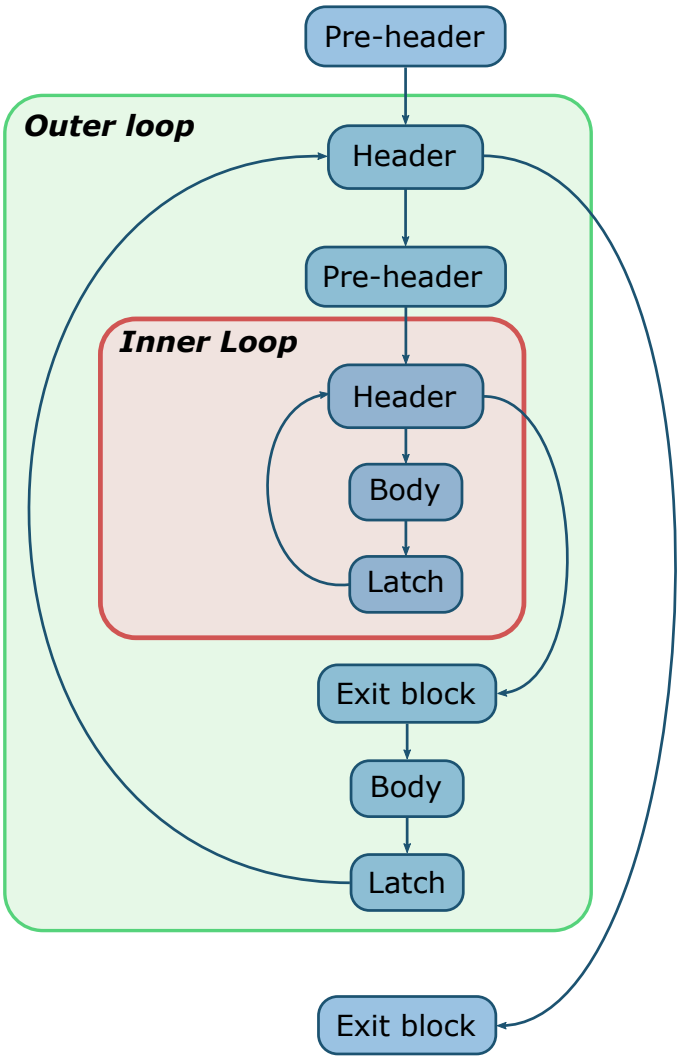

Figure 4.2. Structure of a loop nest composed of two loops after the loop-simplify Pass has been issued

The strategy for gathering loop information

Before starting the discussion about the implemented techniques, it is useful to highlight the characteristics that for-loops must have to be compliant with the new Octantis synthesis process. The introduced novelties allow the tool to effectively gather information regarding:

- The presence of **multiple loops nests** in the program, each with an arbitrary number of inner for-loops.
- The **nesting order of loops** inside a single nested structure.
- The **nodes of a single for-loop** and its **iterator**, which must satisfy several constraints:
  - its *initial value* must be constant.
  - its *final value* can be either a constant or an iterator belonging to an outer loop with respect to the current one.
  - its *increment* must be constant.

Hence, InfoCollector addresses the collection of data regarding both *single loops* and their arrangement towards the creation of *loop nests*. A class called **LoopInfoTable** has been designed specifically to handle these types of information, providing suitable internal data structures that are presented later. The following dissertation deals with the two mentioned aspects separately, along with an explanation of the relative adopted strategies.

As regards single for-loops, the exploration of the LLVM IR code performed by InfoCollector focuses on the *extraction of parameters* that characterize them, such as their iterators, their boundaries and the basic blocks that compose them. The LLVM analysis pass called *LoopInfoWrapperPass* has been exploited to obtain the list of loops that are present in the code, which are provided as instances of the LLVM *Loop* class. This class also implements methods to access the nodes of a single loop, whose investigation is useful to retrieve fundamental features. As a matter of fact, several basic blocks always contain specific operations mainly involving iterators:

- In the pre-header node, the *initial value* of the loop iterator is stored inside related memory location.

- The header performs the comparison between the current value of the loop iterator and its *final value* by means of an *icmp* instruction. Based on its outcome, the control is handed over to the loop body or the exit-block.
- In the latch, the *increment* is added to the iterator variable, which is stored again in memory.

For each loop in the program, InfoCollector retrieves these blocks and analyzes them in order to detect the mentioned parameters. The obtained information is stored inside **loopInfoMap**, which is an internal map data structure of the LoopInfoTable class, depicted in Figure 4.3. loopInfoMap allows to keep track of the relationship between each loop, univocally identified by means of its loop iterator, and the details about its nodes and parameters.

LoopInfoTable contains another map named **nestedLoopMap** that is used to model the arrangement of loops in nested structures. As shown in Figure 4.3, its main aim is to store, for each innermost loop of a nest in the program, the list of its outer loops. As done for loopInfoMap, all of them are identified using their iterator. LoopInfoWrapperPass has been taken again into account, as it provides a method that enable the access to the *parent* of a given loop, namely the loop that directly contains the current one. Starting from the innermost loop and exploiting this method, all the outer ones are obtained in order. This structure is fundamental as it directly implements the "central" part of a LAD graph discussed in Chapter 3.

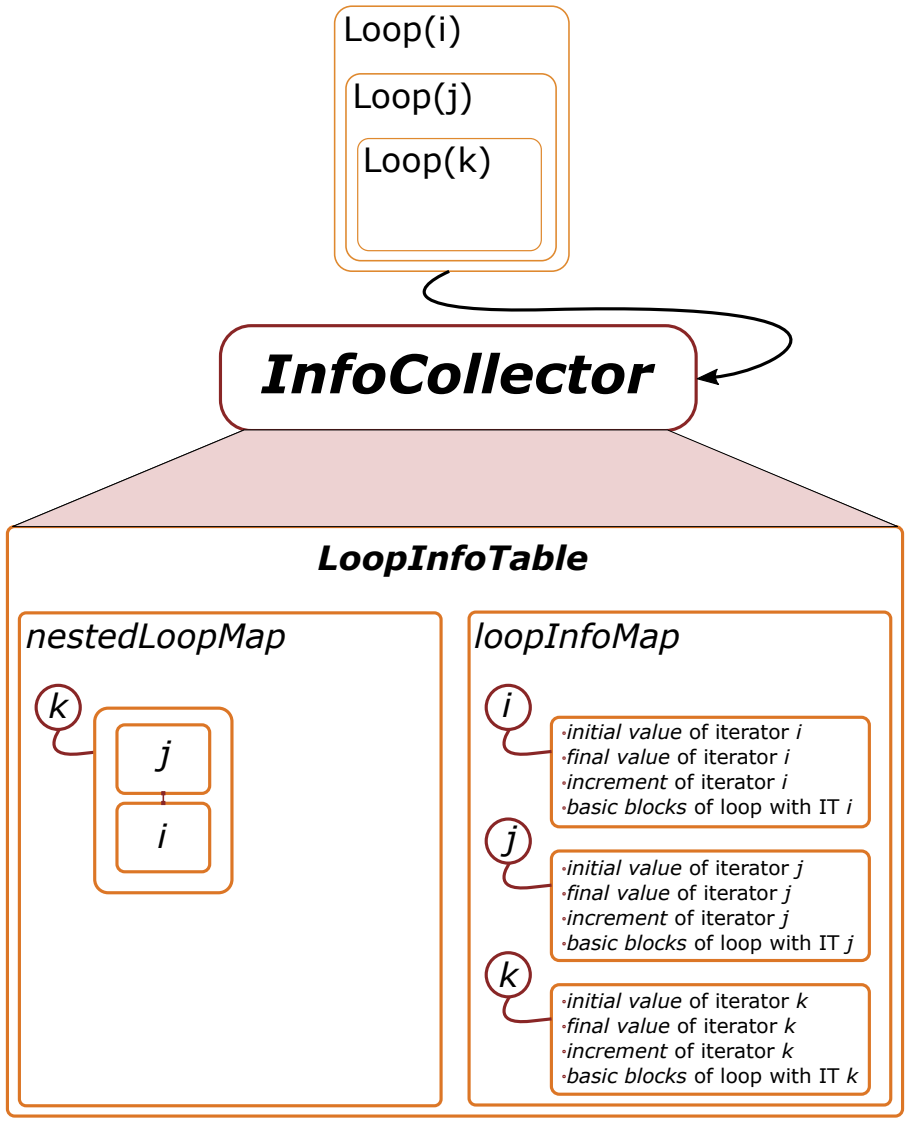

Figure 4.3. A simplified example program composed of 3 nested loops is considered. *LoopInfoTable* and the related organization of data inside *loopInfoMap* and *nestedLoopMap* are detailed.

4.2.3 The handling of pointers

In LLVM IR, arrays are *multidimensional data structures* whose values are stored inside a set of adjacent memory addresses. Performing the access to an array is not as immediate as fetching a single variable, which can be easily loaded from memory and stored again by means of *load* and *store* instructions. In order to retrieve an element from a specific position inside the

array, a more complicated procedure is required, in which an essential role is played by **pointers**. They represent a particular type of variables that is responsible for storing the address of a memory location. A local register that contains a pointer can be effectively exploited by a load in order to fetch the desired value of an array.

The access to multidimensional structures is typically performed inside loops, as *many algorithms require traversing arrays following a well-defined pattern*. Therefore, loop iterators usually contribute to the formation of the indices of an array. As a consequence, in LLVM IR, the construction of memory addresses by means of pointers is tightly connected to loop iterators. Hence, the main goal of *pointer analysis* is to *identify which iterators are used for the creation of each pointer*. This operation is crucial for the realization of a structure inspired to the LAD graph and the subsequent detection of array access patterns, which are useful for the proper mapping of operations on the final LiM system.

The combination of the enhanced loop unrolling capabilities and the handling of more complex access patterns will take advantage of the regular structure of a LiM architecture, in order to achieve high performance due to the parallelization of operations.

Array access in the LLVM IR

The access to aggregate data structures is performed by means of pointers, which are obtained through a **getelementptr** instruction. Although it can be exploited to generate addresses for whatever user-defined *Struct*, Info-Collector only takes them into account with respect to *one/two-dimensional arrays*, as they represent the structures effectively handled by the Octantis synthesis process. In order to better explain how getelementptr works, two examples are examined.

The C code and the relative LLVM IR of the first example are respectively shown in Listing 4.5 and Listing 4.6. In the C implementation, two *Structs* are declared, and function *foo* returns the pointer to an element of array *B* in *RT*, which is in turn contained in *ST*.

Example C code

```
struct RT {
  char A;
  int B[10][20];
  char C;
};
struct ST {
  int X;
  double Y;
  struct RT Z;
};

int *foo(struct ST *s) {
  return &s[1].Z.B[5][13];
}
```

Listing 4.5. C code for the first example

Example LLVM IR

```
%struct.RT = type { i8, [10 x [20 x i32]], i8 }
%struct.ST = type { i32, double, %struct.RT }

define i32* @foo(%struct.ST* %s)
{
entry:
  %arrayidx = getelementptr inbounds %struct.ST,
      %struct.ST* %s, i64 1, i32 2, i32 1, i64 5, i64 13
  ret i32* %arrayidx
}
```

Listing 4.6. LLVM IR code for the first example provided

In the LLVM IR code, the return value of the function *foo* can be directly generated by means of a getelementptr instruction. As it can be noticed by analyzing this operation, more than one argument is present:

- The first parameter is always the **type** used as the basis for the calculations, *%struct.ST* in the example.
- The second argument is always a **pointer**, and it identifies the *base address* to start from. In the example, it is represented by *%struct.ST **,

a pointer to *%struct.ST*, which is a structure composed by the following elements : *i32, double* and *%struct.RT*.

- The remaining parameters are **indices** used for the creation of the correct address of the required element. The first of them always refers to the pointer given as the second argument. In the example, the third parameter points to *%struct.ST**, thus returning the structure type %struct.ST. After that, a set of chained operations is performed, in which each index parameter points to a value of the type retrieved by means of the previous index. In the case of the presented example, the fourth argument points to the third element of the ST structure previously obtained, yielding a *%struct.RT* type. The next parameter indicates the second element of *RT*, returning a *[10 x [20 x i32]]* type, an array. The last two are exploited to point to an element of the mentioned array in order to obtain the final pointer to a *i32* type value.

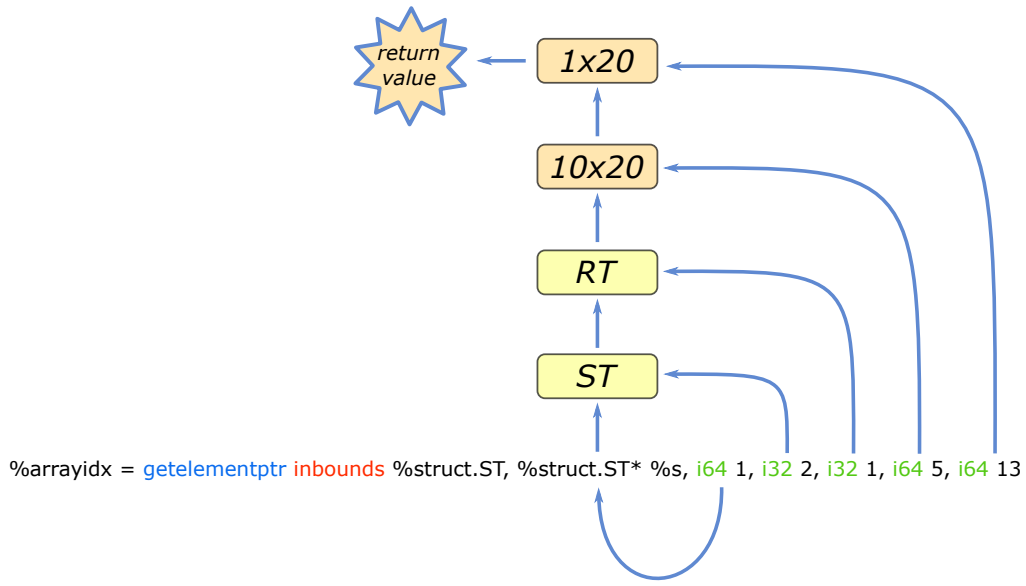

Figure 4.4. Visual representation of how the parameters of a *getelementptr* instruction are used to obtain the final pointer in the first example provided.

Figure 4.4 allows to visualize the discussed functioning. It is also possible obtain "partial" pointers while indexing a structure. Hence, the calculation of the final address can be performed by using more than one getelementptr, achieving the same correct result. This is the case of the LLVM IR code

snippet provided in Figure 4.5. Only the part of code responsible for the access to a 8x8 matrix of 32-bit integer values using loop iterators is reported.

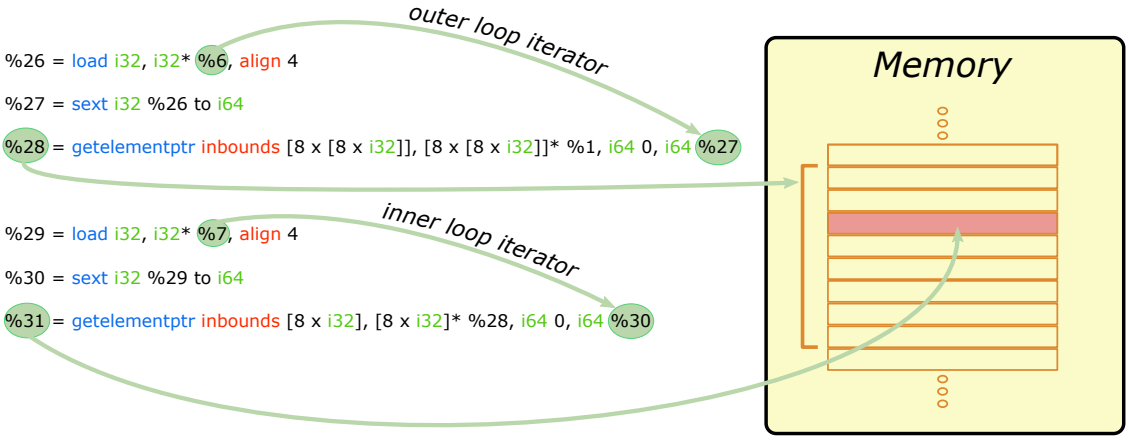

Figure 4.5. Code snippet related to the second example examined. The use of loop iterator in *getelementptr* instructions is highlighted.

In this example, two separate getelementptr are employed to obtain the desired address. The first returns a pointer to a 1-dimensional array of 8 integer elements, which corresponds to an entire row of the allocated matrix. In this case, the variable that contains the iterator of the outer loop is used as index. The generated pointer is taken into account by the second instruction in order to retrieve the address of the required element, using the iterator of the inner loop as index.

In conclusion, the address calculated by means of getelementptr can be exploited either by a load to fetch the needed value from memory, or by a store to put the elaborated data inside the pointed memory location.

The strategy for gathering pointer information

With the introduction of InfoCollector, Octantis is now able to handle accesses to one/two-dimensional arrays inside loops. However, some constraints must be respected mainly regarding the composition of the indexes of an array, which, in order to be effectively managed by the new pass, can be formed by:

1. a single loop iterator;
2. the sum of two loop iterators;

3. the sum of an iterator and a constant;

Although this set is quite limited, it ensures the possibility to exploit many different access patterns. A C++ class named **PointerInfoTable** has been developed specifically in order to handle the information obtained from the analysis of getelementptr instructions. **pointerInfoMap** is a map data structure, internal to PointerInfoTable, that stores, for each pointer used to perform access to an array, useful information mainly regarding the loop iterators exploited for its definition.

Each time InfoCollector identifies a getelementptr, it analyzes its arguments in order to understand which iterators and constant values contribute to the formation of the pointer under study. After that, the same pointer and its related information are inserted in pointerInfoMap as a pair. Hence, *each entry of pointerInfoMap is univocally identified through its pointer operand.* As also highlighted in the second example of the previous section, there is a tight correspondence between a pointer in the LLVM IR and the indexes of an array in the C program. As a consequence, the described analysis is fundamental to effectively *recognize the order in which the input algorithm performs the access to an array.*

Since Octantis aims at supporting the synthesis of 1/2-dimensional arrays, *two main cases can occur.*

The first takes into account the **access to linear arrays**, which are characterized by a single dimension. Hence, only one getelementptr is needed to generate a suitable pointer for these structures. The information about how the index of the array is composed can be obtained by analyzing the *fourth argument* of the instruction, which is represented by a local identifier. InfoCollector explores the operations contained in the loop body in order to know which iterators and/or constants have been considered for the creation of that operand. The *aliasInfoMap* data structure produced by means of alias analysis is highly exploited during this task to properly recognize the operands of each instruction that InfoCollector takes into account. Once the required information regarding the pointer has been identified, the gathered data are stored in pointerInfoMap.

As regards the **access to matrices** (2-dimensional arrays), two getelementptr instructions must be used to define a suitable pointer, since two

indices are required. Each of them is handled following the procedure described above. However, only the pointer operand returned by the second getelementptr is inserted inside pointerInfoMap, along with the information collected from the analysis of both indices.

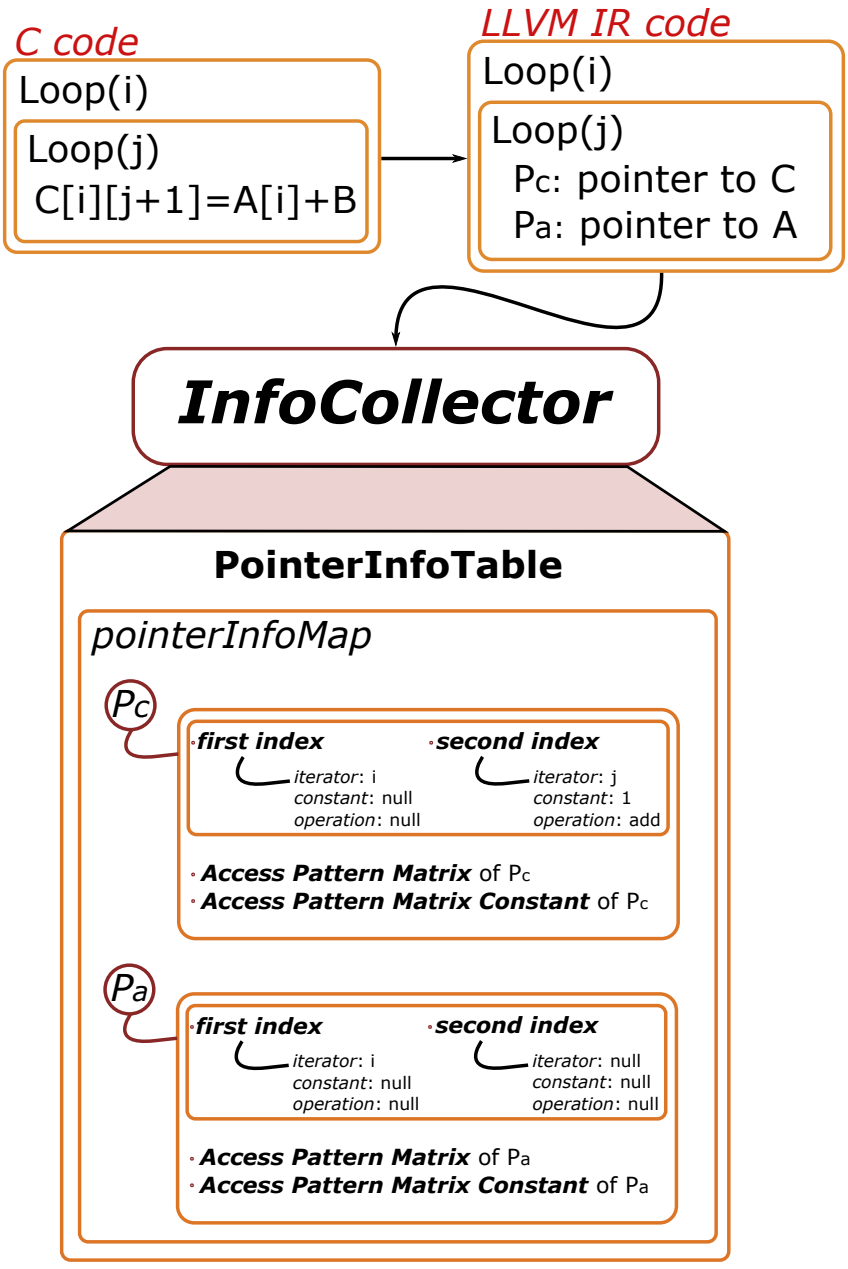

Figure 4.6. Internal structure of *PointerInfoTable* for a simplified code.

Figure 4.6 shows the internal organization of pointerInfoMap for a simplified input program in which a vector C and a matrix A are employed. As a result, two pairs are inserted in pointerInfoMap, one for each pointer used to perform the access to the arrays, respectively named P_c and P_a. On the other hand, the related information are stored in the value field of each pair. Along with the composition of each index, the APM and the APMC of each pointer are stored, even though they are obtained by means of the analysis of the LAD graph structure examined in the next section.

4.3 The construction of Access Pattern Matrices

In the previous discussion, the data structures responsible for storing information about parameters characterizing single loops, their order inside loop nests and the composition of pointers have been examined. The combination of the information handled by LoopInfoTable and PointerInfoTable leads to the generation of a structure totally equivalent to the LAD graph presented in the previous chapter.

As a matter of fact, nestedLoopMap is devoted to the representation of the order in which loops are nested, identifying each of them by means of their iterator. On the other side, pointerInfoMap aims at reconstructing the composition of the indexes of an array, which make use of loop iterators, too. Hence, they constitute a connection point between the two structures, thus creating a LAD graph, as shown in Figure 4.7.

Exploiting the availability of such an infrastructure, InfoCollector performs a detailed *analysis of all pointers in pointerInfoMap with the purpose of creating their related APM and APMC*. These matrices will be explored during the binding phase in order to detect the actual access pattern of the array, which is a fundamental information for the correct mapping and parallelization of operations onto the final LiM architecture. The strategy adopted by InfoCollector for the generation of the APM observes the following steps:

1. A pointer is chosen from pointerInfoMap.

2. The nested structure in which the pointer under investigation resides is fetched from nestedLoopMap.

3. For each index of the current pointer, loops are visited starting from the outermost to the innermost.

4. If the current loop iterator contributes to the index, a 1 is inserted in the matrix, otherwise a 0.

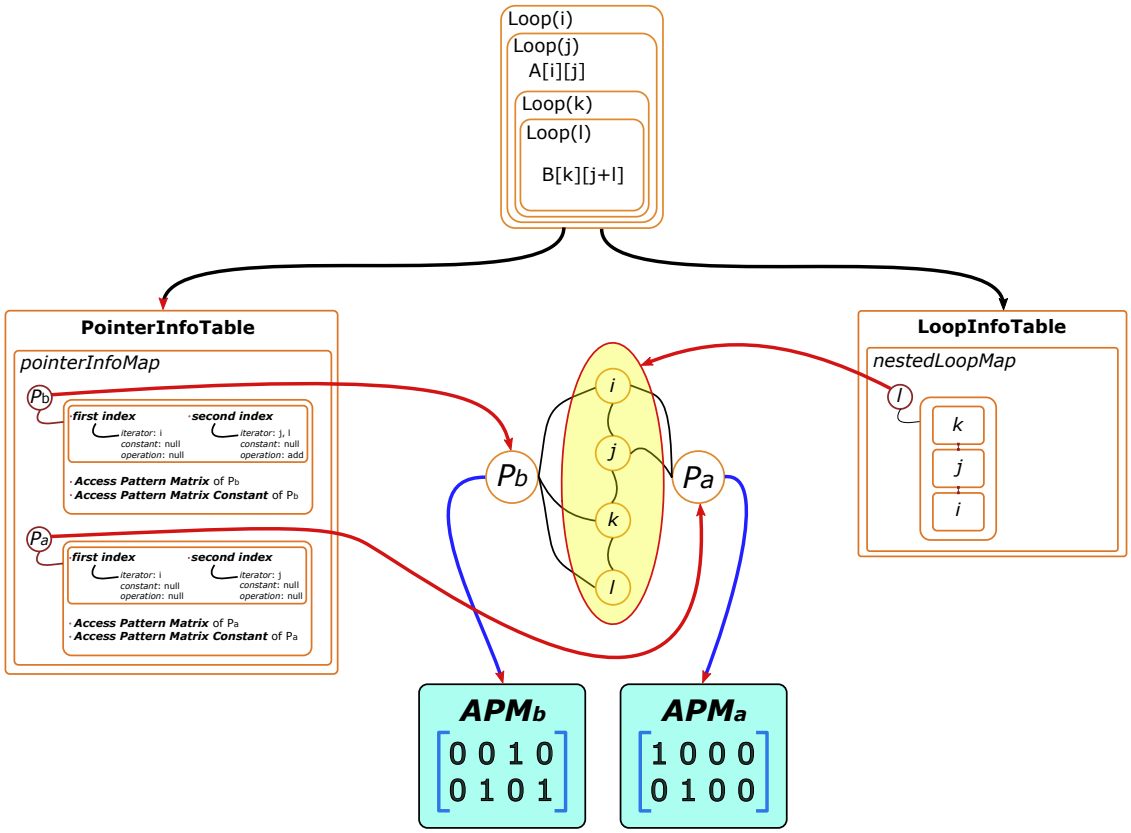

Figure 4.7. A simplified code with 4 nested loops where the access to arrays A and B is provided at the top. The organization of information by means of *PointerInfoTable* and *LoopInfoTable* allows the creation of a *LAD graph*, from which the APMs of the two arrays are extracted.

The creation of the APMC is much simpler, as only the presence of constants must be checked for each index of the array. Hence, visiting the LAD graph is not necessary.

4.4 The identification of *valid* Basic Blocks

During its exploration of the LLVM IR code, InfoCollector also performs an important analysis aimed at the detection of basic blocks whose instructions should not be mapped on the final LiM system. As already highlighted in the previous chapters, only few operations have to be actually scheduled and later implemented on the LiM architecture. On the other hand, the majority of them is only considered for gathering useful information to correctly drive the synthesis process.

Within this scenario, InfoCollector acts as a *filter* interposed between the IR code and the scheduler. It identifies the subset of basic blocks that contain meaningful instructions for the mapping on the LiM system, which are referred to as *valid*, and it gathers them in a data structure that is provided to the scheduler. The described behaviour is depicted in Figure 4.8.

However, even inside valid blocks, several operations can be avoided for scheduling, mainly the ones that generate pointers (getelementptr) or contribute to the definition of indices, such as load, store, sext and add whose operands are loop iterators. Exploiting its other internal data structures, such as aliasInfoMap, InfoCollector also implements suitable *methods* to allow the scheduler to quickly identify these instructions and skip them.

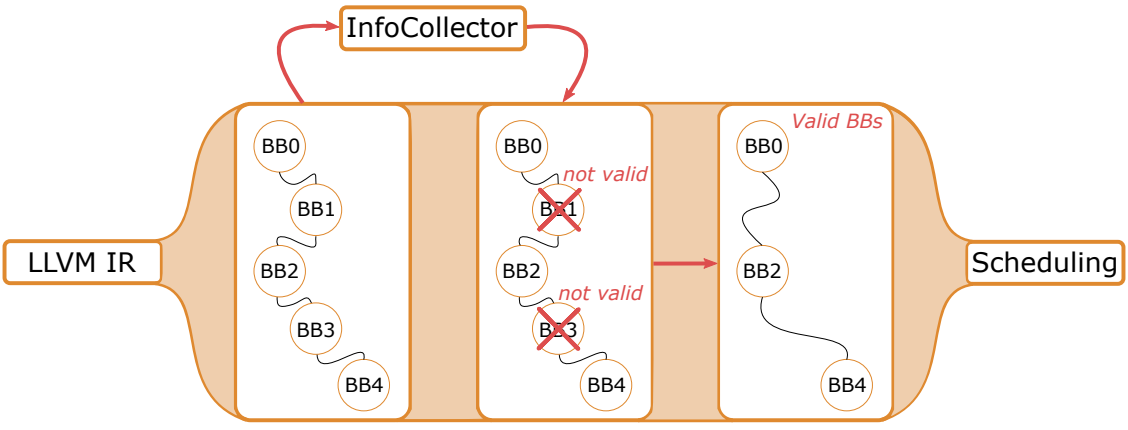

Figure 4.8. Scheme showing the filter-like behaviour of InfoCollector.

Since loops represent the key constructs around which the focus of the new pass revolves, it is useful to explain how the presented analysis addresses the four main blocks that form them:

- The *body* and the *pre-header* of a loop are considered valid. The former is the most important basic block, as it contains the main operations of the algorithm that involves arrays and variables. The latter, especially in loop nests, can include several instructions related to the body of the immediately outer loop. Hence, it is given to the scheduler, which recognizes the useful instructions exploiting the methods provided by InfoCollector discussed above.
- The *header* and the *latch* of a loop are marked as *not* valid, as their only purpose is respectively to initialize the iterator and increment its value.

In conclusion, the presented tasks performed by InfoCollector have been devised in order to improve the performance of the scheduling phase, as it will be better clarified in Chapter 5. This strategy also allows enhancing the modularity of the Octantis structure, by avoiding the implementation of complex logic in the scheduler. Instead, the gathering of this "support information" is addressed by InfoCollector, whose data structures facilitate its detection.

Chapter 5

The evolution of Octantis structure

5.1 Introduction

With the introduction of InfoCollector, both the scheduling and the binding phase have been modified in order to take full profit from the data structures and strategies implemented by the new Pass.

The analysis regarding *alias registers* and *valid basic blocks* carried out by InfoCollector allows the speed up of the scheduling phase. The scheduler can avoid considering all the instructions in the IR code, and it is assisted in the detection of operands. Moreover, the details about pointers collected by the same Pass has to be transferred to the binder for the correct mapping of operations. Hence, the scheduler has the crucial task to *assign each operand with the related pointer*, and convey this information to the subsequent stage through Instruction Table.

The binding phase has undergone a major evolution, due to the fact that it has to handle both the proper mapping and parallelization of operations, which may also involve arrays. In this case, the identification of the *array access patterns* has to be carried out by the binder as well, carefully analyzing the APMs provided by InfoCollector.

The evolution and the improvements brought to the mentioned phases are discussed respectively in Sections 5.2 and 5.3.

5.2 The scheduling phase: leveraging InfoCollector

In the first version of Octantis, the scheduler needed to collect information regarding alias registers and perform checks to avoid instructions that should not be considered for the mapping on the LiM system. Although these operations are essential for the successful execution of this phase, they do not strictly belong to it.

With the introduction of InfoCollector, the implementation of the mentioned tasks is carried out by the new pass before the scheduling stage even starts. This choice allows the scheduler to focus on its main objective, while being supported by InfoCollector, which can address the wide problem of the "collection of information" in a more efficient manner. As already explained, InfoCollector performs alias analysis and detects valid basic blocks, also providing methods for the identification of meaningful instructions inside them. As a consequence, the **performance of the scheduler is improved**. Moreover, its future expansions can rely on the availability of these operations in order to explore new solutions. Furthermore, this design choice also enhances the *modularity* of Octantis structure.

The scheduling phase also represents the *connection point* between the gathering of data about pointers and the actual mapping of operations involving arrays performed by the binder. Hence, the data structure aimed at storing the list of scheduled instructions, called Instruction Table, has been properly modified in order to convey the information regarding pointers.

The main parameters that characterize an operation present in Instruction Table are the *destination operand* and the *source operands*, respectively exploited to contain and generate the final result. As Octantis supports the use of one/two-dimensional arrays, information useful to obtain the related access patterns is needed to be attached to each of the mentioned operands, in case they were arrays, ready to be identified during the binding phase. For this reason, *three additional parameters* have to be specified for each operation in Instruction Table, one for each operand, specifying the pointer employed to perform the access to the related array. The analysis of alias registers carried out by InfoCollector helps the scheduler recognize the operands that need to be associated with a pointer, as they refer either to a linear array or a matrix.

During the binding phase, if the operand that is taken into account is a single variable, the related pointer will be null. However, if a vector or matrix is considered, the pointer is searched in *PointerInfoTable* in order to obtain the information regarding its *APM* and *APMC*, which are later analyzed as explained in Section 5.3 for the correct mapping and parallelization of the operation.

In Figure 5.1, the crucial task that the scheduling phase carries out as junction point between InfoCollector and the binder is depicted.

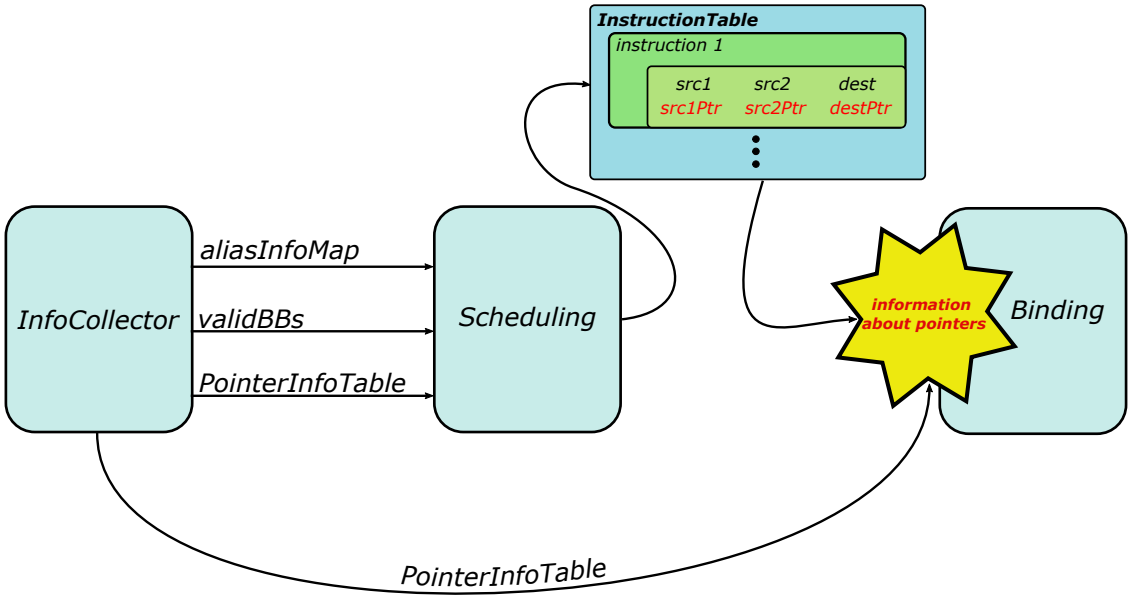

Figure 5.1. Movement of information and data structures among InfoCollector, the scheduler and the binder, with the introduction of parameters regarding pointers in Instruction Table.

5.3 The binding phase: facing higher complexities

The main aim of the binding phase is the actual generation of LiM system that implements the operations required by the input C algorithm promoting parallel execution as much as possible. The first version of Octantis provided efficient solutions for the mapping of the following elaborations:

1. Sum and bitwise operations (and, or, xor, nand, nor, xnor) involving variables outside a loop.

2. Sum and bitwise operations inside a single for-loop, involving both variables and one-dimensional arrays, whose index must correspond to the loop iterator. No data dependencies should be present, allowing the proper unrolling of the loop iterations on the LiM architecture. An example in provided below in Listing 5.1.

Example 1

```
for(int i = 0; i < N; ++i)
    C[i] = A[i] + B;
```

Listing 5.1. Example of operation in a loop.

3. Accumulation of the values of a vector inside a variable. The array is visited by means of a for-loop, its elements are repeatedly summed and the final result is stored in a variable. In this case, a *reduction-tree* strategy, shown in Figure 5.2 is implemented in order to perform the accumulation, which allows the execution time to be $O(log(N))$, where N is the number of items in the array. The C code in Listing 5.2 shows an example of accumulation.

Example 2

```
for(int i = 0; i < N; ++i)
    S = S + A[i];
```

Listing 5.2. Example of accumulation.

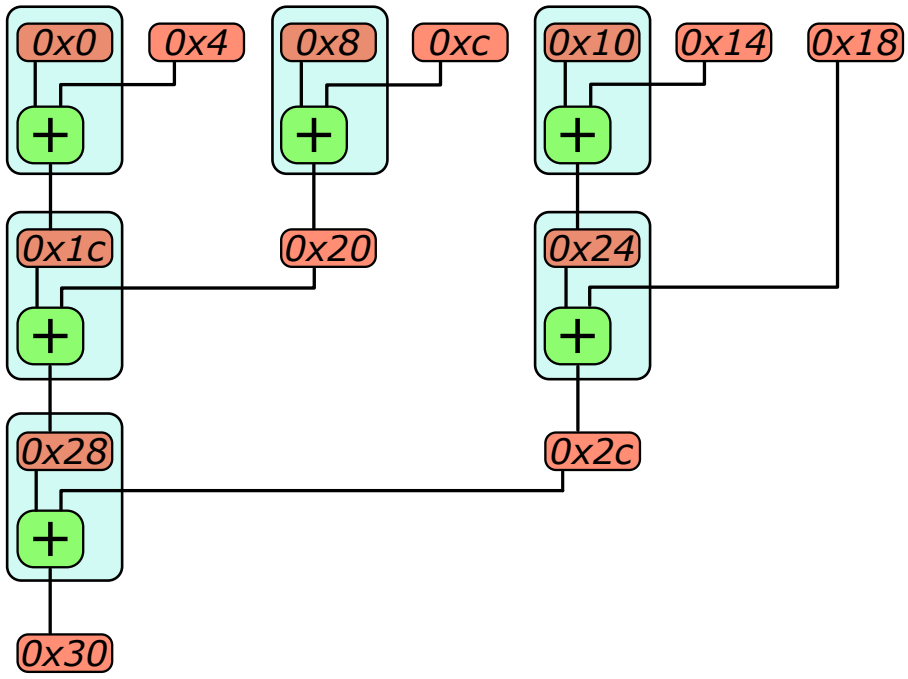

Figure 5.2. Mapping of the accumulation operation on an set of LiM rows implemented following a *reduction-tree* strategy.

Along with the higher complexity of the input program, Octantis binding phase has evolved in order to correctly manage the expansion of the operations mentioned at points 2 and 3. As a matter of fact, *loop nests allow multiple array access patterns to be exploited*. The information about them is taken into account by the binder thanks to the data regarding pointers coming from InfoCollector and effectively conveyed by means of the scheduler. Data dependencies should be avoided as much as possible, in order to enable loop unrolling and the subsequent high parallelization of operations on the final LiM system.

New strategies have been also introduced to address the handling of more than one accumulation set at a time. This kind of operations could lead to a great amount of needed hardware resources as well as a huge final area occupation. In order to tackle these issues, **target-dependent optimization techniques** have been implemented with the purpose of promoting the reuse of intermediate results stored in LiM rows that have already been mapped.

The following sections deal with the evolution that the Octantis binding

phase has undergone, in order to obtain array access patterns from APMs and APMCs, manage more elaborated accumulations, and perform suitable optimization steps aimed at reducing the overall area of the LiM system.

5.3.1 Handling array access patterns

One of the main goals of InfoCollector is to assign each pointer in the LLVM IR, which corresponds to an array instance in the C code, with an APM and an APMC, whose construction is discussed in the Chapter 4. The binding phase has been modified in order to properly handle these data structures towards the extraction of the related array access patterns. Their identification represents a key point around which the correct mapping and parallelization of operations involving 1/2-dimensional arrays on LiM systems revolves. As a matter of fact, they provide crucial information regarding the order in which elements of the matrices or vectors used in the algorithm are visited, thus allowing the correct interconnections among LiM rows that belong to different array operands to be carried out.

Before starting the examination of the new structure of the binder and the implemented strategies, it is useful to highlight several characteristics about the operations supported by Octantis synthesis process. As already mentioned in the previous section, two main categories of operations can occur inside a for-loop, and different solutions and optimization techniques must be devised for their proper mapping on a LiM architecture. These two groups along with their main features are presented in the following:

- **Fully-parallel (FP) operations** are the ones that highly benefit from the implementation of the loop unrolling technique, thus exploiting the entire potential of the intrinsic parallel capabilities of a LiM architecture. This is due to the total absence of data dependencies, which require the destination operand not to appear among the source ones. They can be represented by both bitwise and sum operations that occur inside loop nests. An example of this kind of elaborations is provided in Listing 5.3:

Example of fully-parallel operation

```
for(int i = 0; i < N; ++i)
    for(int j = 0; j < M; ++j)
        C[i][j] = A[j] ^ B[i];
```

Listing 5.3. FP operation.

- **Accumulations** require the sum of all elements that belong to a given set. This kind of elaboration usually exploits a temporary variable inside which the values are repeatedly accumulated, as shown in the following example:

Example of accumulation

```
for(int i = 0; i < N; ++i)
    for(int j = 0; j < M; ++j)
        S = S + A[j][i];
```

Listing 5.4. Accumulation.

  Since a data dependency is present, a complete parallelization is not feasible. Accumulations need ad hoc optimizations and mapping techniques, since operands stored in multiple LiM rows that belong to the same array must be summed together. However, in a LiM system, differently from a serial implementation, data that must be accumulated are all available for calculation in parallel. Hence, as mentioned in the previous section, a reduction-tree mapping technique is adopted in order to let the overall execution time be $O(log_2(N))$, instead of typical $O(N)$ that characterizes the serial approach.

The new internal organization of the binder structure reflects the discussed separation, even though both types of operations take great advantage from the exploitation of APMs and APMCs. The binder can access to PointerInfoTable and obtain the APM and APMC of each pointer present in the code. The related array access patterns are retrieved by means of the analysis of the APMs.

The procedure that allows the retrieval of array access patterns starting from APMs is based on the identification of specific *pre-defined templates* inside the APM itself. As a matter of fact, special arrangements of the elements belonging to APMs can be directly associated with well-defined ways of visiting an array, which are also the most employed ones, as suggested in [23]. Octantis synthesis process can now effectively handle and detect several APM templates for the correct mapping of operations that make use of 1/2-dimensional arrays.

Since APMs assigned to vectors consist of 1xN matrices, where N is the number of nested for-loops, the amount of related templates is much smaller than the one characterizing APMs of matrices, whose size is 1xN. In the following, the list of pre-determined arrangements of APMs supported by Octantis synthesis process is provided, along with the C code from which they are extracted. Also, a brief description follows each case. Firstly, the set of APMs related to matrices are analyzed.

Example C code	**APM of A**

`for(int i = 0; i < N; ++i)`
`  for(int j = 0; j < M; ++j)`
`  A[i][j];`
 	
Listing 5.5. Row-major order.
	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

As it can be easily noticed by the C code in 5.5, this APM identifies the most common way in which a matrix can be accessed, usually referred to as *Row-Major order*. All the elements belonging to a row are visited before moving on to the next one, until the end of the RxC array.

Example C code	APM of A

`for(int i = 0; i < N; ++i)`
`  for(int j = 0; j < M; ++j)`
`  A[j][i];`
 	
Listing 5.6. Column-major order.
	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$

The so-called *Column-Major order* is associated with the APM of array A in Listing 5.6. It requires all the elements in a column of the RxC matrix to be visited before moving on the next one, until the end of the array. It allows traversing the array in the opposite way with respect to the row-major order.

Example C code	APM of A

`for(int i = 0; i < N; ++i)`
`  for(int j = 0; j < M; ++j)`
`  for(int k = 0; k < L; ++k)`
`  A[i+k][j]`
 	
Listing 5.7. Vertical LX1 subsets.
	$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$

As it can be noticed in the APM related to matrix A of Listing 5.7, its 2x2 rightmost part corresponds to the APM relative to the column-major order. It is indeed a slightly modified version, since a Lx1 subset of a column is identified and all of its elements must be accessed before another set in the next column is considered.

Example C code	APM of A

`for(int i = 0; i < N; ++i)`
`  for(int j = 0; j < M; ++j)`
`  for(int k = 0; k < L; ++k)`
`  A[j][i+k]`
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$

Listing 5.8. Horizontal 1XL subsets.

In the case of Listing 5.8, the 2x2 rightmost part of the matrix coincides with the APM relative to the row-major order. It requires a subset of a row to be identified and all of its elements must be accessed before another set in the next row is handled. This access pattern represents the opposite of the previous one, as it consists of a modified version of the row-major order.

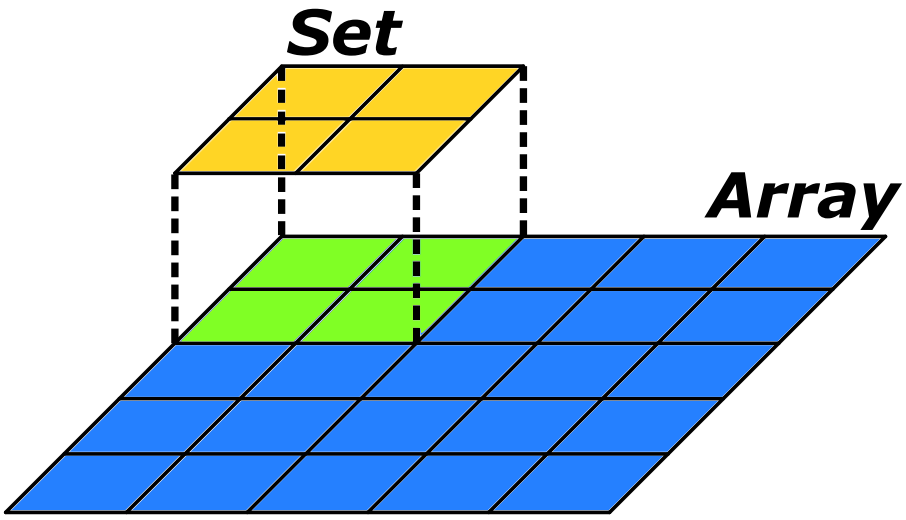

Figure 5.3. Identification of a set inside an array.

From this point forward, the four types of 2x4 APMs considered enable the identification of multiple MxN subsets while traversing the RxC array, with $M < R$ and $N < C$. In order to better understand the way in which elements are visited in these situations, the APMs can be divided in two 2x2 sub-APMs. The rightmost one indicates the order in which the elements of a subset are accessed, while the analysis of the leftmost one allows to know how subsets "move" inside the array. In order to visualize this behaviour,

Figure 5.3 has been provided. The yellow square represents a mask that shifts along the matrix underneath and, at each movement, covers a different part, highlighted in green. This subpart represents the set of elements that is taken into account.

Example C code	APM of A

`for(int i = 0; i < N; ++i)`
`  for(int j = 0; j < M; ++j)`
`  for(int k = 0; k < L; ++k)`
`  for(int l = 0; l < P; ++l)`
`  A[i+k][j+l]`
 	
Listing 5.9. Array A yields a Row-major/Row-major APM.
	$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$

In the case related to Listing 5.9, both the leftmost and the rightmost APMs corresponds to the one of the row-major order. This means that all the elements of each MxN subset are accessed in that order. Moreover, also the "movement" of sets follows the same order.

Example C code	APM of A

`for(int i = 0; i < N; ++i)`
`  for(int j = 0; j < M; ++j)`
`  for(int k = 0; k < L; ++k)`
`  for(int l = 0; l < P; ++l)`
`  A[j+l][i+k]`
 	
Listing 5.10. Array A yields a Column-major/Column-major APM.
	$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$

In the APM of A in Listing 5.10, both the leftmost and the rightmost APMs corresponds to the one of the column-major order. This means that the array is traversed in the opposite way of the previous case. The order in which different subsets are considered is the column-major one, and also the elements of a MxN subset are accessed in this order.

Example C code	APM of A

`for(int i = 0; i < N; ++i)`
`  for(int j = 0; j < M; ++j)`
`  for(int k = 0; k < L; ++k)`
`  for(int l = 0; l < P; ++l)`
`  A[j+k][i+l]`
 	
Listing 5.11. Array A yields a Column-major/Row-major APM.
	$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$

Differently from before, the 2 sub-APMs of the APM belonging to A in Listing 5.11 are not equal. As a consequence the subsets move following the opposite order with respect to the one adopted for the visit of their elements. Specifically for the 2x4 APM of A, the row-major order is considered for the former and the column-major for the latter.

Example C code	APM of A

`for(int i = 0; i < N; ++i)`
`  for(int j = 0; j < M; ++j)`
`  for(int k = 0; k < L; ++k)`
`  for(int l = 0; l < P; ++l)`
`  A[i+l][j+k]`
 	
Listing 5.12. Array A yields a Row-major/Column-major APM.
	$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$

The case provided in Listing 5.12 represents the perfect inverse of the one in Listing 5.11. Elements of a subset are accessed in the column-major order and subsets move in the row-major one.

As regards vectors, the APMs templates that the binder can handle are presented hereinafter.

Example C code	APM of A

`for(int i = 0; i < N; ++i)`
`  A[i];`
 	
Listing 5.13. Trivial access to vector A.
	$\begin{bmatrix}1\end{bmatrix}$

The code in 5.13 represents the most trivial way in which a vector can be accessed. Element of the 1xN array is visited, from the first to the last.

Example C code	APM of A

`for(int i = 0; i < N; ++i)`
`  for(int j = 0; j < M; ++j)`
`  A[i+j];`
 	
Listing 5.14. 1XM subsets in vector A are considered.
	$\begin{bmatrix}1 & 1\end{bmatrix}$

In the case of the APM of vector A in Listing 5.15, multiple 1xM sets are identified inside the array.

In all of the presented cases, each loop iterator contributes to the formation of at least one index of the array. However, in a more general situation, for-loops whose iterators are not present in any index of an array can occur. As a result, the described APM templates remains unchanged but the matrices are enlarged by the presence of additional *zeros*, which can appear on the left or right with respect to the original APM. In order to allow a better understanding of these cases, two simple example are discussed. The related C code is provided as well as the APM of the vector A.

Example C code	APM of A

`for(int i = 0; i < N; ++i)`
`  for(int j = 0; j < M; ++j)`
`  C[i][j] = B[j][i] + A[j]`
	$\begin{bmatrix} 0 & 1 \end{bmatrix}$

Listing 5.15. Zero before the APM of A.

Example C code	APM of A

`for(int i = 0; i < N; ++i)`
`  for(int j = 0; j < M; ++j)`
`  C[i][j] = B[j][i] + A[i]`
	$\begin{bmatrix} 1 & 0 \end{bmatrix}$

Listing 5.16. Zero after the APM of A.

In the first example, a zero on the left is introduced, due to the fact that iterator i is not included in the index of A. This means that all elements of A are accessed as many times as the number of iterations of the first for-loop. In general:

Consideration 1

If a 1xM or 2xM matrix formed only by zeros *precede* one of the APM templates described, the access to the array must be repeated, according to the template itself, a number of times n, where n is the result of the multiplication performed between all the iterations related to the M for-loops that have generated those zeros.

As regards the second example, the opposite situation occurs. A single element of A remains "fixed" for the calculation of all values that belong to the same row of matrix C. In the general case:

Consideration 2

If a 1xM or 2xM matrix formed only by zeros *follows* one of the APM templates described, each element considered while visiting the array must contribute to a number n of consecutive elaborations, where n is the result of the multiplication performed between all the iterations related to the M for-loops that have generated those zeros.

The discussion regarding array access pattern is useful to better understand their crucial importance for the binding phase. As a matter of fact, alongside with the loop unrolling techniques, their identification and exploitation allows the parallel capabilities of a LiM system to be put at the service of an increased amount of applications that could effectively benefit from it. Hence, the detection of array access patterns represents one of the key features of the *new structure of the binder*, depicted in Figure 5.4.

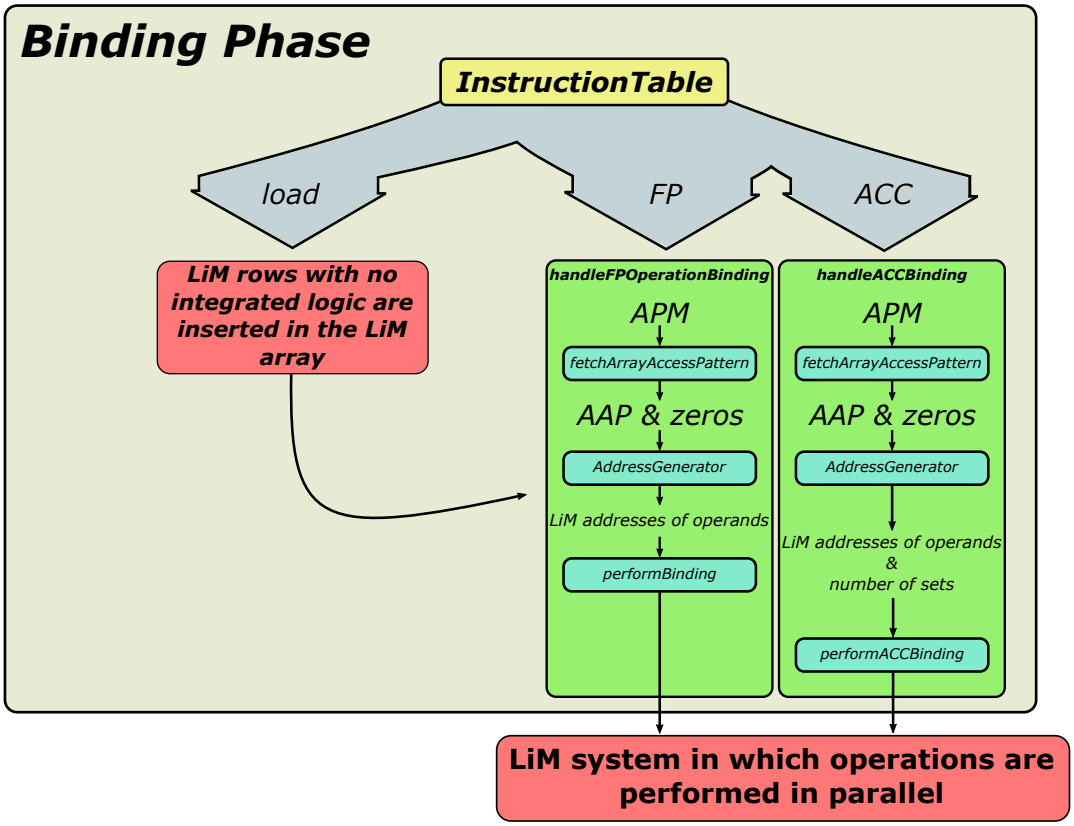

Figure 5.4. New internal structure of Octantis binder.

The main objective of this phase is the mapping of instructions contained in Instruction Table onto a LiM system. As shown in the reference figure, **three main "channels"** can be devised, each one aiming at the correct handling of its related operations, and they are examined in the following:

1. The first channel deals with **load instructions**. They can require the mapping of LiM rows for the allocation of either a variable or an array that is useful for a subsequent elaboration. Hence, the correct amount of rows must be instantiated in order to allow storing the related data structure. The binder keeps track of the relation between each operand and the addresses of its associated rows inside the LiM system. This mechanism enables the proper management of dependencies between instructions.

2. The second and third channel respectively aim at the handling of bitwise and arithmetic instructions, which are carried out between LiM rows related to operands that have been previously allocated in the architecture by means of a load. *The APM of each pointer associated to an operand is fetched in order to enable the detection of eventual array access patterns.* From this point forward, the internal organization of the binder reflects the different type of handling that has to be implemented for the two categories of operations described above:

   - **fully parallel operations**: in this case, the control is given to a dedicated function, called *handleFPOperationBinding*. First of all, the APMs belonging to array operands are given to the *fetchArrayAccessPattern* function. It detects the access patterns templates and zeros related contained into the input APM. Once this task is accomplished, the array access patterns are know, hence the actual mapping of the operation can start. A function called *addressGenerator* takes into account the set of LiM rows associated to each operand in order to create a list that contains the same rows ordered as required by the corresponding access pattern and zeros. At the end of this process, three equally-sized lists are generated, one for each operand. The elements that share the same position inside these lists are the ones that must be elaborated together. LiM rows at the i-th position inside the source operands lists must be taken into account for the mapping of the operation, and the result must be stored in the i-th LiM row of the destination operand list. In case a single variable is used for calculations along with arrays,

it is effectively detected and the same LiM row is exploited for all calculations. The actual mapping task is accomplished by the *performBinding* function, which mainly implements the same strategies already available in the first version of Octantis and described in Chapter 2.

- **Accumulations**: these operations are always characterized by the presence of a temporary variable, which figures both as source and destination operand, and an array. In the previous discussion about APMs templates, the presence of sets has been highlighted, which are mainly exploited by accumulations in order to identify, inside the vector or matrix under consideration, equally-sized subsets whose elements must be summed together. Hence, the access pattern of the array must be retrieved with the fetchArrayAccessPattern function and provided to addressGenerator, which also return the *size of the eventual subsets*. In this way, the ordered list of LiM rows related to the array is "cut" into sub-lists whose dimension is equal to the parameter obtained by means of addressGenerator. One sub-list is considered at a time, and it is given to *performAccumulationBinding*, which implements the mapping of the accumulation adopting the reduction-tree strategy, along with specific optimization techniques described in depth in the following section.

At the end of the binding phase, the final LiM system is obtained along with its FSM, which gives information about the LiM rows that are active in each time frame during the execution of the algorithm. The combination of array access patterns and loop unrolling enables the parallelization of operations on the LiM architecture, thus exploiting its intrinsic parallel structure. The actual composition of the generated circuit is described by means of a dedicated data structure called *LiMArray*, which was already developed during the first version of Octantis. LiMArray and the FSM are provided to the code emission phase for the generation of the output files.

5.3.2 A new target-dependent optimization

Octantis synthesis process does not only aim at the correct generation of the LiM architecture but also at its *optimization*. As a matter of fact, along with the increased complexity of the data structures that can be exploited for the input algorithm, *area occupation and needed hardware resources can become*

rather large. Hence, suitable strategies have been introduced with the purpose of **reducing space occupation** while keeping the overall **execution time as low as possible**, which represents the most valuable benefit that is provided by parallel computation.

Since the first version of Octantis, two main approaches had been devised in order to limit the introduction of unnecessary LiM rows inside the array:

- When operands stored in two different LiM rows are considered for an elaboration, if one of them is made up of cells that do not present any integrated logic yet, the insertion of the logic ports needed to carry out the required operation is performed inside those cells.
- When one of the two LiM rows involved in an operation has already the needed logic for the elaboration, 2-to-1 multiplexers are integrated inside its LiM cells.

These techniques have been preserved in the expansion of the binding phase, as they are implemented in both *performBinding* and *performAccumulationBinding* functions, and an additional one has been introduced. Its main aim consists in the **detection of elaborations between LiM rows that have already been mapped**. This strategy allows a great reduction in area occupation when considering algorithms characterized by multiple fully-parallel operations that exploit the same operands, or accumulations in which the various subsets of an array share several elements.

As regards fully-parallel (FP) operations, a simple example of C code where the discussed optimization can be exploited is provided in Listing 5.17.

Example 1

```
for(int i = 0; i < N; ++i)
  for(int j = 0; j < M; ++j)
    D[i][j] = A[j][i] & B[j][i] & C[i];
    F[i][j] = E[j] & A[j][i] & B[j][i];
```

Listing 5.17. Fully-Parallel operations whose mapping can benefit from the new optimization strategy.

When the binder addresses the synthesis of the AND operations responsible for the generation of the matrix D, it implements the normal flow

described in the previous section. On the other hand, the new optimization step takes place when it takes into account the elaborations for the array F. The same LiM rows used for the mapping of the previous AND between matrices A and B are indeed present in the list obtained by *addressGenerator*. The binder recognizes that the same operation is already mapped on the LiM system and it avoids the introduction of additional rows as well as AND logic ports. Hence, it directly considers the rows that contain the result of $A[j][i]B[j][i]$ for the next mapping.

The advantages brought by this approach are even larger when dealing with *accumulations*. In this case, as explained before, multiple sets inside a single array can be identified by the array access pattern. Depending on the increment related to each for-loop iterator, sets may happen to be *overlapping*, as shown in Figure 5.5. This means that the same elaboration can be shared among multiple sets. A simple example is provided in Listing 5.18 in order to better clarify this condition.

Example 2

```
for(int i = 0; i < N; ++i)
  for(int j = 0; j < M; ++j)
    for(int k = 0; k < P; ++k)
      S += A[j][i+k]
```

Listing 5.18. Accumulation with overlapping sets.

The access pattern characterizing the matrix A requires various 1xM subsets inside the matrix A to be considered for accumulation. Supposing that M is greater than 1, since the increment of the iterator i is 1, two consecutive sets on the same row share two elements. The introduced optimization allows the detection of these kinds of situations, thus enabling a great saving of hardware resources.

However, in case of accumulations, this approach is carried out in a slightly different manner compared to Fully-Parallel operations. Since the great benefit derived from the *reduction-tree* implementation relies on the lowering of the execution time ($O(log_2(N))$), several measures have been exploited in order to keep this parameter as small as possible. The strategy adopted requires keeping track of the **depth** inside the reduction tree of each LiM row

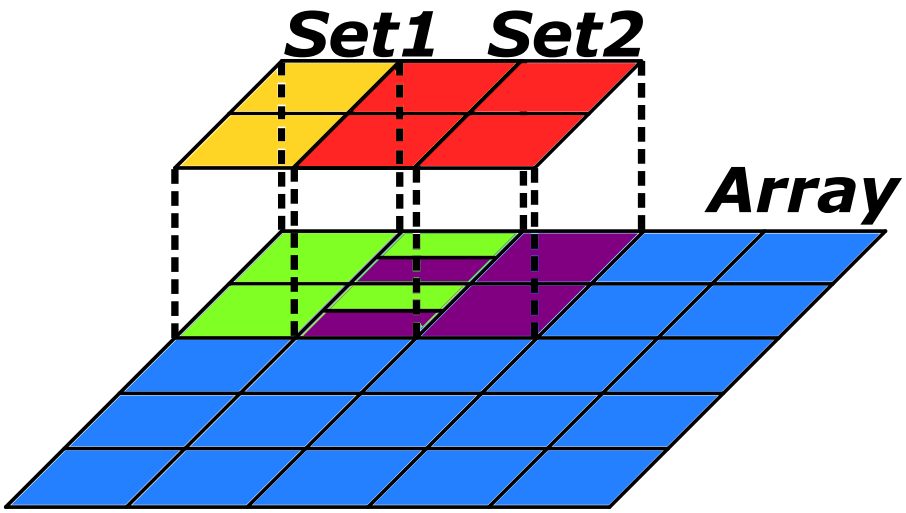

Figure 5.5. Two overlapping accumulation sets inside an array.

that has been generated. The depth of a row is strictly associated to the time frame in which the data is available in the same row. When the binder considers an accumulation set, it performs the following operations in order to properly carry out the presented optimization technique:

1. After identifying all the LiM rows that compose the initial set, the eventual ones that contribute to an already mapped elaboration are removed from the set, and the correspondent result row is inserted.

2. The operation described at point 1 is iterated until no sums among LiM rows of the set are already present in the LiM array.

3. Acting on the *reduced set* generated by the previous steps, the binder starts mapping sum operations considering LiM rows with the lowest depth, since they are the ones in which data is available first. The depth of the produced result row will be equal to the highest between the two source rows incremented by one.

4. Result rows are reinserted into the set, which is repeatedly reduced following this method until a single result LiM row remains.

In order to understand this procedure better, an example is shown and discussed in the next section.

Example of application

In Figure 5.6, two accumulation sets and the related LiM rows are shown.

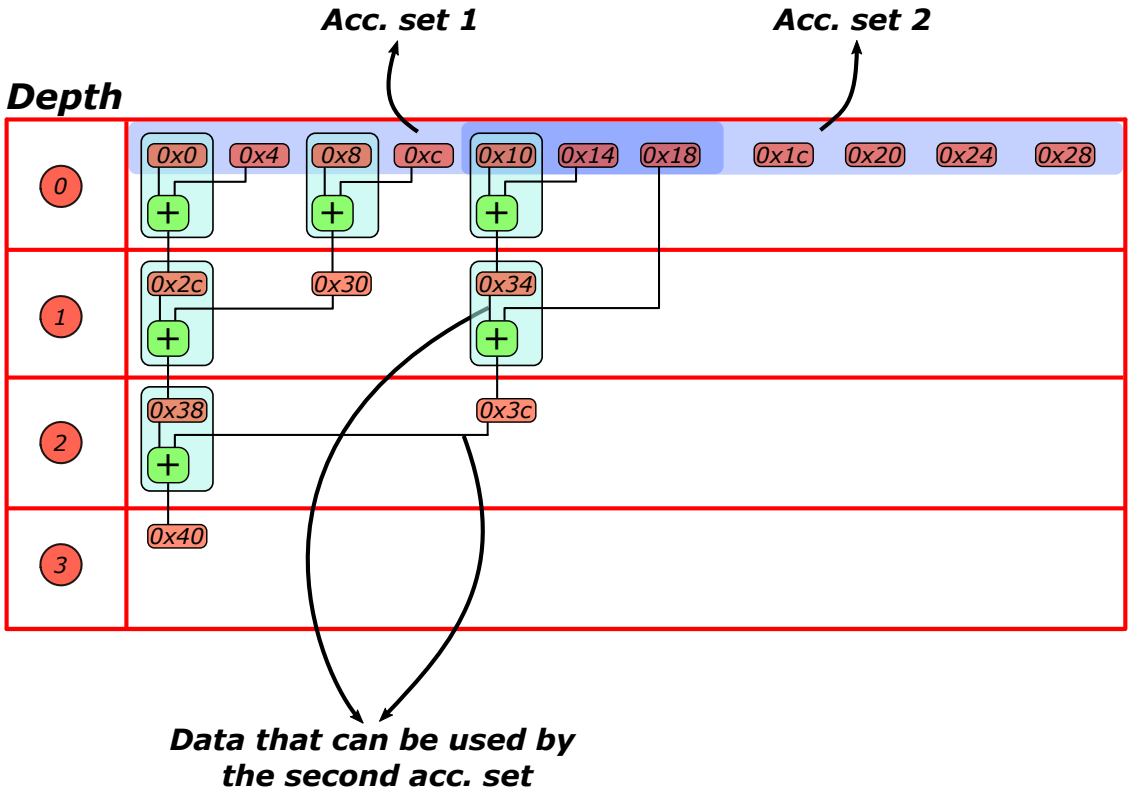

Figure 5.6. Two overlapping accumulation sets.

Rows with addresses 0x10, 0x14 and 0x18 are shared by the two sets. After the mapping of the accumulation regarding the first one occurs, LiM rows that contain intermediate results can also be exploited for the mapping of the second. Following the algorithm presented above, the reduction of the second accumulation set happens as follows:

- At the beginning, the set is composed by rows with addresses 0x10, 0x14, 0x18, 0x1c, 0x20, 0x24 and 0x28.
- The binder recognizes that rows 0x10 and 0x14 generate row 0x34. Hence, the former are deleted from the set and the latter is inserted. The optimization could also stop at this stage, thus performing the accumulation on a set formed by rows with addresses 0x34, 0x18, 0x1c, 0x20, 0x24 and 0x28. In this case, the final LiM system would not fully

benefit from the new optimization step. As a matter of fact, the binder identifies that a further reduction of the set is feasible.

- The binder also detects that rows 0x34 and 0x18 generate row 0x3*c*. Hence, it is inserted inside the set and the two initial rows are deleted. At this point, the set is reduced as much as possible and the mapping of the accumulation can start following the other steps of the algorithm described above.

In the case study discussed in Chapter 7 regarding the generation of Integral Image, multiple tests have been conducted stopping the optimization technique at different levels.

In conclusion, the presented approach ensures a **great limitation for what concerns area occupation and hardware resources**, which would grow exponentially without the introduction of these optimization strategies, especially for accumulations. In this last case, it also guarantees that the overall execution time is kept $O(log_2(N))$, which represents a great advantage with respect to a serial implementation.

Chapter 6

The expansion of the code emission phase

In the first version of Octantis, the code emission phase consisted of a single module, *PrintDexFile*, aimed at providing the configuration files for DEXiMA, useful for the simulation of the circuit. However, the *LiMArray* and *FSM* data structures, generated during the binding phase, are objects that can effectively be handled to produce different types of output descriptions of the final LiM system.

Recently, DEXiMA has been expanded with a *CAD tool*, named **DEXiMA-CAD**, that allows the availability of a visual representation of the LiM architecture under study. It needs several files to properly configure this functionality. Hence, the connection between Octantis and DEXiMA has been strengthened by means of the introduction of a new module whose purpose is the generation of these files.

Furthermore, the scope of Octantis code emission phase has been expanded in order to support the production of two **VHDL files**. The former describes the LiM system generated by Octantis synthesis process, also providing a control unit to correctly drive the execution of the algorithm. The latter consists of a *VHDL testbench* that gives the possibility to simulate the architecture using commercial tools for simulation, such as *Mentor Graphics ModelSim*. In this way, the correct behaviour of the LiM system synthesized by Octantis can be verified, too.

In the following sections, the description of the modules introduced in Octantis code emission phase to carry out the mentioned tasks is discussed.

6.1 The generation of VHDL files

The generation of the *VHDL description of the LiM system and its associated VHDL testbench* is based on the data received from the binding phase. As a matter of fact, the binder provides the LiMArray and FSM data structures that respectively define the LiM architecture and the timing information that are useful for the proper execution of the algorithm. Hence, a new module called *PrintVHDLFiles* has been introduced and it implements the translation of these data structures into a VHDL entity, which is characterized by:

- a Memory-like Interface. As a matter of fact, the developed component is a special type of memory, hence it must be compliant with the protocols usually implemented by these kinds of devices.

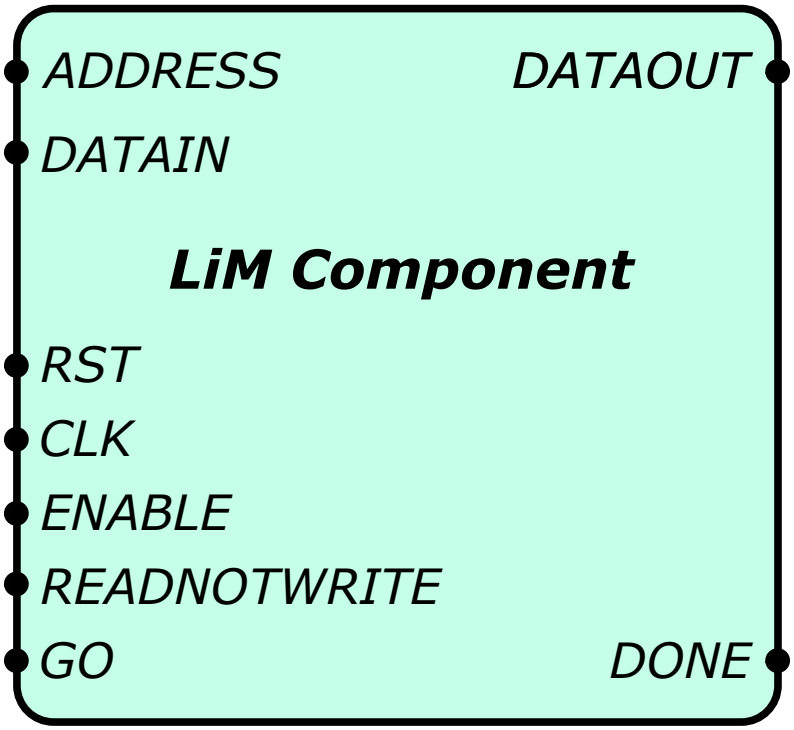

Figure 6.1. Interface of the entity generated by means of PrintVHDLFiles.

The interface is depicted in Figure 6.1. Several ports that generally characterize memories can be noticed, such as:

  - *CLK*: it is needed to provide the clock to the system.
  - *RST*: it is the asynchronous active low reset pin.
  - *ENABLE*: it allows reading and writing processes to be carried out when it is at logic '1'.
  - *ADDRESS*: it specifies the memory address to read from or write to.
  - *READNOTWRITE*: it indicates which operation must be executed. When it is at logic '1' a read operation is performed, a write operation when it is at logic '0'.
  - *DATAIN*: it is the bus where the data to be written inside the LiM row identified by ADDRESS is placed.
  - *DATAOUT*: it is the bus where data to be read from the LiM row identified by ADDRESS is provided.

  The listed ports can be exploited to issue both a **writing** and a **reading protocol**. The former is useful for the loading of the input operands inside their associated LiM rows. The writing of a values inside a given row can be issued by setting *ENABLE* to logic '1' and *READNOTWRITE* to logic '0'. A *synchronous write* of the data present on *DATAIN* inside the row pointed by *ADDRESS* is performed. The latter is exploited to retrieve the elaborated data from the LiM system. The reading of a value contained in a specific row can be started by setting *ENABLE* to logic '1' and *READNOTWRITE* to logic '1'. A *synchronous read* of the data stored in the row pointed by *ADDRESS* is performed, and it is provided on the *DATAOUT* bus. Moreover, two additional ports are available:

  - *GO*: it can be set at logic '1' when all the input operands have been loaded in order to start the execution of the algorithm in the LiM device.
  - *DONE*: it can be set at logic '1' by the system when results are available for a read operation.

- a **Control Unit** that coordinates the "movement" of data inside the LiM architecture, ensuring the correct behaviour of the system with respect

to the algorithm that has to be performed. It exploits the information obtained from the *FSM* structure in order to implement the generation of several signals called T_i. A T_i is set to logic '1' in clock cycle i. When active, a T_i signal causes the input of memory elements composing LiM cells that must be active at time i to be transferred to their output, ready for the subsequent elaborations. T_i signals are also responsible for the correct activation of *selection signals* that belong to 2-to-1 multiplexers. In conclusion, they ensure a proper timing behaviour and, as a result, the execution of the algorithm is properly carried out.

- a **Datapath** which is constructed starting from the data structure that describes the LiM architecture. LiM rows have been modeled as registers whose output is connected to the logic required for their processing. Thanks to T_i signals, registers are enabled with the right timing, thus transferring the data on their input to the output, ready for elaboration.

As regards the associated VHDL testbench, it has been designed to be compliant with the reading and writing protocols discussed above. It loads a set of input operands inside the input LiM rows, which are always the ones with the lowest address. After that, it activates the *GO* signal and waits until *DONE* is asserted. Once it detects *DONE* at logic '1', it starts reading results from correspondent rows. The addresses related to LiM rows that contain the elaborated data can be known before the generation of the testbench file. Rows whose output is not connected to other ones or integrated logic are identified and considered by the testbench as result rows.

6.2 The generation of DExIMA-CAD configuration files

The main aim of DEXiMA is to offer the possibility to characterize the developed LiM architecture with respect to area occupation and static and dynamic power consumption. However, it is now also provided with a CAD tool, named *DEXiMA-CAD*, that enables the visualization of the LiM array. It allows to check the overall composition of the system by looking at the different types of LiM rows that are present. The intra-row logic can be inspected as well, as it is the one needed to carry out instructions that require the connection between cells of the same LiM row, such as sum operations. As a matter of fact, in this case, the carry out signal has to be propagated from the cell that generates it to the next one. The tool also allows to go

deeper and analyze the internal structure of each LiM cell.

Several configuration files are required to achieve the correct representation using this tool. They are used to specify all the types of LiM cells employed in the system as well as the disposition of rows inside the LiM array. As regards Octantis, a new module called *PrintCadFiles* has been introduced in the code emission phase, with the aim of generating the required files for DEXiMA-CAD starting from the data structure that describes the LiM architecture received from the binder, which is LiMArray. In the following, the kinds of files needed by the tool are listed and their purpose examined:

- **.limcad** files are responsible for the description of LiM cells that are used inside the system. A .limcad file has to be provided for each different type of cell in the design. It provides the architectural description of the LiM cell, and its internal organization is similar to the one of a VDHL file. At the beginning, the required pins are declared along with their type (input, output) and their (x, y) coordinates, which are important for the visual representation. Most of the pins are fixed, such as:

  - CLK: clock pin.
  - RST: reset pin.
  - BL: input bit to the memory cell.
  - WL: enable pin of the memory cell.
  - OC: output of the memory cell.
  - $OLiM$: output of the LiM cell, that is the one given by its internal logic.

  Moreover, S_i pins can appear if 2-to-1 multiplexers are present inside the cell. After this section, the list of components used for the internal architecture of the LiM cell is specified along with their pins. The declaration of the memory cell is always present. Then, 2-to-1 multiplexers and logic ports can be identified, based on the operations that have to be performed. Differently from what happens for VHDL files, in this case the declaration and the port mapping of internal components are implemented simultaneously. Hence, each pin of a component is assigned with another one that belongs to another component. In this way, interconnections are also specified.

- **.irlcad** files are used to determine the intra-row logic. At the moment, the only kind of intra-row logic that can be effectively employed is the one related to sum operations. As regards the file, it has to contain the declaration of the hardware component that performs intra-row elaborations, which is represented by an adder in the case of an addition.

- **.csv** files are in charge of describing the overall structure of the LiM system. Two *.csv* files are required. The former is used in order to specify the disposition of LiM cells declared by means of *.limcad* files inside the array. The latter determines the position of the intra-row logic defined in the *.irlcad* files. The characteristics of the *.csv* format make its use suitable for the organization of this kinds of information.

The combination of these three kind of files allows DEXiMA-CAD to represent the structure of the LiM systems that has been devised by means of Octantis synthesis process.

Chapter 7

Tests

7.1 Image Processing algorithms

The enhanced capabilities of Octantis have been tested in order to verify their correctness, but also to identify their strengths along with the features that could be improved in the next expansions of the program. At the end of the synthesis process, the tool is also capable of providing information regarding both the *timing performance* and the *composition of the final LiM architecture*. Hence, they are reported for each case study that has been taken into account, and they have been also exploited in order to check the effectiveness of the target-dependent optimization introduced inside the binder and discussed in Chapter 5.

As regards the C algorithms used for testing, a specific application field has been chosen: **Image Processing**. It is a branch of *Computer Vision* that aims at the manipulation of an input digital image in order to get an enhanced version or to extract some useful information from it. Image processing algorithms are often *data-intensive* and the classic serial approach does not allow achieving great performance. The main issue is indeed represented by the overall execution time, which can become rather large for certain applications. Hence, hardware accelerators that implement parallel processing are usually exploited to carry out this kind of algorithms efficiently, such as GPUs.

Logic-in-Memory systems may represent an alternative way to address the elaboration techniques employed in the image processing field, due to their intrinsic parallel computation capabilities, as already proposed in [24].

In the next sections, three image processing algorithms are considered and the related results obtained at the end of Octantis synthesis process are provided and commented. They have been chosen as they are characterized by the presence of the C constructs, data structures and operations around which the expansion of the tool has revolved, such as loop nests and array accesses. For what concerns the operations, it has been noticed that a vast number of image processing algorithms actually exploit *accumulations*. These can be used either to generate intermediate data structures required by other elaboration stages or to modify the initial image with the implementation of spatial filters. Furthermore, also *bit-wise logic operators* can be effectively used to extract specific regions of interest by means of masks. They are also present in the image encryption domain, where *XOR* and *XNOR* operators are mainly employed.

7.1.1 Synthesis of the Integral Image algorithm

The *Integral Image* algorithm aims at the generation of a data structure that takes its name from the algorithm itself, even though it can also be referred to as *Summed Area Table*. It consists in a preprocessing element which has become quite well-known as it represented a key point in the object detection framework [25] proposed in 2001 by Paul Viola and Michael Jones. The Summed Area Table is defined as an image in which each pixel $P_{IM}(x_i, y_i)$ corresponds to the sum of all the pixels above and to the left of its equivalent pixel $P_{input}(x_i, y_i)$ in the initial image.

As one can easily imagine, this algorithm has been chosen as it allows to test the new features that aim at the handling of loop nests and array accesses. More importantly, the presence of *multiple overlapping accumulation sets* also activates the optimization strategy introduced in the binding phase and examined in Chapter 5.

Integral Image generation algorithm

```
//Image sizes
#define R 16
#define C 16

void sat(){
    //Input image
  int Image[R][C];
  //Integral Image
  int SAT_Image[R][C];
  //Temporary variable for storing the result of the
   accumulation
  int S;

    //Cycling over input image pixels
  for(int i = 0; i < R; ++i){
    for(int j = 0; j < C; ++j){
        //Initializing temp variable for accumulation
   result
      S = 0;

      //Cycling over all pixels above and left to the
   current one
      for(int k = 0; k <= i; ++k){
        for(int l = 0; l <= j; ++l){
            //Accumulating pixels
          S += Image[k][l];

        }
      }
      //Assigning temp variable to output Integral Image
   proper pixel
      SAT_Image[i][j] = S;

    }
  }
}
```

Listing 7.1. C implementation of the algorithm that generates the *Integral Image*

The C code that implements the algorithm for the generation of the Integral Image has been defined and it is shown in Listing 7.1. A 16x16 grey-scale input image has been selected for the purpose of this test, and it is represented in the C code by means of the RxC matrix named $Image$. Being each pixel value represented on 8 bits, a parallelism of 16 bits has been defined as parameter in the Octantis configuration file. The image obtained after the execution of the algorithm has the same size of the input one and it is identified as SAT_Image in the code.

The C implementation provided in Listing 7.1 has been given to Octantis along with the configuration file. Results regarding the LiM architecture and timing information have been obtained and they are shown in Table 7.1 and Table 7.2. As it can be noticed, various tests have been conducted with different *optimization levels*. The number associated with this parameter indicates the maximum depth of the LiM rows that contain intermediate results taken into account by the optimization technique discussed in Chapter 5. As already mentioned, the depth of each row refers to the time frame in which the data is available for calculation in that same row. As a result, the depth of the LiM rows that store the initial values of the accumulation set is smaller than the one associated with the row that contains the final result. Hence, higher the optimization level, "deeper" the LiM rows considered to be containing an already available result by the new optimization step.

Opt. Level	*Tot. Mem. Rows*	*LiM Rows*	*Non-LiM Rows*	*Density of LiM Rows*
0	27457	18240	9217	66,4%
1	13811	9216	4595	66,7%
2	7096	4800	2296	67,6%
3	3866	2704	1162	70,0%
4	2406	1792	614	74,5%
5	1820	1436	384	78,9%
6	1643	1328	315	80,8%
7	1621	1313	308	81,0%

Table 7.1. Results regarding the types of rows present in the LiM system that implements the generation of the *Integral Image*. Data are provided for each optimization level.

As it can be seen in Table 7.2, along with the increment of the optimization level, the memory dimension and the amount of full-adders and half-adders drastically reduce, while the overall execution time remains the same, equal to $log_2(RC)$.

Opt. Level	*Mem. Dimension*	*Integrated Logic*	*Exec. time*
0	439312 bits	Full/Half-Adder: 291840	8 T_{clk}
1	220976 bits	Full/Half-Adder: 147456	8 T_{clk}
2	113536 bits	Full/Half-Adder: 76800	8 T_{clk}
3	61856 bits	Full/Half-Adder: 43264	8 T_{clk}
4	38496 bits	Full/Half-Adder: 28672	8 T_{clk}
5	29120 bits	Full/Half-Adder: 22976	8 T_{clk}
6	26288 bits	Full/Half-Adder: 21248	8 T_{clk}
7	25936 bits	Full/Half-Adder: 21008	8 T_{clk}

Table 7.2. Results obtained from the synthesis of the algorithm for the generation of the Integral Image. Along with the optimization level, different data are provided.

Moreover, an additional benefit derived from the introduction of the new target-dependent optimization is shown in Table 7.1. Here, the number of rows provided with internal logic, referred to as *LiM rows*, and the amount of normal memory rows, identified as *Non-LiM*, is reported. One of the main aims of Octantis is to generate a LiM architecture in which the main characteristics of the LiM paradigm are fully exploited. Hence, the *use of normal memory rows must be minimized as much as possible*, with the purpose of creating a compact structure in which the needed operations are carried out using as few rows as possible. Table 7.1 shows how the percentage of LiM rows grows with the increment of the optimization level.

7.1.2 Synthesis of a multi-image encryption algorithm

The second test case concerns the *multi-image encryption algorithm* proposed in [26]. The main objective of this research work consists in the definition of a *encryption scheme*, along with the relative *decryption* one, ensuring the secure transmission of multiple images.

The main phases of the defined encryption procedure are:

1. Multiple NxN images are manipulated and by means of the *Linear Wavelet Transform* operation, thus obtaining the so-called *sparse images*.

2. Sparse images undergo a further procedure named *scrambling*, which

consists in randomly rearranging pixels in order to break the correlation between neighbouring one, thus making the image visually unreadable. **Scrambled images** are thus obtained.

3. The *XOR* operator is applied on the scrambled images to generate the **XOR-Image**. Equation 7.1 better clarifies this operation, where IMG_i refers to the i-th scrambled image.

$$XOR\_Image = IMG_0 \oplus IMG_1 \oplus ... \oplus IMG_{n-1} \oplus IMG_n \tag{7.1}$$

Meanwhile, **XOR-Keys** must be generated for each scrambled image, in order to be used in the decryption process. The *XOR-Key* for the i-th image can be retrieved by means of the application of the XOR on all the scrambled images, except the i-th itself, as shown in equation 7.2.

$$XOR\_Key_i = IMG_0 \oplus IMG_1 \oplus ... \oplus IMG_{i-1} \oplus IMG_{i+1} \oplus ... \oplus IMG_{n-1} \oplus IMG_n \tag{7.2}$$

A possible C implementation of the operations required to obtain both the *XOR-Image* and the *XOR-Keys* is reported in Listing 7.2. The presented algorithm has been given in input to Octantis along with the configuration file, where the word length parameter has been set to 8 bits.

Differently from the tests conducted in [26], six 16x16 8-bit grey-scale images have been used for the scope of this test. Since the computation of each result image in the proposed algorithm is not dependent on the others, all calculations can be performed in parallel, taking great advantage from a LiM implementation. Moreover, this example allows to highlight the introduced synthesis optimizations aimed at reducing the number of LiM rows to be inserted in the array in the case of fully-parallel operations.

As it can be observed in Table 7.3, the minimum amount of LiM rows needed to properly carry out the algorithm is reached also by means of the introduction of 2-to-1 multiplexers inside Xor LiM cells. All the optimization measures adopted led to the optimal solution for what concerns the saving of space occupation. The overall results related to memory dimensions and integrated logic have been retrieved and they are reported in Table 7.3. Finally, it can be noticed how the overall execution time is rather small, being

equal to only $5T_{clk}$, which is the key advantage of the parallel computation enabled by a LiM architecture.

XOR-Image and XOR-Keys generation algorithm

```
//Image sizes
#define R 16
#define C 16

void XorImage_XorKey_Generation(){

  //Scrambled images
  int Image_0[R][C], Image_1[R][C], Image_2[R][C], Image_3
   [R][C], Image_4[R][C], Image_5[R][C];

  //Xor-Keys to be used in decryption
  int Key_0[R][C], Key_1[R][C], Key_2[R][C], Key_3[R][C],
   Key_4[R][C], Key_5[R][C];

  //Xor Image
  int Xor_Image[R][C];

  //Xor Image and Xor Keys generation
  for(int i = 0; i < R; ++i){

    for(int j = 0; j < C; ++j){

      //Performing Xor operation on all scrambled images
   to get the Xor Image
      Xor_Image[i][j] = Image_0[i][j] ^ Image_1[i][j] ^
   Image_2[i][j] ^ Image_3[i][j] ^ Image_4[i][j] ^ Image_5
   [i][j];

      //Xor-Key 0 generation
      Key_0[i][j] = Image_1[i][j] ^ Image_2[i][j] ^
   Image_3[i][j] ^ Image_4[i][j] ^ Image_5[i][j];

      //Xor-Key 1 generation
      Key_1[i][j] = Image_0[i][j] ^ Image_2[i][j] ^
   Image_3[i][j] ^ Image_4[i][j] ^ Image_5[i][j];

      //Xor-Key 2 generation
      Key_2[i][j] = Image_0[i][j] ^ Image_1[i][j] ^
   Image_3[i][j] ^ Image_4[i][j] ^ Image_5[i][j];
```

```
        //Xor-Key 3 generation
        Key_3[i][j] = Image_0[i][j] ^ Image_1[i][j] ^
    Image_2[i][j] ^ Image_4[i][j] ^ Image_5[i][j];

          //Xor-Key 4 generation
        Key_4[i][j] = Image_0[i][j] ^ Image_1[i][j] ^
    Image_2[i][j] ^ Image_3[i][j] ^ Image_5[i][j];

        //Xor-Key 5 generation
        Key_5[i][j] = Image_0[i][j] ^ Image_1[i][j] ^
    Image_2[i][j] ^ Image_3[i][j] ^ Image_4[i][j];

     }

   }

}
```

Listing 7.2. C code for Xor Image and Xor-Key generation

Xor-Image - Xor-Key generation algorithm			
Memory Row Type	***Number of Memory Rows***	***Memory Dimension***	***Integrated Logic***
Simple memory	2560	20480 bits	None
Xor	3056	24448 bits	Xor: 24448
Xor with 2-to-1 mux	1024	8192 bits	Xor: 8192 Mux 2-to-1: 8192

Table 7.3. Aggregated results of the synthesis by Octantis of the algorithm for the generation of both the XOR-Image and the XOR-Keys.

7.1.3 Synthesis of an approximated Arithmetic Mean Filter

In Image Processing, the class referred to as *Spatial Filters* encompasses the set of techniques that operate on an image by taking into account, for each pixel, the intensity values of other pixels in a well-defined nearby area. Starting from the initial image, the general method adopted by a spatial filtering algorithm is implemented as follows:

- A *filter mask* is chosen. It is a NxN matrix whose size and values strictly depends on the type of filtering required.
- A pixel of the input image in position (i, j) is selected.
- The center value of the filter mask is "moved" in order to coincide with the chosen pixel. All the input image pixels included in a NxN area around it are taken into account, and they are elaborated by means of the coefficients of the filter mask. A single intensity values is obtained, which will be assigned to the pixel of the "filtered" image in position (i, j).

As regards the **Arithmetic Mean Filter** (AMF), it implements the arithmetic mean among the intensities of all pixels in the NxN area around the selected one. A possible mathematical formulation of this operation is provided in Equation 7.3, where $P_{input}(k, l)$ refers to the pixel of the input image in position (k, l).

$$P_{filtered}(i, j) = \frac{1}{N^2} \sum_{k=i-\frac{N}{2}}^{i+\frac{N}{2}} \sum_{l=j-\frac{N}{2}}^{j+\frac{N}{2}} P_{input}(k, l) \tag{7.3}$$

This kind of filter is usually employed with the purpose of removing short-tailed noise, such as uniform and Gaussian type noise, from the initial image at the cost of blurring it. As a result, the input image is "smoothed", and a more pronounced effect can be obtained by enlarging the size of the filter mask.

The application of an AMF filter to a 16x16 grey-scale image (8-bit pixels) has been considered for the synthesis on LiM with Octantis, and the related C implementation is provided in Listing 7.3. As a matter of fact, it is perfectly suitable for the testing of all the introduced novelties and optimizations. The access pattern employed for the visit of the array *Image*

allows the identification of multiple overlapping accumulation sets, whose size is given by $(N+1)$x$(N+1)$. However, the application of the filter on a LiM system cannot be strictly compliant with its definition, as divisions are not feasible. On the other hand, right-shifts can replace divisions and, if the second operand of this operation is equal or almost equal to a power of 2, an approximation can be successfully implemented.

Algorithm for the application of the approximated *AMF*

```
#define N 2
#define R 16
#define C 16

void amf(){
    //Input Image
  int Image[R][C];
  //Final Filtered Image
  int Filtered_Image[R-2][C-2];
  //temp variable
  int S;


  for(int i = N/2; i < R - N/2; ++i){
    for(int j = N/2; j < C - N/2; ++j){
      S = 0; //Initializing the tmp variable
      for(int k = -N/2; k <= N/2; ++k){
        for(int l = -N/2; l <= N/2; ++l){
          S += Image[i + k][j + l]; //Accumulation on a
   set
        }
      }
      Filtered_Image[i-1][j-1] = S/(N+1)(N+1); //
   Arithmetic Mean
    }
  }
}
```

Listing 7.3. C code for a (N+1)x(N+1) arithmetic mean filter applied on a RxC input image

In the algorithm showed in Listing 7.3, the area of the filter mask is equal to 9 pixels, hence the final division should consider this value. As regards the mapping on LiM, Octantis allocates rows of type *rightShift*, which are capable of right shifting, at each clock cycle, the bits stored in their cells by

means of intra-cell connections.

The results obtained from the synthesis of the algorithm in Listing 7.3 are shown in Table 7.4 and 7.5. The organization of data in the mentioned Tables is similar to the one used in the section about the synthesis of the Integral Image generation algorithm. In this case, only two optimization levels are considered. As a matter of fact, the maximum depth of a LiM row for an accumulation set of 9 is equal to $\lfloor log_2(9) \rfloor + 1 = 4$, with the value 4 being assigned to the result rows. With the optimization level parameter set to 3, no further improvements have been found, hence it has not been inserted in Tables. As already noticed in the synthesis of the Integral Image generation algorithm, a quite remarkable reduction in area occupation is obtained by means of the new optimization technique. The final division costs three additional clock cycles after the accumulation in order to perform the needed right shift. As a result, 9 T_{clk} are required for the overall execution of the algorithm.

Opt. Level	*Tot. Mem. Rows*	*Add Rows*	*rightShift Rows*	*Non-LiM Rows*	*Density of LiM Rows*
0	2550	1568	196	787	69,2%
1	2108	1337	196	576	72,7%
2	2101	1331	196	575	72,7%

Table 7.4. Results concerning the types of rows inside the LiM system that implements the application of the approximated Arithmetic Mean Filter. Different values are provided along with the optimization level.

Opt. Level	*Mem. Dimension*	*Integrated Logic*	*Exec. time*
0	20400 bits	Full/Half-Adder: 12544 cells connected for shift: 1568	9 T_{clk}
1	16864 bits	Full/Half-Adder: 10696 cells connected for shift: 1568	9 T_{clk}
2	16808 bits	Full/Half-Adder: 10648 cells connected for shift: 1568	9 T_{clk}

Table 7.5. Overall results obtained from the synthesis of the algorithm implements the application of the approximated Arithmetic Mean Filter. Along with the optimization level, different data are provided.

Chapter 8

Conclusions and future works

The introduction of InfoCollector and the subsequent evolution of the back-end phases have surely led to a *more mature version of Octantis*. The new Pass and the other C++ classes that have been presented, such as *PointerInfoTable* and *LoopInfoTable*, provide data structures and analysis techniques that can be employed for the management of information regarding:

- **multiple *n*-deep loop nests**, where n is an integer number;
- **one/two-dimensional arrays** are accessed by means of well-defined **array access patterns**;

The data gathered have been exploited towards the creation of *Access Pattern Matrices*, which have been considered as useful mathematical objects capable of "compressing" the complex information about array access patterns inside 1xN or 2xN matrices. Hence, it has represented the connection point between InfoCollector and the binder, which has been restructured in order to properly analyze APMs and exploit them for mapping and parallelizing operations on the final LiM system.

Moreover, InfoCollector has also aimed at the speeding up of the scheduling phase by means of *Alias Analysis* and the identification of *valid basic blocks*. These tasks are crucial for reducing the time spent on scheduling and they could also provide important advantages as regards future expansions, where Octantis could be able to handle more than one function and sophisticated code constructs.

As also highlighted in Chapter 7, the benefits derived from the introduction of the new optimization strategy are remarkable and they could become even larger with the synthesis of more complex architectures. As a matter of fact, even though Octantis is a HLS tool whose main aim is the "translation" of a C algorithm toward its equivalent LiM implementation, it must also devise suitable target-dependent techniques in order to allow the optimization of the system with respect to relevant figures of merit, such as timing and area occupation.

Several advanced features have been developed but room of improvement is still present. The main reason why the management of loops and array accesses has been undertaken lies in the great benefits that they could draw from the parallel processing on LiM devices. However, it also represents the first step that has been made in a well-defined direction: *the synthesis of benchmarks*. Several C codes used for the testing of HLS tools have been gathered through the years, resulting in the generation of few libraries, such as *MachSuite*[27] or *Polybench/C*[28]. They respectively consist of a set of 19 benchmarks suitable for hardware acceleration and a collection of benchmarks containing static control parts. However, the majority of the C programs here contained highly employs multiplications and divisions, which are well-known for their resource consuming implementations. Hence, future works could consider mapping operations on a more complex system where a LiM device and a conventional processing unit can coexist. The former would allow the parallel execution of bitwise, sum and subtraction instructions while providing the latter with the data needed for more sophisticated instructions. Octantis would have to devise several strategies aimed at the allocation of operation to one of the two units, whose connections must be designed with the purpose of reducing power consumption as much as possible and ensure good performance.

In conclusion, the hope is to have contributed to the evolution of the Octantis project, which will become one of the key tool within the framework that the *VLSI Laboratory* is developing for the design and characterization of LiM systems.

Bibliography

[1] Giulia Santoro, Giovanna Turvani, and Mariagrazia Graziano. *New Logic-In-Memory Paradigms: An Architectural and Technological Perspective*. *Micromachines*, 10(6):368, May 2019. doi:10.3390/mi10060368.

[2] Nicola Piano. *DExIMA: a Design Explorer for In-Memory Architectures*, 2019. URL: https://webthesis.biblio.polito.it/12547/.

[3] Fabrizio Riente, Izhar Hussain, Massimo Ruo Roch, and Marco Vacca. *Understanding CMOS Technology Through TAMTAMS Web*. *IEEE Transactions on Emerging Topics in Computing*, 4(3):392–403, Jul 2016. doi:10.1109/tetc.2015.2488899.

[4] Shyamkumar Thoziyoor et al. Palo Alto: HP Laboratories, 2008. URL: https://www.hpl.hp.com/techreports/2008/HPL-2008-20.html.

[5] Andrea Marchesin. *Octantis - A High-Level Explorer for Logic-in-Memory architectures*, 2020. URL: https://webthesis.biblio.polito.it/15852/.

[6] Andrea Marchesin, Giovanna Turvani, Andrea Coluccio, Fabrizio Riente, Marco Vacca, Massimo Ruo Roch, Mariagrazia Graziano, and Maurizio Zamboni. *Octantis: An Exploration Tool for Beyond von Neumann architectures*, Jun 2021. doi:10.1109/dtis53253.2021.9505135.

[7] Greg Wilson et al. Amy Brown. *The Architecture of Open Source Applications: elegance, evolution, and a few fearless hacks*. 2011.

[8] LLVM Developer Group. Clang: a c language family frontend for llvm. URL: https://clang.llvm.org/index.html.

[9] LLVM Developer Group. *LLVM's Analysis and Transform Passes*. URL: https://llvm.org/docs/Passes.html.

[10] Bruno Cardoso Lopes and Rafael Auler. *Getting Started with LLVM Core Libraries*. Packt Publishing, 2014.

[11] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. *The Organization of Computations for Uniform Recurrence Equations*. *Journal*

of the ACM, 14(3):563–590, Jul 1967. doi:10.1145/321406.321418.

[12] P. Feautrier. *Array expansion. Proceedings of the 2nd international conference on Supercomputing - ICS '88*, 1988. doi:10.1145/55364.55406.

[13] Leslie Lamport. *The parallel execution of DO loops. Communications of the ACM*, 17:83–93, Feb 1974. doi:10.1145/360827.360844.

[14] Sanjay V. Rajopadhye. *Synthesizing systolic arrays with control signals from recurrence equations. Distributed Computing*, 3(2):88–105, Jun 1989. doi:10.1007/bf01558666.

[15] Paul Feautrier. *Parametric integer programming. RAIRO - Operations Research*, 22(3):243–268, 1988. doi:10.1051/ro/1988220302431.

[16] V. Loechner. *PolyLib: A Library for Manipulating Parameterized Polyhedra*. URL: http://www.irisa.fr/polylib/.

[17] *Polyhedral.info, The Polyhedral Compilation Community.* URL: https://polyhedral.info/.

[18] The Polly Team. *Polly Documentation.* URL: http://polly.llvm.org/docs/.

[19] Fabien Quilleré and Sanjay Rajopadhye. *Optimizing memory usage in the polyhedral model*, Sep 2000. doi:10.1145/365151.365152.

[20] Wei Zuo, Peng Li, Deming Chen, Louis-Noel Pouchet, Shunan Zhong, and Jason Cong. *Improving polyhedral code generation for high-level synthesis*, Sep 2013. doi:10.1109/codes-isss.2013.6659002.

[21] Nam Khanh Pham, Amit Kumar Singh, Akash Kumar, and Mi Mi Aung Khin. *Exploiting Loop-Array Dependencies to Accelerate the Design Space Exploration with High Level Synthesis. Design, Automation Test in Europe Conference Exhibition (DATE), 2015*, 2015. doi:10.7873/date.2015.0199.

[22] LLVM Developer Group. *LLVM Loop Terminology (and Canonical Forms).* URL: https://llvm.org/docs/LoopTerminology.html.

[23] Wei Zuo, Yun Liang, Peng Li, Kyle Rupnow, Deming Chen, and Jason Cong. *Improving high level synthesis optimization opportunity through polyhedral transformations. Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays - FPGA '13*, 2013. doi:10.1145/2435264.2435271.

[24] Mario Cofano, Marco Vacca, Giulia Santoro, Giovanni Causapruno, Giovanna Turvani, and Mariagrazia Graziano. *Exploiting the Logic-In-Memory paradigm for speeding-up data-intensive algorithms. Integration*, 66:153–163, May 2019. doi:10.1016/j.vlsi.2019.02.007.

[25] P. Viola and M. Jones. *Rapid object detection using a boosted cascade*

of simple features. Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001. doi:10.1109/cvpr.2001.990517.

[26] Xianye Li, Xiangfeng Meng, Xiulun Yang, Yurong Wang, Yongkai Yin, Xiang Peng, Wenqi He, Guoyan Dong, and Hongyi Chen. *Multiple-image encryption via lifting wavelet transform and XOR operation based on compressive ghost imaging scheme. Optics and Lasers in Engineering*, 102:106–111, Mar 2018. doi:10.1016/j.optlaseng.2017.10.023.

[27] *MachSuite, Benchmarks for Accelerator Design and Customized Architectures.* URL: https://breagen.github.io/MachSuite/.

[28] *PolyBench/C, the Polyhedral Benchmark suite.* URL: https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.