POLITECNICO DI TORINO

Master degree course in Electronic Engineering

Master Degree Thesis

Octantis

A High-Level Explorer for Logic-in-Memory architectures

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

Candidate
Andrea Marchesin
ID: 262776

Academic year 2019 – 2020

To the people I love.

Summary

Today, one of the problems the scientific community is called upon to tackle is the well-known *von Neumann bottleneck*. Among the various solutions under study, in the recent years the VLSI Laboratory of Politecnico di Torino has proposed the concept of *Logic-in-Memory (LiM)*: a memory device which embeds simple computational elements between the different cells, overall arranging a distributed processing system. The key idea consists in reducing access to the memory for the CPU, implementing a precomputation of raw data within the same memory. The CPU is left only with the task of performing the more complex operations on that pre-processed information.

Different architectures have been proposed during the years and since 2019 the research group decided to go beyond the realization of specific case of study. In this context, *DExIMA* was born as a "beta version" of a software tool able to characterize generic Logic-in-Memory architectures. The information which the program can provide refers to *space occupation*, *maximum performance* and *static* and *dynamic power consumption*.

The thesis work has been profoundly influenced by the discussed background and it introduces *Octantis*, a simple *High-Level Synthesis* tool, inspired by other modern ones considered for FPGAs applications. It has been designed from scratch and it reveals useful for the exploration of LiM architectures. The purpose of the program is the analysis of an input algorithm, described in Standard-C language, to provide an effective Logic-in-Memory unit for its implementation. In order to best accomplish this job, Octantis receives as input also a configuration file, whose information guides the synthesis process toward the definition of more accurate solutions. At the output of the program the DExIMA configuration files are provided, so that the simulator engine can perform complete analysis of the defined architecture.

The open-source *LLVM compiler infrastructure* has been considered to shape

the whole program and thus to carry out the process of translation. The associated libraries are useful to optimize and generate a code for a target architecture starting from an input source. The project structure, strongly hierarchical, is constituted by different modules well integrated inside this framework. Firstly, the input algorithm, enriched with the information expressed inside the configuration file, is decomposed in a scheme that best express the logic which ties the various operations to be performed. Then, optimizations techniques are applied and, step-by-step, the Logic-in-Memory unit takes form, both in its architecture and its control. At the end, the results of the compilation process are written inside a file, compliant with the DExIMA's syntax and ready to be simulated by the latter.

The strengths of the developed project are many. First of all, the *speed* with which the solutions are synthesized and proposed to the user. Then, the *modularity* principle to which its structure is inspired, for guaranteeing a good *maintenance* of the code. In this regard, a detailed documentation is provided together with Octantis. Moreover, the capability of generating both the description of a *hardware LiM architecture* and its *control flow* starting from simple input algorithms.
Through the use of an input configuration file, the concepts of *configuration* and *customization* have been adopted too.

Encouraging results have been obtained during the conducted tests, where some of the most recent Logic-in-Memory research proposals have been successfully synthesized.

Starting with this first release of Octantis, many additional and more advanced features can be brought to the project. Chief among the various improvements, a more *intimate* configuration capability to allow each electronic engineer the exploration of specific Logic-in-Memory devices for the application of an input algorithm. Another future work goal could be the identification of the sections, within a generic input code, suitable for a Logic-in-Memory implementation, distinguishing them from those that are not. In this way, part of the instructions would be executed inside the LiM array, the others by means of an external processing element. Therefore, information about the behavior of a more complex system, composed of the association of a LiM unit and a CPU, would be gathered, highlighting the possible benefits.

Contents

Introduction

The Octantis project proposes a High-Level Synthesis tool for the exploration of *Logic-in-Memory* architectural solutions. It is intended to work in pairs with another tool developed inside the VLSI Laboratory of Politecnico di Torino, a Logic-in-Memory simulator named DExIMA.

The present dissertation in composed by the union of two parts. The former which offers an introduction on the project background and on the reasons behind the development of Octantis. The latter which *dives* in its implementation details, unfolding a structural analysis before presenting the results of the test conducted on it.

In particular, the pages of the first part of the document begin by retracing the research history of Logic-in-Memory devices, to which also DExIMA belongs. Logic-in-Memory systems proved to be effective in the execution of parallel tasks. Their architecture, strongly regular and hierarchical, makes them particularly suitable to benefit of High-Level Synthesis.

To date, parallel computing represents a particularly interesting solution for dealing with the crisis of Moore's law and so the performance *stagnation*. Parallel general purpose architectures need parallel input codes to make the most of their capabilities. Therefore, numerous techniques to define parallel algorithms are discussed together with the major available APIs and libraries to adapt, to parallel processing, high level languages (e.g. C, C++ and Fortran), originally meant for a serial execution. These arguments reveal useful not only in the definition of the input C-code for Octantis, but also for the adoption of automatic parallelization strategies inside the same compiler.

During the last decades, *High-Level Synthesis* tools are viewed with great interest to speed-up the design process and along the verification phase. Attention is then given also to the related researches. In fact, Industry makes

currently a wise use of them for certain applications and, in particular, for the development of the highly optimized IP cores. Moreover, many studies have indicated the Electronic System-Level Design, to which High-Level Synthesis represents an integral part, as the near future of electronic design. As High-Level Synthesizers represent, for all intents and purposes, compilers, a description of the general structure of these kind of software programs is given too.

Due to these reasons, the investment in the present research project has been decided.

For what concerns, instead, the second part of the thesis work, it is focused on the characterization of Octantis' structure and its capabilities. Enrich the discussion a brief digression about the *LLVM framework*, within which the project has been defined, together with the presentation of the results of the operated tests.

Part I

The Logic-in-Memory concept and today's need for parallel computing

Chapter 1

Motivation and background

1.1 An introduction to Logic-in-Memory systems

During the last decades, electronic devices have undergone a significant increase in performance thanks to technological advances in the industry. Moore's law represented the landmark of that progress and the continuous scaling of the technological node led to numerous benefits. At the same time, many problems have appeared and designers had to do their best to compensate them and to ensure the benefits would prevail.

Inside a traditional electronic processing system we can identify two main components: the *CPU* and the *memory*, where the former elaborates the data stored within the latter. Architectures thus described are defined as Von Newmann. The two elements continuously exchange among themselves information to implement a specific algorithm. However, this communication has its costs in terms of both power consumption and delay. Moreover, while processing units could take advantage of CMOS scaling in order to improve their performances, the memories couldn't, since they are based on completely different technologies. For this reason, nowadays memories are not able to reach the same performance of the CPU and many strategies have to be adopted to fill the gap, like implementing a hierarchy of memory and adopting statistical approaches to manage data.

In particular for data intensive applications, the presented problems become so relevant that a real bottleneck is created inside the systems. To Industry members, this particularly disadvantageous situation is know as *"Memory Wall"*, or in a more intuitive way, *"Von Neumann bottleneck"*.

Among the numerous proposals for resolving these critical issues, *Logic-in-Memory (LiM)* architectures have been introduced. The latter achieve a complete integration between memory and processing unit, radically overcoming the *wall*. The key concept behind Logic-in-Memory architectures consists in considering a memory made of cells capable to perform local computation of the data stored inside them. In this way, information doesn't need to be transferred outside the storage device for the elaboration, significantly reducing the associated costs. Logic operations are performed within the same memory array while arithmetic ones are demanded to more complex peripheral circuits.

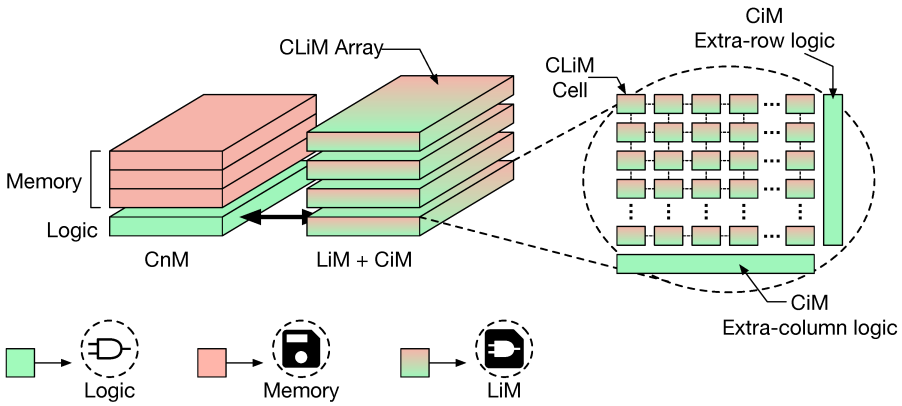

Figure 1.1: Basic scheme of Configurable Logic-in-Memory Architecture (CnM: Computation near Memory, CiM: Computation in Memory). *Courtesy of Giulia Santoro from Article [1]; p. 5, Figure 2.*

At *VLSI Laboratory* of Politecnico di Torino, an important research work has been carried out during the last years to investigate the possibilities of these new computational units. Some examples of the derived fruits considers CLiMA, a *Configurable Logic-in-Memory Array* developed in the same research team, for the implementation of a *Quantized Convolutional Neural Networks*[1] and an accelerator for the *Bitmap Indexing algorithm*[2], both data intensive applications. CLiMA, whose scheme of principle is depicted in Figure 1.1, is composed of memory cells provided by different logic ports (i.e. and, or, xor) and a full-adder, all connected by configurable interconnections,

so that a designer can easily implement a wide variety of algorithms. Promising results have been achieved in terms of performance and power consumption. Results that become even more important considering *beyond-CMOS* technological solutions in the definition of these architectures.

In order to simplify the researches in this field, in 2019 a Logic-in-Memory systems simulator has been introduced, *DExIMA*[3] in its first version. In fact, there was the need to speed up the test phase, filling the gap of the modern EDA tools which are not currently capable of handling these devices.

1.2 DExIMA: a simulator for LiM systems

The name of the tool is an acronym for *Design Explorer for In Memory Applications* and, as it can be seen by its name, it is a support software for the design of Logic-in-Memory architectures. It has been fully developed in C++ and it allows to characterize the behavior of those systems in terms of *space occupation*, *performance* and, above all, *static* and *dynamic power consumption*.

The systems that DExIMA can simulate are complex and composed of a Logic-in-Memory unit connected through a bus to an external CMOS circuit. The idea consists in considering the traditional logic to perform more sophisticated calculations on the data and the Logic-in-Memory architecture to implement simpler ones but capable to exploit its *parallel nature*.

The description of the whole circuit, via a properly formatted configuration file, occurs at an abstraction level similar to the RTL[1] one. In particular, the Logic-in-Memory component can be defined as a set of interconnected cells, each of which characterized by a one-bit storage unit associated to few logic elements. The flexibility of this design approach is fundamental for the aims of the same program, i.e. the exploration of the capabilities of Logic-in-Memory systems.

[1]The *Register Transfer Level* is a way to describe the behavior of an electronic circuit through signals, memory elements and logic operators.

1.2.1 The models library

The software can rely on a wide library of components constituted by both traditional logic and Logic-in-Memory cells and suitably described in terms of functionality, geometry, internal delay and capacitive load. The models adopted to define the logic, consider the modular approach and the information derived from *TAMTAMS*[4], a web-based framework useful to define the behavior of a CMOS circuit starting from its characteristic parameters (e.g. device currents, circuit delay and interconnects noise) and developed within Politecnico di Torino too. Regarding the description of the memory cells, many proposals are currently under investigation. A first approach, still implemented at the state of the art, considers the parameters extracted by a open-source simulator of memories developed by Hewlett-Packard Laboratories, *CACTI*[5]. However, the derived information has turned out to be inappropriate for a precise characterization of Logic-in-Memory cells and today a new path is being sought to improve it. The studied solutions are not only focused on traditional memory architectures but researchers are trying to look beyond on more advanced technologies. Researches from the VLSI Laboratory are investigating *spintronic devices* and, in particular, NanoMagnet Logic and perpendicular NanoMagnet Logic. Interesting results from these studies can be deepened through articles [6] and [7].

In conclusion of the description of DExIMA's library and concentrating on the functional aspects of the implemented models, the available RTL-like cells a designer can consider during the study of a Logic-in-Memory architecture are depicted in Figure 1.2.

1.2.2 The description of the Logic-in-Memory unit

The Logic-in-Memory cells have to be distributed on a rectangular array of dimensions $R \times C$, where R and C are respectively the numbers of rows and columns of the memory device. Additional logic blocks can be inserted between adjacent cells in order to perform more complex inter-row and inter-column operations. When all the elements are disposed in the desired positions, the interconnections between them need to be defined. To do this, the designer has to simply declare the extremes of the different links (i.e. output port and input port of the cells involved in the connection) and DExIMA itself will take care to arrange the specific path following a Manhattan routing technique.

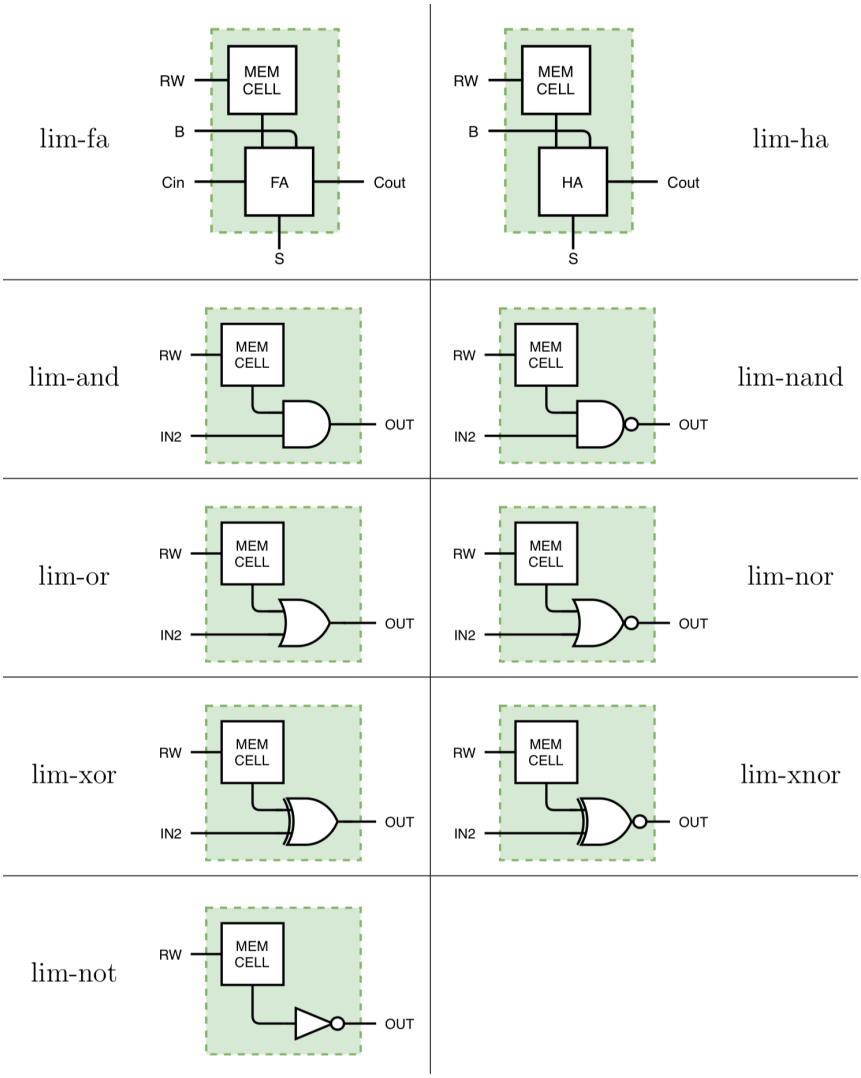

Figure 1.2: LiM cells currently available inside DExIMA's library. *Courtesy of Nicola Piano from his Master's Degree thesis [3]; p. 50, Table 6.2.*

1.2.3 The description of the Out-Of-Memory Logic

The traditional CMOS circuit placed outside the memory can be described in an even simpler way. In fact, the designer has to instantiate the required RTL blocks, always described inside DExIMA's library, and to define the interconnections, without worrying about the relative position of the useful elements. At this point, the software organizes the structure of the Out-Of-Memory architecture, ready for the subsequent analysis phases.

1.2.4 The simulation of the system

As things stand today, the simulation inside DExIMA of the developed Logic-in-Memory system is divided into two separated phases:

- Simulation of the Out-Of-Memory logic;
- Simulation of the Logic-in-Memory architecture.

In fact, although there is already an accurate model for the description of the behavior of the bus, which is able to connect the two sub-parts, it has not been implemented yet in the program code.

Each component belonging to the system is then subject to the same sequence of analysis:

- Static;
- Dynamic.

In particular for the latter, the designer can provide the full algorithms that will be executed on the two circuits. This information has to be detailed inside the configuration files of DExIMA, following a specific syntax[2].

The parameters derived from the static analysis are *space occupation*, *static power consumption* and, the most important, *maximum delay* (i.e. critical path, useful to define the maximum performance of the circuit). Through the dynamic analysis another important information can be obtained: the *dynamic power consumption* of the device under test, more relevant with respect to the static one as it represents the most important parameter in the comparison with traditional microprocessor systems.

In conclusion, DExIMA allows a designer to get an idea of the possibility of Logic-in-Memory systems. The latter, a promising solution to the *Memory wall* problem.
The program, month by month, evolves and continuously expands its functionalities, aspiring to become an effective tool for the design of tomorrow's electronic devices.

[2]For a more exhaustive description about the organization of DExIMA's input files refer to the related documentation, constantly updated as the program functionalities.

Chapter 2

Parallel computing

As mentioned in the previous chapter, one *key-feature* of a Logic-In-Memory systems is the intrinsic capability of parallel computation. Retracing the evolution history of modern computer industry[8], since the middle of nineties both logic and memory products have relied on CMOS technology to improve their performance. However, soon the same performance had to be limited to contain the rising power consumption of these devices.

The solution to the problem, in order to guarantee an increase of computational performance, was identified in the parallel processing of information. Multi-core processors were born in this respect by the middle of the previous decade. Also today, the outputs of data elaboration are achieved as the result of several concurrent computations each of which performed in a different processing core[1]. In this way, the working frequency of the CPU could be scaled down with a trend inversely proportional to the number of integrated cores. Hence, the power limits were effectively respected again. To make the concept clearer, the relationship between the presented quantities can be observed in Eq. 2.1, which characterizes the dynamic power consumption of an integrated circuit. In particular, N_{gate} is the number of integrated transistor, α_{sw} is the activity factor, C_L is the average load capacitance of transistors, f is the working frequency and V_{DD} is the power voltage.

$$P_{dyn} = N_{gate} \cdot \alpha_{sw} \cdot C_L \cdot f \cdot V_{DD}^2 \quad \text{with} \quad N_{gate} \propto N_{core} \tag{2.1}$$

[1] A core is an independent instructions processor, equipped with its own cache and controller and integrated inside the same CPU package. Their structure still relies on the Von Neumann principle

Programming languages and compilers had to be progressively adapted to make the most of those new architectures and *multi-threading programming* started to be talked about. The software tools and the mathematical techniques useful for translating a serial code into a parallel version are covered in the current chapter. Part of the discussion, refers to the contents present inside Chapter 7 of the book *"Algorithms and Parallel Computing"* of F. Gebali[9].

2.1 The available processors for parallel computation

Generally, for the exploitation of parallel computing without considering the expensive design of integrated circuits, different devices can be taken into account, both general purpose and application specific ones.
The choice depends on the characteristics of the algorithm that has to be implemented and on the performance required for its execution. A designer, in this respect, can rely on the following hardware platforms:

- Multi-core Central Processing Units (Multi-core CPU)
- General Purpose Graphical Processing Units (GPGPU)
- Field Programmable Gate Arrays (FPGA)
- Digital Signal Processors (DSP)

All the presented devices, except for FPGAs, execute compiled high-level language codes (e.g. C and C++). The performance depends on the optimizations introduced by the designer inside the input code also considering the characteristic technical details of the chosen device. Therefore, bearing in mind the techniques useful to define parallel quality code is of paramount importance. Few of them will be discussed in the following sections.

2.2 From serial to parallel code

Programming languages, with the advent of Von Neumann architectures, were adapted to the definition of optimized codes with respect to their working principle. Those languages were defined imperative[2] but, most of all,

[2]An imperative language is composed of a set of instructions which modify, step by step, the state of a machine (i.e. the processor, in computer science).

they were meant for a *serial execution*.
When multi-core processors began to spread in the computer market, the same way of thinking about writing code had to evolve again, through an important change of perspective. In this context, the program structure was subdivided into *threads*, small sections of the whole code each of which associated to an available CPU core for its processing. Hence, part of the algorithm, which was more suitable for a parallel execution, could benefit from an important speed-up.

Even today, the written code intended to run on modern processing units is based on these last principles.

2.2.1 The Amdahl's law for parallel processing

The Amdahl's law[10] is a well known formula to computer scientists since it was presented for the first time by Gene Amdahl during a conference in 1967. The formula provides information on the theoretical speed-up factor of a specific algorithm that can be expected, in terms of latency, if part of the processing system, on which it is executed, is improved.

The overall enhancement can be expressed by the following equation, considering a performance improvement factor of s relative to a fraction p of the total system:

$$S_{latency}(s) = \frac{1}{(1-p) + \frac{p}{s}} \tag{2.2}$$

Multi-core processing architectures introduce improvements over the execution of an input code which can be estimated qualitatively by the same expression[11], assuming as meaning for the variables:

- p: the fraction of the code which can be made parallel;
- *(1-p)*: the remaining part that will be executed serially;
- s: the number of available core.

It is interesting to note how much the serial part of the algorithm affects the resulting enhancement since it imposes an upper limit. In view of the above, as well as intuitively, it can be concluded that to make the most of multi-core processors and, in general, for any parallel processing system, mostly parallel codes have to be executed.

2.2.2 The translation process

As mentioned, programs are usually described in a serial way, compromising their performance when they are executed on the modern computational systems. Programmers need to focus their attention to the optimization of the written code considering all the technique useful for the transformation in parallel version, whenever it would be convenient. This *"translation"* process has to be performed by hand because not even the most modern and advanced compilers can do it independently. In fact, the latter are only able to proceed with the *parallelization* of portions of a code definable as *embarrassing* (i.e. perfectly) *parallel*, so trivial cases.

The program needs to be fully restructured, keeping in mind few mathematical rules and some of the most common parallelizing methods to effectively carry out this conversion procedure. The target architecture on which the code will be executed has to be considered from the outset, so as to be aware of its intimate principles of operation and to understand its optimization possibilities.
It is in the programmer's responsibility analyzing the whole algorithm following few basic steps resumed below:

- Identification of the different tasks
- Identification of the dependencies between these tasks
- Identification of the primary inputs set of the algorithm
- Identification of the primary outputs set of the algorithm

An *Algorithm Dependence Graph (ADG)* can be defined through the collected information, as the one in Figure 2.1, where any node represents a specific task while the edges, the data used by them.

From the conformation of these graphs, the type of the algorithm can be identified[3], as well as its highest degree of improvement. Many optimization techniques can be applied to the different cases, some of which will be discussed.

[3]The possibilities are: *serial*, *parallel*, *serial-parallel*, *non serial-parallel* and *regular iterative*.

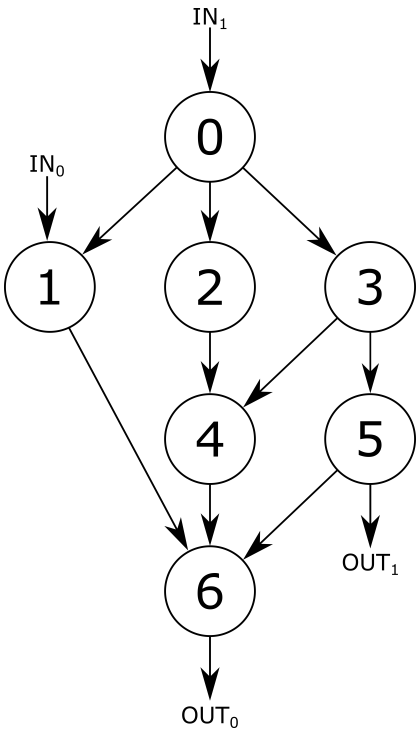

Figure 2.1: An example of an ADG representing a program. As no loop is present, it is defined *Directed Acyclic Graph(DAG)*.

When the algorithm turns out of be optimized, net of specifications, it can be adapted to the target architecture through the subsequent logic steps:

- Mapping the tasks to the available parallel processors/cores
- Scheduling the execution of the tasks, respecting the constraints of data dependency and I/O requirements
- Identifying the data communications between the processors and the I/O

Only at this point the parallel code can be defined. Over the years, many libraries and APIs were born to expand the traditional programming languages to the parallel computing capability of recent CPUs and GPUs. Some of the most important, state of the art, will be deepened in Section 2.4.

2.3 Code parallelization techniques

There are many useful techniques to parallelize an algorithm from a mathematical point of view. The discussed methods are meant to be applied in simple context, where the data dependencies are not so significant and the parallelization possibilities are consistent. Among them, the most promising for the purposes of the present dissertation consider *loops transformation*, so they will be investigated in more detail. Other techniques, less relevant, will be subsequently exposed as an in-depth study.

2.3.1 Loop transformations

Loops represent one of the most effective ways to describe parallel algorithms. However, different types of loops can be distinguished according to their structure and the dependencies that affect the contained data. In order to proceed with an easier discussion, it is appropriate for some definitions to be provided. In particular, loops can be classified as *independent* or *dependent*, while their internal variables as:

- Input variables
- Output variables
- Intermediate or I/O variables

An *independent loop* is characterized by the absence of intermediate or I/O variables. Any of its iterations is uncorrelated to the others, so that they can be directly mapped on a parallel processing unit, not including the execution order. This is the *luckiest* case, from which the best results can be achieved. An example of loops belonging to this category is represented by FIR digital filters, discrete-time devices whose mathematical expression is presented in Eq. 2.3. In particular, N represents the degree of the filter and n the time instant considered.

$$y[n] = \sum_{i=0}^{N} b_i \cdot x[n-i] \tag{2.3}$$

Considering N parallel computing elements and I iterations of the loop, the *speed-up factor* obtained through the parallelization of the algorithm can be

expressed as:

$$Speedup(I, N) = \frac{I}{\lceil I/N \rceil} \tag{2.4}$$

A *dependent loop* is instead characterized by intermediate or I/O variables. In this context, the data dependencies are relevant to determine the degree of optimization of the algorithm. If the intermediate information has no data dependency, each iteration of the loop can be associated to separate processing element, executing the whole loop in a parallel way. Thus the same performance of a parallelization of an independent loop, in terms of speed-up factor, is obtained.
Otherwise, more sophisticated techniques have to be considered, including *loop spreading* and *loop unrolling*.

Loop spreading

The technique turns out to be useful in case of nested loops with simple data dependencies. If the latter are confined within the innermost loop, each iteration of the outer loop can be assigned to different processing elements. In this way, the various parallel elaboration units execute all the iterations of the inner loop. The basic concept is illustrated in Figure 2.2.

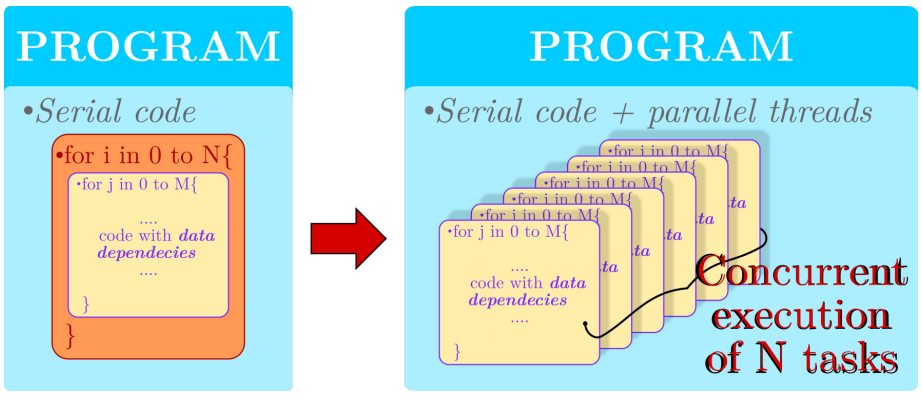

Figure 2.2: An example of the loop spreading technique.

Loop unrolling

Loop unrolling is an optimization technique implemented by compilers and hardware designers to improve the execution of a loop. It reduces the overhead, in terms of computational effort, introduced by the updating of the loop index. In particular, multiple operations belonging to different loop iterations are gathered within a unique iteration. The number of the instructions which can be grouped depends on the data dependencies involved in the cyclic structure. Therefore, the overall iterations are reduced, each of which could be executed in parallel processing elements.

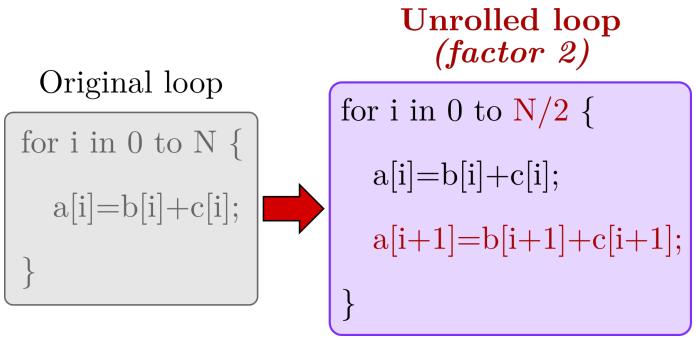

Figure 2.3: An example of the loop unrolling technique.

2.3.2 Other general techniques

The methods proposed in this section are universally applicable, nevertheless, their generality makes them a *design philosophy* to bear in mind rather than a specific prescription.

Problem partitioning

The technique suggests to subdivide the computation of large tasks into smaller parts of similar size. The idea consists in the generation of reduced dimension sections of the original code, possibly devoid of data dependencies and so executable in parallel. The method can be applied many times, in a recursive way, and in this case the technique is called *recursive partitioning* or, quoting the Roman Latins, *divide-et-conquer approach*. Typically, the subdivision proceeds generating, for each task at every iteration, a couple of sub-tasks. Therefore, the algorithm is organized following a binary-tree structure, as depicted in Figure 2.4.

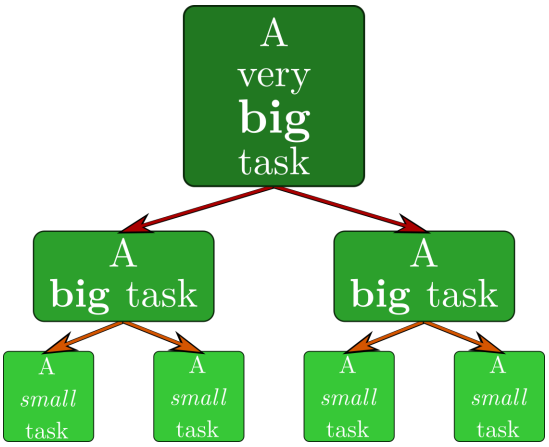

Figure 2.4: An example of the recursive partitioning approach.

Pipelining

Traditionally, the pipelining is an effective optimization technique for the development of integrated circuits. In particular, the execution of a slow algorithm is subdivided in smaller and faster steps, which can be executed both serially and parallel, one per clock cycle. In this way, the latency with which the results are obtained from specific inputs increases. However, the throughput rises too together with the number of computations executed per unit of clock cycle.
In software context, algorithms can be considered to produce results for *successive approximations* at high rate, so implementing pipelining. An example of this technique is represented by the CORDIC algorithm[12].

2.4 Libraries and APIs for parallel computing

The multi-threading programming approach is important for the execution of optimizations on both multi-core CPUs and GPUs. Today, particular attention is given to the latter for their intrinsic predisposition to perform parallel computations. In fact, they are increasingly considered as an advantageous alternative for the implementation of data intensive algorithms which have nothing to do with graphical purposes. For this reason, the concept of *General-Purpose GPUs (GPGPUs)* has been recently spread and it will become more and more important in the next future[13].

The need of an even more consistent growth of computational power has

led many representatives of electronic Industry to come together to form consortia and, in this way, to define new standards for the development of parallel applications on both CPUs and GPUs. Among them, *OpenMP Architecture Review Board* and *Khronos Group* which introduced respectively OpenMP in 1997 and OpenCL in 2008. Separate mention should be made to Nvidia, which is part of both the previous organizations, however, as a leading GPUs manufacturer it decided to develop a new standard for its own products, CUDA in 2007.

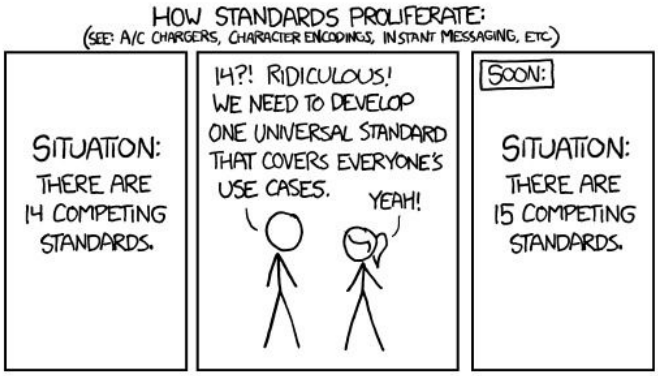

Figure 2.5: "How standards proliferate" *by xkcd.com, licensed under CC BY-NC 2.5.*

An undisputed standard does not exist to date. The adoption of one of them rather than another depends on the target architecture that will execute the defined parallel code. A possible classification of the contexts in which these software utilities are considered the most, where they represent a *de-facto standard*, follows:

- OpenMP: multi-threading applications for multi-core CPUs;
- OpenCL: highly parallel applications for GPGPUs;
- CUDA: highly parallel applications for Nvidia's GPUs.

The three proposed solutions for defining parallel codes will be analyzed, highlighting their strengths and weaknesses.

2.4.1 OpenMP

OpenMP (Open MultiProcessing) is an *Application Program Interface (API)* introduced to make the design of multithread applications easier. It supports multi-platform shared-memory parallel programming[4] in C/C++ and Fortran. It is a collection of compiler directives, library routines, and environment variables for the definition of high efficient and portable codes[14]. The optimizations are fully effective at run-time, i.e. during the execution. Therefore, OpenMP codes have to be run on a compatible operating system (e.g. Linux, Windows and macOS).

The OpenMP API is based of the *fork-join* model, whose principle scheme is depicted in Figure 2.6.

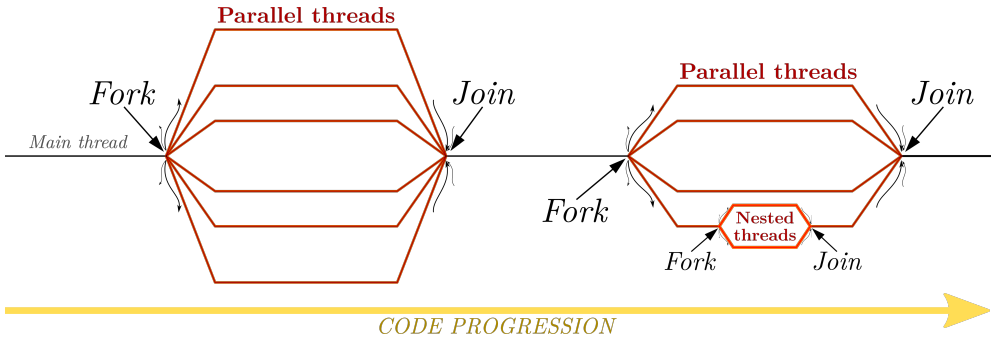

Figure 2.6: Conceptual scheme of the *fork-join* model implemented by OpenMP API.

Traditional programs consist in the union of two types of code: serial and parallel. The aim of this API is to execute, as best as possible, the parallel portion, dividing it in multiple threads. The remaining serial part, which can't be optimized, is left as it is. It is interesting to note that, inside a parallel region, also nested parallel threads can be defined, making possible finer-grain optimization techniques.

In order to make a program compatible with the standard, the starting point is a complete and working version of the code that has to be optimized. As already pointed out, the concurrency has to be considered right from the first line of the code, so that the final application will result tailored to the target architecture. The *divide-et-conquer* approach is then applied in order

[4] A *shared-memory system* is composed of different processing elements which share the same memory address space.

to separate the entire work in different threads. The defined sections must represent *structured block*, i.e. sequence of statements characterized by an entry point at the top and an exit point at the bottom. For each thread the parallelism of the associated code has to be identified, paying attention to eventual synchronization needs. In fact in presence of shared data, race conditions can occur and the designer has to properly organize the computation in a way that the correctness of the final results is ensured.
Therefore, the OpenMP constructs have to be inserted inside the original code, most of which are *compiler directives* whose prototypes are present in the include file *"omp.h"*.

Generally, when a code section turns out to be advantageous if executed in parallel, the designer shall assess the degree of parallelism sufficient to meet the performance goals. Here, a *fork* region is defined and the master thread is spawn in a team of concurrent threads. At the end of the parallel section a synchronization barrier has to be considered to guarantee that every thread has been executed.

2.4.2 OpenCL

OpenCL (Open Computing Language) is a framework for general-purpose parallel programming of heterogeneous systems (e.g. CPUs, GPUs or FPGAs) and, in particular, hardware accelerators[15]. OpenCL consists of the same three components[16] of the previous standard:

- Language specification
- Platform layer API
- Run-time API

In particular, concerning the language specifications, OpenCL programming languages relies on C99 and C++11 with restrictions and added extensions. In this way, the programmer can focus in the definition of a parallel code, optimized for a specific architecture.

The target device is modeled in a structured way and it is subdivided in several *compute units*. Each compute unit[5] is then composed of multiple

[5]For a multi-core CPU, the definition of compute unit is coincident with the one of a core.

processing elements which can work in parallel.
The structuring, through which the hardware is considered, is reflected in the code. There, two components can be distinguished: the *host programs* and the *kernels*. The former are useful to organize the correct execution of the latter, properly managing the memory hierarchy. Kernels are sections of code, comparable with normal C-functions, which contain the instructions that benefit from a parallel implementation.
An heterogeneous system, in these terms described, considers a CPU (the host) and one or more accelerators (the devices on which are run the kernels).

The great complexity of designing OpenCL code is rewarded with the possibility to write optimized cross-platform programs, accelerated through parallel processing.

2.4.3 CUDA

Nvidia's CUDA (Compute Unified Device Architecture) is the name that refers to both a parallel computing platform and a programming model, developed by the same company. In particular, the provided API is considered to develop general purpose applications compliant with CUDA-enabled Nvidia's GPUs[17].

The algorithm to be optimized can be developed through a wide variety of programming languages (including also C, C++ and Fortran), making quickly and easily accessible the hardware resource. Also in this case, the capabilities of the compatible languages is expanded to make the expression of the parallel code more comfortable to developers.

The structure of the code has to take into account the general structure of the architecture which will execute the algorithm and, in particular, a system composed of both a CPU and a Nvidia's GPU. It is appropriate to consider, indeed, that a Graphics Processing Unit is a *coprocessor*, an hardware unit that supplements the functions of the primary processor (i.e. the CPU).
Similarly to OpenCL, the whole CUDA code consists of the union of two different parts, which will be intended for the two processors (called the *host* and the*device*), and each of which refers to a separate memory space. The developer has to identify the sections of the original algorithm which will benefit of an execution inside the GPU and to organize them into functions, called *kernels*. Inside each kernel, many threads can be defined that will be

mapped on the several parallel processing elements, characteristic of a SIMD architecture[6], the one of the GPUs. The developer is responsible for the proper management of the code transfer from the *host memory* to the *device memory*, so that the latter could be correctly processed by the GPU.

Unlike the other presented APIs, CUDA is the only one to be a *proprietary software* and it is bound, as previously mentioned, to the use on Nvidia's GPUs only.

[6] According to Flynn's taxonomy[18], a *Single Instruction Multiple Data (SIMD)* architecture is composed of many processing units which execute, at the same time, the same instruction but referring to different datasets.

Chapter 3

Compilers

Compilers are computer programs with the purpose of translating an input *source code*, expressed through a specific programming language, into an *object code*, expressed in another language or in a machine code[1].

The first compilers appeared starting from the second half of the last century, in parallel with the advent of the first programming languages from *Assembly* to the ones with an increasing abstraction (e.g. Fortran). Precisely, the idea consisted in abstracting the way through which the algorithms were described to increase the flexibility of programmers. In the last seventy years of history, Computer Science has had a strong development and programming languages, that have been introduced, required the *birth* of increasingly complex compilers. The latter were meant for being more and more *smart*, capable of autonomously providing advanced optimizations for the input programs, in order to generate efficient executable codes. Moreover they were defined to detect any error in the input code and to properly report them to the programmer. Therefore, compilers have always been a valid aid for the development of formally correct programs.

Compilers are structured in a modular way, to compensate their complexity and, above all, to make the definition of improvements and upgrades easier. In this chapter, the general structure of a modern compiler is presented together with an overview of today's most important and widespread ones. The discussion takes its cue from the book that is known to specialists in the sector as *The dragon book*[19].

[1] A machine code is a low-level programming language consisting in a sequence of *instructions* that can be executed by the CPU of a processing system.

3.1 The structure of a modern compiler

The compilation process can be subdivided into common logical phases, which find different expression among specific compilers. Traditionally, its structure is organized in two main parts, as depicted in Figure 3.1: the *front-end* and the *back-end*.

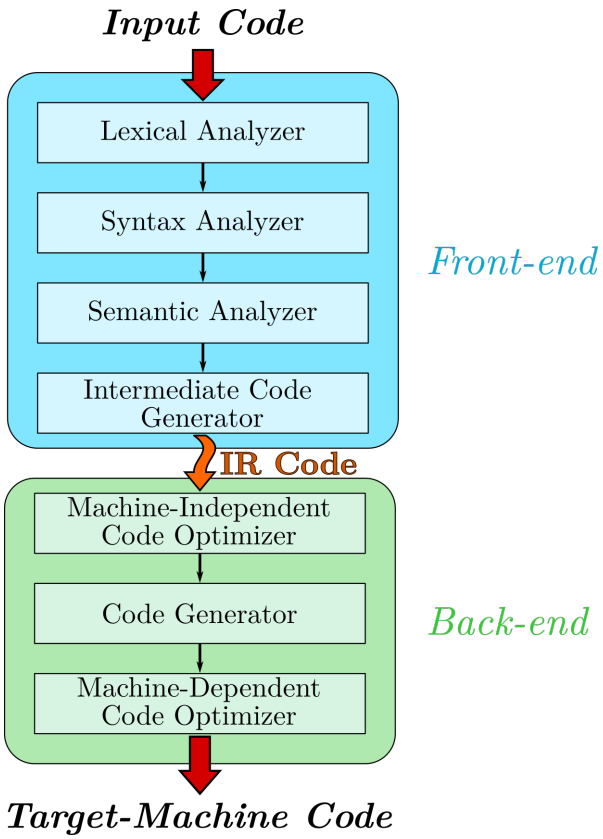

Figure 3.1: The logic scheme of a modern compiler.

Together with the two presented sub-parts, another important element has to be considered: the *symbol table*, which stores information about the source code useful all along the compilation process.

In the following, a brief description of the internal components of a generic compiler is given to better understand how the translation is performed. However, as the intimate structure of a compiler is not among the purposes of the present dissertation, the argument will not be deepened. For further details, refer to the above-mentioned book.

3.1.1 The compiler *front-end*

The compiler front-end is responsible for the analysis process and, in particular, it parses the input program and it checks the general correctness of the code, both from a syntactic and semantic perspective. The output of this block consists in the same input algorithm but expressed into another form, the so called *Intermediate Representation (IR)*. The latter, whose structure can vary, is constituted by a sequence of low-level instructions intended for an ideal and abstract machine.

Lexical analysis

This first phase of the analysis process reads the input characters belonging to the source code to transform them into *lexemes*. The latter are continuous characters grouped together, from which the Lexer produces *tokens*. Each token is then characterized by two elements:

- *Token name*: an abstract symbol useful to identify the specific lexeme
- *Token attribute*: the relative position inside the symbol table where the original lexeme characters are stored

Thereby, a classification of the names used inside the program is performed. The information derived from both lexems and symbol table are then passed to the subsequent analysis phase.

Syntax analysis

The current phase implements the *parsing* of the input lexems for the generation of a *syntax tree*. In particular, it considers the *tokens* generated by the lexer to represent the grammatical structure of each instruction. The tree is organized in nodes distributed along a branch and its leafs: the internal nodes are constituted by operators (e.g. $+$, $-$ and x) while the derived nodes, so the external ones, represent the arguments of the operations. The different operations are positioned in the branch following the conventions of arithmetic (e.g. a multiplication is performed before an addition).

Semantic analysis

The syntax trees coming from the previous phase and the information inside the symbol table are considered by the semantic analyzer to verify that the

instructions are consistent with the target language specifications. After many checks, information is collected to enrich the contents of the symbol table for the subsequent elaborations. Moreover, possible errors are reported if any inconsistency is found.

Intermediate code generator

At the end of the analysis process, a machine-like intermediate representation of the input source code is produced. Despite the variable forms, two properties have to be ensured by this representation:

- It needs to be easy to generate
- It needs to be easy to derive the target code from it

Leaving out the details, the common structure for the IR code is based on *three-way instruction* (i.e. for each instruction three operands are considered at most).

3.1.2 The compiler *back-end*

The compiler back-end, which works in tow, considers the IR code and it performs the actual translation into the target language. During this operation, it also implements algorithmic optimizations some of which general, others characteristic of the system on which the program will be executed.

Machine-independent code optimizer

The input IR code is here optimized though specific algorithms, typical of each compiler. The aspects that can be privileged are *performance* or *short dimension* of the output code, as well as the related computational effort in their execution which affects the total *compilation time*. Among all this is one of the most critical phases and the choices adopted there can have significant relapses to the quality of the resulting code.
After the elaboration, the output code is provided as a reworking of the inputs, expressed in the same format.

Code generator

The optimized IR code is finally converted into the output language, considering also the information previously stored inside the symbol table. Depending on the effective target of the compilation process, be it an hardware

machine or another software, different strategies can be adopted. The requirements on this translation phase are severe and the equivalence, in terms of algorithm, between the input IR and the output codes shall be guaranteed. Moreover, the generated code must be of high quality, so that it can make the most of the potential of the target entity. The main tasks of the code generator are:

- Instruction selection
- Register allocation and assignment
- Instruction ordering

The process is an optimum one, however, over the years the related complexity has made the adoption of heuristic techniques necessary.

Machine-dependent code optimizer

Generally, the code generators provide codes already optimized for the target. However, some compilers use *naive* approaches for the code generation, hence a further optimization phase is required. The results of these elaborations are not necessarily optimal and typically simple transformations are applied locally in the code. Characteristic modifications introduced are:

- Algebraic simplifications
- Redundant instruction elimination
- Flow-of-control optimizations
- Use of machine idioms

3.1.3 Final comments on the internal organization of a compiler

The strength of the presented subdivision is based precisely on the translation of the original input algorithm into the IR form. Compiler designers can focus only on the part they are interested in, relying on the existing other. For example, if there is the need to develop a compiler for a new programming language but intended for an existing target architecture, only the front-end component has to be rewritten. On the contrary, if a new target system or architecture has to be supported, only a new back-end module needs to be

defined. The applications, in which the whole compiler has to be completely designed from the head, are rare.

It is important to highlight how the optimizations that can be performed on the IR code are abstract from a specific context and so they can be treated as *mathematical problems*, acting on algorithms. Hence, many techniques, described in details in literature, can be adopted to find an optimal for any *recipient* of the output code.

Finally, the compiler theory can be considered to define translation processes which could involve both hardware and software targets. Therefore, also other programs can benefit from compiler services and receive the derived output code. Among them, the modern EDA tools for the design of electronic devices, whose related compiler is called *High-Level Synthesizer*. This last topic will be deepened in the following chapter.

3.2 The most popular C/C++ compilers

In the panorama of modern C/C++ compilers, both *commercial* and *open source* products are available. Even though the latter are the most diffused and adopted, they often do not represent the best performing[20]. However, free compilers, by their nature, allow anyone to delve into their implementation details in order to understand how they work and, eventually, to introduce improvements. Among the possible modifications, a compiler designer can extend the compatibility of these software to additional source codes or target devices. The success of the spreading of open-source compilers in the market is also related to the above-mentioned aspects.

Today, the two main free-license compilers are *GCC*, from the GNU project, and *Clang*, from the LLVM project. GCC has a very *ancient* development history and its first stable version dates back to 1987. Over the years, it has had the opportunity to grow even more and to establish itself as the reference compiler for the majority of applications.
Clang is a younger compiler, whose first release appeared in 2007. The software has received great impulse by many Industries during the last decade, and it has been diffused to the point of becoming the direct rival of GCC.

Within the same article cited at the beginning of the section a comparison

between these two compilers is presented. The differences between the two can be summarized as follows:

- if a developer needs to improve the performance of its own code during the execution, the choice falls on GCC;
- if a developer needs to build a large project fast, Clang has to be taken into account.

However, for a compiler designer the choice among the two software programs is not only determined by *benchmarks* results and some further considerations have to be made. In particular, the context they come from needs to be taken into account, so the *GNU Project* and the *LLVM Project*.

3.2.1 The GNU Project

The project, whose name is a recursive acronym of *"GNU's not Unix"*, was funded by R. Stallman in 1983 with the aim of creating and distributing free software destined for any computing devices, from the operating system to the applications which are run on it. The concept of freedom is particularly interesting and it is defined by the *GNU General Purpose Licence (GNU GPL)*. Any user is free to:

- run the program, for any use;
- study how the program works and to modify it;
- redistribute copies of the program;
- redistribute modified copies of the program.

Another important prescription associated to the project is the *copyleft*, in which any derived product from the GNU software directly inherits the same licence (i.e. the GNU GPL).

The community around the project includes, not only software enthusiasts, but also great Companies from System to Chip Vendors (e.g. RedHat, Intel, ARM and IBM). They provide constant support in order to guarantee software compatibility with their own products.

Many resources are made available also to compiler developers, from consulting services to an extensive documentation, without considering the numerous contributors who pool their expertise both in literature and online.

3.2.2 The LLVM Project

The project started at University of Illinois at Urbana–Champaign in 2000, under the responsibility of V. Adve and C. Lattner. Their aim was to provide a modern compilation strategy capable to support both static and dynamic compilation of arbitrary programming languages. Today, it represents a compiler framework that embodies these original principles and Clang specifically serves as the related front-end compiler.
The definition of free license, under the *University of Illinois/NCSA Open Source License*, is more fleeting with respect to the GNU GPL one. In particular, it allows to derive also commercial products from the original project and this is one of the reasons behind the success of LLVM[2].

Also around this project a lively and varied community is gathered, from both the private and the commercial worlds. Moreover many competitive resources are provided to compiler developers too, despite the important *age difference* with the GNU Project.

[2]From 2019 the license became *Apache 2.0*, extending even more the capability of making use of LLVM in commercial applications and so the possibility of adoption in Industry.

Chapter 4

High-Level Synthesis

The concept of High-Level Synthesis, also called *behavioral synthesis*, refers to the capability of a system to translate a mathematical model, expressed through an algorithm, into a particular hardware implementation. Over the past few decades, the *Electronic Design Automation(EDA)* Industry has shown growing interest in this research field and today many efficient tools are available on the market. Nowadays, the EDA software are an essential support for engineers in the design and analysis of electronic circuits (e.g. Integrated Circuits and Printed Circuit Boards), so that the high complexity introduced by the the large integration capabilities of transistors can be overcome.
However, the reasons to explain why High-Level Synthesis represents a current topic lie in the past and, in particular, in the evolution history of EDA tools over the years.

In the current chapter, mention is made about these motivations to better understand the importance of High-Level Synthesis. Afterwards, the *state-of-art* of the most modern systems which implement it is described, dwelling on their general structure.

4.1 The historical stages of the EDA Industry and the *troubled* research on HLS

During the 40^{th} Design Automation Conference held in 2003, Professor A. L. Sangiovanni-Vincentelli presented an analysis of the history of EDA[21],

since the first tools appeared, subdividing it into three evolution eras[1].
It is important to emphasize how the progression of the success of this Industry has gone along the evolution of the Electronic Industry, from the very first moment.

Alongside this analysis, particular attention will be paid to how the research on High-Level Synthesis has developed[22].

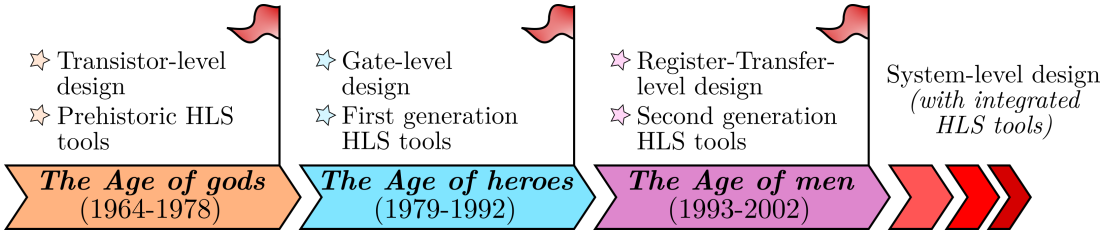

Figure 4.1: The evolution history of both EDA and HLS tools.

4.1.1 The *Age of gods (1964-1978)*

The first period coincides with the foundation of the EDA Industry itself. During this time, the *groundwork* of modern EDA tools was laid and, in particular, the key themes that were explored are:

- Circuit simulation
- Logic simulation and testing
- MOS timing simulation
- Wire routing
- Regular arrays

Since the beginning, the aim was to reduce the effort, and so the time, of the design process of electronic circuits. However, the input language was complicated and tools suggested little foresight in understanding the market's evolution. As a result of these problems, with the addition of an important lack of innovation, the available software became obsolete soon. As proof of

[1] As he explained in the relative article, the name associated to each period is inspired by the masterpiece *"Scientia Nova"* of Philosopher Giambattista Vico, whom retraces mankind history.

the above, none of them is alive today.

Academic researches about EDAs had taken into account also the *synthesis* aspect, even if it had limited impact.

4.1.2 The *Age of heroes (1979-1992)*

The subsequent period saw, on the contrary, an explosion of the EDA tools from *logic synthesis* to *formal verification* and from *system-level design* to *hardware acceleration*. The most important achievements were:

- Faster verification and testing procedures, orders of magnitude with respect to the ones conducted through *Spice*, also thanks to the introduction of *formal verification*
- Simpler circuit layout definition, with the implementation ad advanced techniques of *simulated annealing* and for *placement* and *routing*
- Design through logic synthesis, with the support of *logic optimizers*
- Introduction of the well-known Hardware Description Languages (HDLs), VHDL and Verilog, which made possible to represent digital circuits more effectively
- Rising interest in parallel computing and parallel architectures, both seen as an indissoluble unit in the design phase[2]. Particular attention was given to *hardware accelerators*.

In this respect, the early researches about High-Level Design and High-Level Synthesis appeared, as a bridge to the *system-level design*. The idea consisted in growing the degree of abstraction during the definition of an electronic circuit and so increasing the *productivity*. However, these tools revealed too complicated and not mature enough to meet the needs of the market.
Due to this reason, the High-Level Synthesis approach didn't make breakthrough into Electronics Industry and there was not substantial investments in this field, in those years. Therefore, these studies remained within the walls of Universities.

[2]As mentioned in Chapter 2, the design of optimized parallel architectures is strongly bound to the algorithm which will be executed on it. Under these circumstances, we talk about *co-design*.

4.1.3 The *Age of men (1993-2002)*

The last era overlaps the period of maximum expression of integration capabilities within electronic circuits. At the same time, the Moore's law became increasingly difficult to comply with, as previously mentioned, and technical innovation started to slow down.
Therefore, these aspects were reflected to EDA tools, which did not undergo major innovative upheavals. However, they increased in complexity to face the technical challenges introduced by the massive integration of transistors (e.g. non-ideality, incidence of parasitic parameters and leakages).

Major EDA companies and, in particular, Synopsys, Cadence and Mentor Graphics launched the fist commercial HLS tools, arousing great interest. However, they revealed unsuccessful in the effective implementation. The reasons behind this failure lie not only in the not so promising results derived by the synthesis process, but, above all, for whom the tools were thought to. In fact, among the other choices which were discovered to be problematic, there was the adoption of behavioral HDLs as inputs. Software and algorithm designers didn't find it affordable to learn an additional language and how to use the new tools. RTL-designers, on their part, believed that the needed effort for changing the already well-founded design perspective wasn't worth it, considering the poor quality of the results.
Victims of the context and of their own deficiencies, also these new software programs were ignored.

Despite the dissertation of Professor A. L. Sangiovanni-Vincentelli concludes the analysis considering the first years of the new millennium, it is reasonable to extend the *Age of men* to the last two decades.

4.1.4 The *Contemporary Age (from the early 2000s)*

At the beginning of the new millennium, great impetus was given to High-Level Synthesis by more mature versions of the available commercial tools together with other new ones that appeared in those years, when new companies entered the market (e.g. Mathworks and Xilinx). Algorithm and systems designers were finally involved, introducing higher-level input languages like *C* and *C++*. More attention was paid to RTL-designers too with the definition of new standards, *SystemC* and *SystemVerilog*, as evolution of HDLs. The latter inherited part of the typical syntax of VHDL and Verilog

but also a greater flexibility from *C++*. The quality of the results of the synthesis process saw enhancements for both the definition of the *dataflow* and the *control*. The automatic synthesis was then embedded into more complex systems for *Electronic System-Level (ESL) Design and Verification*, so that HLS tools could represent a complementary element, supporting the entire design process. Therefore, the sales for the industry started to grow significantly, as depicted in Table 4.1.

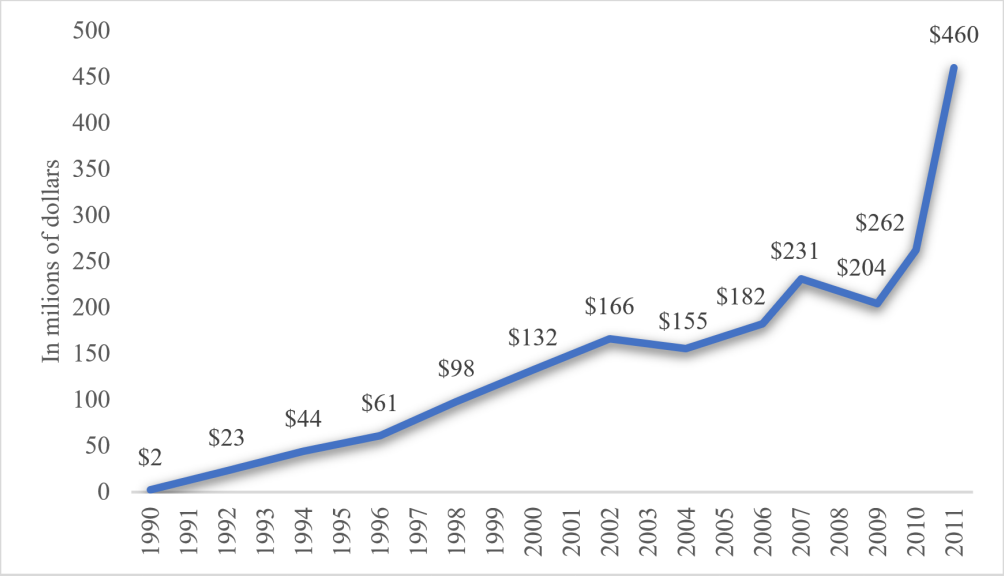

Table 4.1: ESL Market Trend 1990-2012. *Gary Smith EDA statistics at Design Automation Conference (DAC), 2012.*

In order to achieve these successes, some decision revealed fundamental. Firstly, the reduction of the field of action of the tools. The original idea of HLS researches consisted in the definition of a software capable of exploring a wide *space of solution*. However, the approach was too pretentious and its *failure* has been written in the history. The new tools was designed for working on more specific target architectures and, in particular, *ASIC*, *DSP* and *FPGA*. In this way, the synthesis problem was circumscribed, resulting easier to solve. Moreover, the tools were optimized to work with specific class of algorithms, the ones from which reasonable good results were expected.
Secondly, the repeatedly stressed choice of high-level input languages. Programming languages like *C/C++* are *by nature* capable of expressing algorithms, also complex ones and, above all, they are target independent. At an higher abstraction level, mathematical optimizations reveal more flexible,

so powerful, with respect to ones possible at RTL-level.

The evolution of the Electronic and EDA Industries found a converging point, this time including also mature supports for HLS. FPGAs found their fortune on the market, with a wide adoption not only for prototyping purposes. Alongside them, hardware accelerators gained importance as a result of the growing complexity of modern electronic systems, including the *Intellectual Property (IP)* cores, also called *blocks*. Interesting articles about FPGAs and their diffusion in the market are [23] and [24].

4.2 The modern HLS Tools

As previously discussed, the modern design of electronic circuits or applications cannot disregard the problem of the power consumption. Today any device off-the-shelf has passed though a more or less significant optimization process, during the design phase, to reduce its consumption. Low power devices are more and more diffused, especially for the mobile applications like *smartphones*, *wearable* or *IoT sensors*. In all of this pervasive technology, a reduced amount of energy is stored inside the batteries. However, having a long *off-line life* is a prerogative even before being a specification.

This optimization procedure is costly, in terms of design time and validation, and it results significant considering that it contributes to the overall increase of an already substantial complexity. The concept of *design reuse* is widely adopted in an attempt to contain the phenomenon and in this context IP cores have their place. These *electronic blocks* implement specific algorithms or functions in an optimal way. Their functional models are provided to the RTL designer in order to allow her/him to include them inside a more complex design. The same models are then *parameterized* and many characteristics can be defined by the designer (her/him)self. Implementation details are known only to the IP core provider, only the functionality is of the final user's interest.
As can be guessed, IP blocks are complex in their internal structure, making any operation aimed at extending or modifying the associated functionalities difficult (e.g. in case of changing in the technology node, standards or specifications).
The IP cores are defined in a limited design space and the implemented algorithm is first described through an high-level formalism. For all these

reasons, the Intellectual Property blocks represent a valid candidate for the exploitation of HLS capabilities.

The mathematical models can be expressed through an high-level language and many parameters can be modified at *compile-time* allowing the synthesizer to adopt all the possible optimization directly on the algorithm. Among the noteworthy, parallelization techniques represent the most effective, especially loop transformations, pipleining (previously described in section 2.3) and retiming. Therefore, different hardware implementations can be obtained from the same source. Industries have their internal HLS tools to easily generate the needed IP cores.

Another reason for the rising adoption of High-Level Synthesizers lies in all the software tools working around them. In particular, the ones useful in the *verification* process, one of the most important steps of today design. The circuit resulting from the synthesis stage is proved directly with the same test vectors of the high-level input algorithm, properly adapted in an autonomous way[3], in order to guarantee equivalence between the high-level model and the circuit itself.
Other tools are needed to extract from the synthesized circuit all the characteristic "figures of merit", as *switching activity* and *reachable performance*. Everything happens inside the same development environment and the information can be obtained in a reasonable time, reduced with respect to traditional design process. So, the productivity of designers rises and they can invest the excess time in other optimization procedures.
A wider perspective on these arguments can be found in article [25]. An interesting research on the state-of-art of available HLS tools is represented by [26].

The same reasoning can be applied in the design or in the prototyping on FPGA. An example of the current applications of synthesizers on these programmable devices concerns the development of wireless communication network. Over the years, HLS tools have been considered in 3G/4G designing [27] and today the same is carried out with 5G technology. To deepen the available HLS solutions for the *FPGA world*, please consider the articles [28] [29].

[3] A standard methodology, widely adopted, for the automatic verification of digital circuit is the UVM.

4.2.1 The typical structure of an High-Level Synthesizer

In the analysis of the general structure of an HLS tool, academic works have been considered for two main reasons. Firstly, for an historical motivation as these software were born in University research centers and there they had the opportunity to grow in a lively environment. Secondly, commercial software are protected by patents and most of their implementation details are not accessible, differently from the *open source* projects developed in Academia.

In particular, two interesting projects were examined, which present, despite many differences, the same organizational structure: the LegUP project [30], developed at University of Toronto from 2011 and Bambu[31] developed at Politecnico di Milano from 2013.

A closer look at LegUp and Bambu projects

LegUP

The LegUp High-Level Synthesis tool is under development at the labs of Professors Jason Anderson and Stephen Brown at University of Toronto since 2010. The project was meant to introduce a resource to experiment HLS algorithms in the definition of FPGAs accelerators inside a computational system provided also by a CPU. The program, written in $C++$ and belonging to the C-to-Verilog category, is based on the LLVM framework and it is constituted by few optimization algorithms and a back-end. The input code is directly compiled by Clang in order to derive from it its Intermediate Representation. Then, LegUp works on it and, also thanks to specifications introduced by the final user through a configuration file, it generates the Verilog code intended for the target FPGA. Along with it, also the machine instructions useful for the interfacing between the accelerator and the microprocessor are provided. After the definition of the implementation details, LegUp organizes a testbench for the debugging process too. It is important to highlight that the input code can be defined only through a subset of ANSI C, whose specifics are reported inside the program documentation.

Specific support for many commercial

FPGAs (e.g. Altera Cyclone II, Altera Stratix IV and others) is made available, where synthesized circuits have been verified by the same research group.

Two versions of the synthesizer are available: the *free* one until now discussed, intended for a non-commercial use and whose last release is the 4.0 of 2015, and a *commercial* one, derived from it and currently under development.

Bambu

The Bambu High-Level Synthesizer, categorized as C-to-HDL (i.e. both VHDL and Verilog formats are supported as output files) and written in $C++$, was introduced into 2013 from PhD C. Pilato and Professor F. Ferrandi from Politecnico di Milano. The program belongs to a wider project, named *PandA*, whose aim is providing a support to designers for the exploration of *hardware-software co-design* solutions. The target of the synthesis process is the arrangement of an FPGA accelerator for memory-intensive applications. Originally, the software was designed to work with the GCC front-end, however, the compatibility with Clang has been also introduced later. Bambu parses as input code the Intermediate Representation of the source algorithm and it applies target-independent optimizations. After that, the synthesis is executed and the description of the final architecture is provided. Also the related test-bench is generated, considering the specifications defined by the designer. The input code that can be processed is a restricted version of ANSI C, whose details are present inside the documentation.

The software is distributed under the GNU GPL v.3 and so it represents a completely free software. It is provided inside the whole PandA framework, updated to version 0.9.6 of 2020.

They are both *C-to-HDL* tools, performing the synthesis process through common steps and targeting FPGAs. As the input algorithm has to be *converted* into another form, passing from a C code to a HDL, the synthesizer actually represents a *compiler*. As previously expressed in Chapter 3, compilers are very complex computer programs but they have the benefit of being organized in a hierarchical structure. In this way, the concept of *reusability* is widely guaranteed, allowing the developers to write only the code that is

necessary to expand the existing functionalities.

The accepted input code refers to the ANSI/ISO C standard, however some advanced features of the language are not included. Only constructs which make sense with the objective of the synthesis are available, as for any HLS tools. Examples of these discarded functionalities are *dynamic memory usage* and *recursive functions*. As a matter of fact, if the specifications of the available commercial High-Level Synthesizers are considered, the language of the input code results depleted of some standard *flexibility* but enriched in many preprocessor directives and actual functions to support and to properly drive the synthesis process. Discussing about *specialized versions* of input languages is more appropriate and engineers have to take this fact into account when they approach these tools.

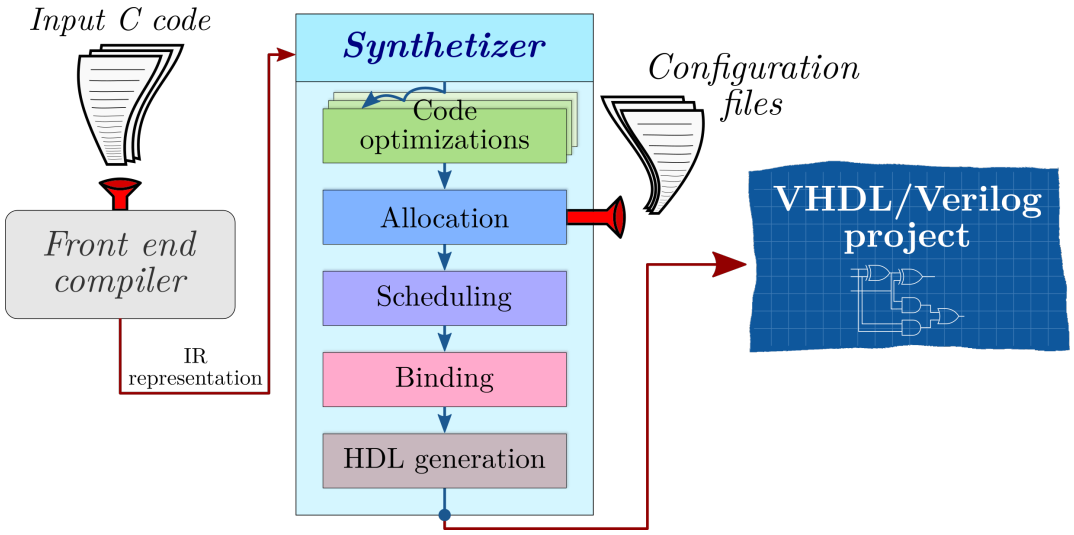

Figure 4.2: Basic scheme of the typical functional blocks inside a High-Level Synthesizer.

Considering again the two HLS tools, they are constituted by five main components, which act in cascade starting with the *Intermediate Representation* of the input C-code, properly generated by the compiler *front-end*:

- *Specific optimization algorithms*: they are useful for the organization of the IR code to be compliant with a subsequent steps of translation. The techniques operate at algorithmic level and some of these have been previously discussed in Section 2.3.

- *Allocator*: it is responsible to read the configuration files in order to drive the synthesis process with the constrains imposed by the user. Typical settings regard the characterization of the target device (i.e. the FPGA) which will implement the produced circuit and the timing and resource constraints.
- *Scheduler*: one of the most critical parts of the synthesizer, it implements specific and optimal scheduling algorithms over the provided instructions to organize the *Finite State Machine* of the final architecture.
- *Binder*: it maps the different instructions to functional units which will implement them.
- *HDL generator*: considering the information derived both from the scheduling and binding processes, it generates the output file, formatted according to the target HDL.

Chapter 5

Conclusions

During the pages of this fist part of the dissertation, all the arguments that underlie the Octantis project have been discussed. The need of overcoming the Von Neumann bottleneck led to the *birth* of the Logic-in-Memory architectures. The promising results of the associated researches gave *energy* for the development of a simulator engine, DExIMA, in order to proceed these studies in a more comfortable way. A watchful look to the future has been preserved, providing the software with both a modular structure, for constant updates and improvements, and a flexible library, open to new experimental technologies. Octantis makes its appearance in this lively environment, as an important contribution to the progress of these researches.

Alongside, the *guidelines* have been laid down for the project. In particular, the structure of a typical compiler and of High-Level Synthesis tools have been described in details. Furthermore, the mathematical techniques to define parallel algorithms and the related supports to bring them into the software context have been presented.

In the part of the document that follows, the Octantis project is deepened, starting with the philosophy it is based on. Its structure is detailed and the main implemented algorithms are discussed. At the end, the results obtained by Octantis during the synthesis process of some reference architectures are argued about.

Part II

Octantis, a Logic-in-Memory explorer

Chapter 6

The Octantis project

Regarding the project

Tracing the history of the project from the beginning, Octantis takes its name from both an object and a star. The first is the octant, a measurement instrument invented between the XVII and XVIII centuries which revolutionized the navigation. It was introduced together with the sextant to replace all previous navigational instruments. The second is the giant star sigma-Octantis which belongs to the constellation of Octans and which is officially the current South Star, named Polaris Australis for its lucky position and whose counterpart is the Polaris one. The idea was to create a software for the exploration of possible alternatives in bringing computational systems into being, changing the current perspective and dictates of modern digital electronics.

6.1 Introduction

Octantis presents itself as a software useful for the exploration of *Logic-in-Memory (LiM) architectures*. By its nature it's a flexible solution to analyze an input algorithm described in standard C language and to identify which LiM architecture would implement it better.

Octantis takes for input a C code whose expressiveness is limited. Only a subset of operations is allowed, the ones which are closer to the Hardware Description Languages *(Verilog or VHDL)*. The algorithm so described must be accompanied by a configuration file, in which the LiM designer has to

impose the constraints that Octantis considers to model a LiM architecture in accordance with them. The subset of C-code that has to be considered for the description of an input algorithm, has been widely described inside the program documentation. However, as a general rule, the designer has to be aware of what she or he is writing, keeping in mind the capabilities of a LiM architecture. For example, instructions for the management of the memory are not allowed, simply because they would not make sense. In case of ambiguity, Octantis discards these instructions and proceeds in the mapping of the others. At the output of the program, the LiM architecture is described through the DExIMA configuration files, in order to enable the simulator engine to perform the related analysis.

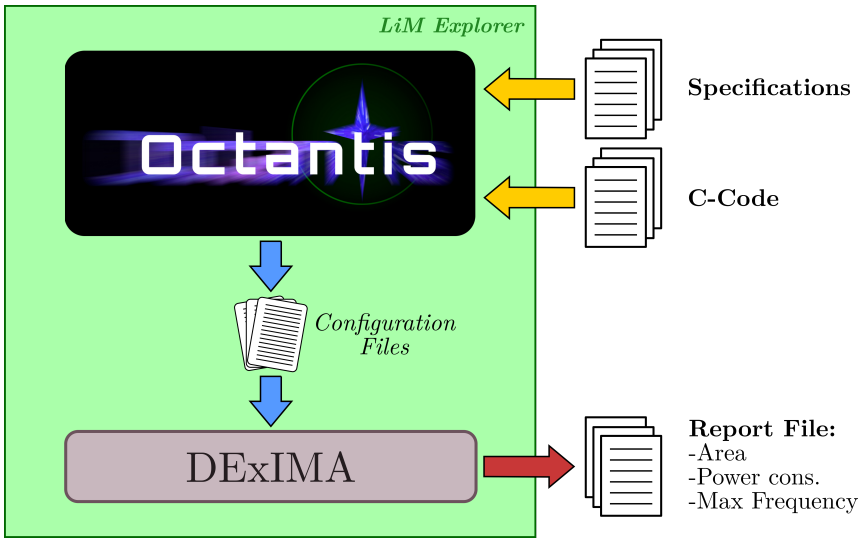

Figure 6.1: The "binary system": Octantis and DExIMA.

Octantis, together with DExIMA, represents an agile tool which allows the designer to get an idea of what the benefits might be in the implementation of a specific algorithm through a Logic-in-Memory architecture. They merge in a unique software of first approach in the design phase, to understand if the proposed solution could be considered valid. The term of comparison is the execution of the same algorithm in a traditional microprocessor system.

6.2 More details about *The LLVM Project*

The Octantis' project considers the *LLVM compiler infrastructure* to realize the whole process of translation from C-code to DExIMA input files. The

LLVM framework, previously mentioned in Section 3.2, provides libraries useful to optimize and generate a code for a target architecture starting from an input source. It is distributed under an open-source license, which allows a free personalization of each of its components. In the panorama of C compilers, the choice fell on LLVM, to the detriment of the tools provided by the competitor *GNU-GCC*, considering specially the flexibility of the former infrastructure, which can count on a more active community of developers and richer technical documentations.

The LLVM project is in turn composed of different sub-projects, among which the *LLVM Core libraries*, *Clang* and *Polly*.
A brief description of the listed components is given in the following, as they are considered important for the definition of the Octantis' project and, perhaps, for its future developments. However, to better understand the basic terminology, some information about the LLVM *Intermediate Representation* and its structure is argued before them.

The discussion dwells on the main points that have revealed useful for the development of the researches. For a more detailed descriptions of the presented arguments, please refer to the official documentation, whose sections are gradually cited along the next pages. Finally, an interesting reading to get in touch with the whole LLVM world, despite being a bit "dated", is attached to the bibliography[32].

6.2.1 The LLVM IR Language

The LLVM Intermediate Representation, also called *LLVM assembly language*, represents the core around which the whole LLVM project revolves. It is the meeting point between the source code and the target one. In fact, the algorithm, whatever the input language through which it is formalized, is explicitly expressed by the front-end compiler into this common representation. All the optimizations are performed on the code so defined and any back-end acts on it to produce the output code. Therefore, the general structure of this *link* is discussed in the next paragraph.

The LLVM IR is considered a low-level language, expressed in a form similar to the *three address code*, however, enough flexible to represent any kind of high-level languages[33]. In terms of Instruction Set, it is classifiable

as *RISC*[1]. There are three equivalent forms, each useful for a particular application:

- An in-memory compiler IR
- An on-disk bitcode representation
- A human-readable form

In particular, the latest version listed is meant for debug purposes and to directly *visualize* the results of the analysis and transformation phases that are run on the input algorithm. Only this representation is discussed in the present section. An example of an LLVM code defined in this way is depicted in Figure 6.2.

```
; The performed operation is: C = A + B
define dso_local void @addition() #0 {
  %1 = alloca i32, align 4 ; Stack allocation for operand A
  %2 = alloca i32, align 4;  Stack allocation for operand B
  %3 = alloca i32, align 4 ; Stack allocation for operand C
  %4 = load i32, i32* %1, align 4; Load op. A inside an internal register
  %5 = load i32, i32* %2, align 4; Load op. B inside an internal register
  %6 = add nsw i32 %4, %5 ; Execution of the addition
  store i32 %6, i32* %3, align 4 ; Store the result inside the allocated
                                 ; stack space for operand C
  ret void
}
```

Figure 6.2: An example of LLVM code representing an addition operation.

The code is based on the *Static Single Assignment (SSA)* representation, i.e. each variable of the program is assigned only a single time and every use of it must be previously declared. When a variable is defined, a new temporary and unique name is associated to it, limiting its scope until a re-definition of the same variable. This approach makes the implementation of optimizations on the code easier.

[1]The *Reduced Instruction Set Computing (RISC)* represents a highly-efficient set of instructions, each of which accomplishes a reduced amount of work.

The LLVM code, at the macro level, is structured in a hierarchical way and, more properly, as *Chinese boxes*. In particular, the program is composed of elements which in turn contain other ones. From the outermost, the code is formed by:

- Modules
- Functions
- Basic Blocks

Modules contain also *global variables* and *symbol table entries*, while functions are rich in various information. However, the program flow is organized in a set of basic blocks. The latter represents the *Control Flow Graph (CFG)*[2] of the function itself. In particular, a basic block consists of a collection of LLVM instructions that have to be executed in sequence and characterized by a single input point and a single exit point. Moreover, each basic block is identified and introduced through a label and it ends with a termination instruction (e.g. a return or a branch statement). Inside functions, the first basic block is considered "special", because it is the only one that is immediately executed at the entrance of the function and that is devoid of predecessor basic blocks.

6.2.2 The LLVM Core libraries

The libraries belonging to this category represent one of the most important support for a compiler developer. They provide a complete set of tools, which act on the LLVM Intermediate representation, useful for the implementation of both *optimizations* and *code generation* functions.

The LLVM Optimization Passes

The optimizations are organized in a chain of elements, called *Passes*, which receives at its input the IR code to return its optimized version[34]. The order though which they operate depends on the choices of the developer and on her/his purposes.

[2] A CFG is a representation, through a graph notation, of all the possible paths that can be traversed inside an algorithm.

The passes can be classified as follows:

- *Analysis Passes*: they collect advanced information from the input IR code, suitable for the subsequent optimization procedures, without introducing any modification. Examples of these passes are *Natural Loop information*, *Memory Dependence Analysis* and *Scalar Evolution Analysis*.
- *Transform Passes*: they mutate to various degree the program flow and, unlike the analysis ones, they don't produce any result in terms of information. The new processed IR code must be valid and equivalent to the original version that the pass has received at its input. Typically, after the elaboration, the previously performed analyses are invalidated and they are no longer made available. Examples of transform passes are *Dead Code Elimination (also its "aggressive version")*, *Promote Memory to Register*, *Simplify the CFG* and the whole set for Loop Transformations[3].
- *Utility Passes*: they provide general utility functions that can't be categorized neither into an analysis nor in a transformation context. An example of these passes is *View CFG of function*.

The LLVM tools for code generation

In support of back-end compiler designers, many libraries are provided with the target code generation. The latter are gathered to form, as defined inside the official documentation[35]:

> *"A suite of reusable components for translating the LLVM internal representation to the machine code for a specified target".*

The available classes encourage the development of efficient and quality code for standard register-based microprocessors. The compilation stages of the input IR code refer to the basic principles previously introduced in Section 3.1.2 while discussing about the typical back-end compiler structure.

[3]Among the most interesting: *Loop Invariant Code Motion*, *Delete dead loops*, *Loop Strength Reduction*, *Rotate Loops*, *Canonicalize natural loops* and *Unroll loops*.

In particular, the LLVM framework provides for:

- *Instruction selection*: the input IR code is translated into the target language, however, the registers remain expressed into the SSA-form. The information here collected are then mapped onto a *Directed Acyclic Graph (DAG)*.
- *Scheduling and Formation*: from the DAG generated during the previous phase of compiling, a list of ordered *instructions* is issued, following the strategies defined by the implemented scheduling algorithm.
- *SSA-based Machine Code Optimizations*: the scheduled instructions are optimized through specific algorithms.
- *Register Allocation*: during this phase all the SSA locations are substituted with the real registers belonging to the target microprocessor.
- *Prolog/Epilog Code Insertion*: for each function inside the algorithm the code for the related prologue and the epilogue is defined, since at this point of the compilation process the dimensions of the stack memory region are known. Other optimizations on the obtained code are applied.
- *Late Machine Code Optimizations*: target oriented optimizations are performed, like the *peephole*[4] one.
- *Code Emission*: the closing stage emits the final code intended for the proposed target machine.

Each component of the presented sequence can be modified at will by the compiler designer and as needed. The open-source nature of the available tools makes easy the definition of a code generator for new destination languages. The concept of reusability is an integral part of the LLVM project philosophy and typically few elements have to be specified in order to introduce a new back-end inside the LLVM system. An interesting guide for the design of a back-end of a generic CPU is reported inside the bibliography[36]. An additional useful reading about these arguments is the official tutorial, provided by the LLVM developer group[37].

The entire speech is valid under the assumption that the objective of the compilation fits into the LLVM machine description model. Unfortunately,

[4]The technique considers a sliding window of target instructions (i.e. the *peephole*) and it replaces, if possible, some of the latter with a shorter or a faster sequence.

generic hardware architectures obtained through High-Level Synthesis do not belong to this category and additional work is required for the implementation of such a back-end. However, the LLVM infrastructure represents a more than valid support during the design process of these particular compilers and many classes reveal profits to reduce the overall workload.

Clang

Clang is the reference front-end compiler, belonging to the LLVM project, for the generation of machine codes starting from C, C++ and Object-C input languages[38]. As previously discussed in Section 3, it represents one of the most popular and open-source compilation tools available. Clang supports also various parallel programming frameworks, among which the already mentioned OpenMP, OpenCL and CUDA.

It is defined as a *library-based architecture*, as the whole LLVM framework, allowing for an easy introduction of additional modules. The related classes can also be considered in the implementation of other front-end compilers. The resulting flexibility evidently becomes one of the most important values of the project.

As Clang represents a front-end, only an intermediate representation is obtained through the compilation process. Therefore, it relies on the available resources provided by the LLVM infrastructure to complete the translation into the target language or machine code.

A further interesting feature of its libraries consists in the *compatibility with GCC*. The same Clang developers underline how the latter is currently the *"defacto standard open-source compiler"* and including the support on its *extensions* makes Clang more "appealing" to the developers audience.

Polly

Polly is an LLVM framework for high-level loops and data-locality optimization, based on *integer polyhedra*[5]. Polly is capable to optimize loops through

[5]The more general concept of *Polyhedral compilation* is associated to the elaboration of programs, containing in this case loops and arrays, represented into a parametric polyhedra. The introduced optimizations on these structures are then transposed to the original algorithm.

transformations in order to improve the data-locality inside the code. The information about the target of the compilation process is considered to make even more effective the introduced improvements. In particular, it supports the OpenMP APIs to exploit the instruction level parallelism and the SIMD opportunities.

For a more detailed description of the Polly framework, please refer to the article [39] and to the official documentation[40].

Chapter 7

The structure of Octantis

Ocatantis is, for all intents and purposes, a compiler for Logic-in-Memory architectures. Is is written in C++ and to achieve its aims, the *LLVM Pass Framework* has been taken into consideration. Since the high-level description of the architecture is defined in C-code, also Clang has been included in the process of compilation. Specifically of this thesis work, some LLVM Passes have been written together with the back-end in order to obtain from an input code the configuration files for DExIMA. The general structure of the Octantis Project it depicted inside Figure 7.1.

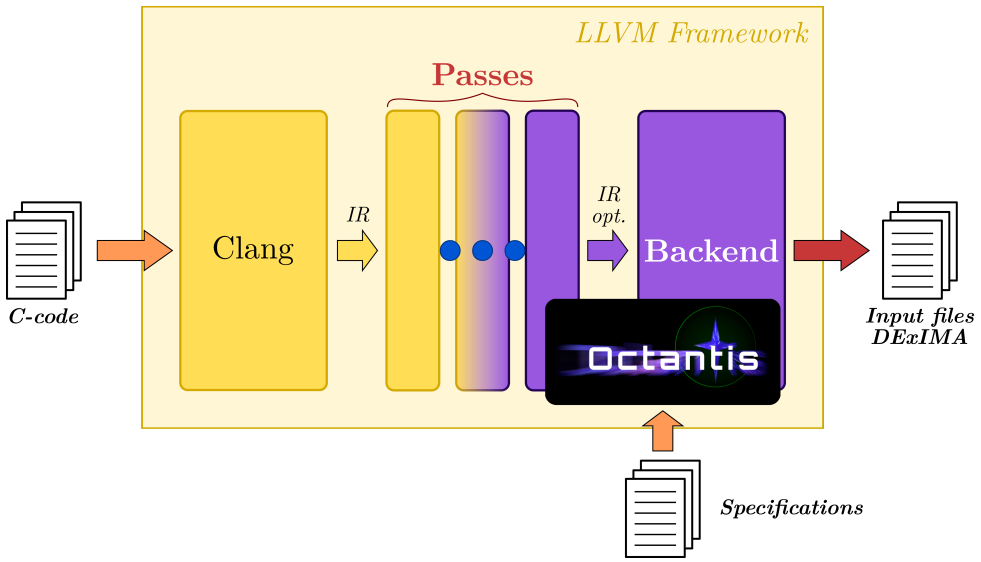

Figure 7.1: The conceptual framework of the Octantis Project.

Through different steps, the input code, expressed into the LLVM Intermediate Representation, is optimized and the operations that will be effectively

implemented inside a LiM architecture are selected. As previously discussed, all the instructions not recognized as useful for the definition of an appropriate solution are here discarded. Hence, the enhanced code is translated by the back-end pass into a file destined for the LiM simulator engine.

During the following sections, the implementation details will be discussed for each component belonging to the project. However, in order not to undermine the effectiveness of the present dissertation, the deepening of the not strictly necessary technicalities has been avoided. Together with the program, also a Doxygen documentation is provided where all the Octantis' code is reported with comments. Please, refer to it for additional insights.

7.1 From the input C-code to the LLVM IR

As mentioned above, Clang has been considered as the front-end compiler for the project. The input description of the LiM architecture is directly translated into the LLVM Intermediate Representation without bringing any optimization. The latter choice became necessary for a precise control on each modification introduced inside the original algorithm.

The current version of Octantis is capable of compiling *simple* architectures, constituted by regular arrays of LiM cells. Out-Of-Memory components are not considered, as they are not properly part of the LiM Unit. These conditions result in constraints in how a designer can approach to the program. The same input C-code is limited in its expressiveness and its syntax is oriented for a more "pragmatic" use, as with all High-Level Synthesis tools. The basic concepts of Logic-in-Memory computation, as intended by the *VLSI research group* at Politecnico di Torino[1], shall be clear to the designer.

7.1.1 Input C-code constraints

Octantis, as repeatedly pointed out, is an useful tool for the exploration of LiM architectures. The decision about the adoption of the C Standard for the description of the behavior of these hardware accelerators referred to abstract their definition at an higher level with respect to DExIMA's syntax

[1]Please refer to the description included in the first chapter of the present dissertation, Section 1.1.

(i.e. at Register Transfer Level). The choice allows to speed up the design phase, however, the fact that the developed algorithms regard an *hardware component* should never be overlooked. In fact, the users of Octantis would not be deceived by the great flexibility introduced by the use of a high-level language and they should be focused on the project aims: Octantis does not represent a programming tool, rather, a *hardware synthesizer*.

In order to become familiar with the program and to be aware of the general constraints, the following prescriptions are proposed:

1. All the data must be defined as integers As the LiM architecture is capable of executing arithmetic operations on integers, declaring and using floating-point data would not make sense.

2. The content of the declared variables can also be undefined The aim of the compiler is to produce an architecture and, in particular, it allocates LiM cells. Therefore, their content, in terms of information, is not relevant.

3. Dynamic allocation of memory and the consequent management are not allowed The algorithm has to be described bearing in mind the effective implementation. The needed hardware resources must be declared and they will be definitely integrated into the final architecture.

4. Recursive function calls are not allowed either If the flexibility of the C language to organize the execution of the code in this way was implemented, hardware complexities, that are not easily (in many cases possibly) optimized, would be introduced.

5. Among the possible arithmetic operations, multiplication and division should be avoided The hardware implementation for the execution of these operations is notoriously complex and resource consuming. Introducing their circuits inside a memory is currently inconvenient so, it is suggested to consider approximated calculations through *shift operations*.

6. All the bit-wise logic operations are implemented Even if some logic operations are not included in the syntax of Standard C, in particular the negative ones (i.e. nand, nor, xnor), they can be expressed anyway

through the composition of more bit-wise operators. Octantis will recognize the patterns.

As an additional tip, common sense should be adopted during the use of Octantis, considering its purposes in the *exploration* of LiM solutions. In particular, it represents a compiler for the implementation of *relatively simple* algorithms. With time, the Octantis project will grow up, together with the even more advanced capabilities of Logic-in-Memory architectures.

7.1.2 The definition of compilation constraints

In order to drive the synthesis process, Octantis receives in input also a configuration file. It is defined with *".cfg"* extension, an INI file format with a text-base content, typically used to store the programs information necessary for their proper execution. An example of the syntax is reported in Figure 7.2.

```
;*********************************************************************
;                  The Octantis Project - Configuration file
;*********************************************************************
;automatically generated at <time> <date> by <user>

[MemoryType]
;Constraints on the memory array
WordLength = 32
MaxDimension = 128KB

[SynthesisProcess]
;Constraints for the synthesys process
AdoptedOptimizations= [list of optimizations]
SchedulingAlgorithm = ASAP
OptimizedParameter = performance

[AdvancedLiMSettings]
;Explorable LiM solutions
ReferenceLiMArchitecture = default
```

Figure 7.2: Configuration file format.

State-of-the-art, only two parameters can be defined: the *word length*, which determines the dimension of the memory LiM rows, and the *adopted optimizations*. However, other ones have been set up for future expansions, among which:

- Scheduling algorithm
- Memory maximum dimensions
- Memory technology and typology
- Parameters to be optimized

The concept of *configuration* is essential for the aims of the program. Many implementations of Logic-in-Memory architectures have been proposed over the recent years and it would be advisable to gather them into categories of application. The latter should be associated to the class of algorithms they are best able to fulfil, allowing the designer to correctly *guide* the synthesis process.
Thanks to this information, Octantis will be able to provide highly optimized solutions for an input source code and, consequently, establishing itself as a reference tool for the design of innovative electronic systems.

7.2 The adopted optimizations

After the compilation of the C source code and the generation of the related Intermediate Representation by Clang, some optimizations are applied to the input algorithm in order to simplify it and, above all, to make the possibilities of parallelization explicit. In particular, before the execution of the synthesis process, different *passes* are run with this purpose, some of which have been cited along the previous chapter. They implement abstract enhancements, partly including the strategies detailed in Section 2.3. However, as has been discussed, the most effective techniques to promote an algorithm to parallel from a mathematical perspective, regard *loops*. Their importance has been highlighted when discussing both the software procedures to define parallel codes and the ones to increase the efficiency of high-level synthesis tools. For these reasons, during the definition of Octantis' modules, particular attention has been paid to *loops optimization*. Over the following sections, all the adopted methods to improve the quality of the input algorithm are deepened.

7.2.1 General optimization techniques

Many passes belonging to the LLVM framework can be considered for the optimization of an input algorithm. The ones that are made available to the designer of Logic-in-Memory architectures inside Octantis, are:

- *mem2reg Pass*
- *simplifycfg Pass*

Outside Octantis, also other passes can be considered to modify the IR code of the input algorithm. However, the compatibility with the LiM compiler is not ensured, differently from the presented cases.

mem2reg Pass

The pass promotes memory references with register ones. In particular, it tries to limit the operations that can be performed to the data stored inside the stack region to only *load* and *store* ones. Hence, the *Static Single Assignment* form of the code is straightened, making the subsequent compilation process easier.

simplifycfg Pass

The pass performs *dead code elimination* and the *merging of basic blocks* whenever possible, in order to simplify the algorithm and to produce a more efficient code.

7.2.2 Loop Analysis and Transformation

An important contribution to the increasing of performance in the implementation of an input algorithm regards loop optimizations. It represents the main structure through which to define parallel codes and many passes are provided by the LLVM framework to enhance them. The most relevant ones have been considered in the Octantis project and in particular:

- *licm Pass*
- *loop-deletion Pass*
- *loop-reduce Pass*
- *loop-simplify Pass*

Beside these, a new pass has been introduced to perform the *loop unrolling* operation, despite the availability of the *loop-unroll* pass. In fact, the

latter implements the transformation only when it holds the modification to be "convenient". If it was implemented, Octantis would have lost control over possible optimization opportunities for simple loops. For this reason, a custom pass has been developed.

licm Pass

The name of the pass is the acronym of *Loop Invariant Code Motion* and it tries to remove as many instructions as possible from the body of the loop. Among them, also the ones that require access to the stack memory region, promoting in this way the execution of the operations on the internal registers. Therefore, the resulting loops may be more compact and efficient, allowing the synthesizer to reduce the allocated resources for their implementation.

loop-deletion Pass

This pass executes the pruning of the input IR code in order to delete all the loops that do not take part to the computation of the final results. It results useful to generate a more efficient code for the following synthesis process.

loop-reduce Pass

The pass reduces the number of array references inside the loops and, in particular, the ones regarding the management of the variable used as index. Also this pass allows to make the code inside loops more compact, reducing the effort of the LiM compiler.

loop-simplify Pass

The pass, as its name suggests, is responsible to transform loops in simpler forms whenever possible. The derived benefits regard both the execution of the previous loop optimization passes and the synthesis process.

Loop unrolling

The technique represents the most righteous optimization element for the execution of loops. All clear of eventual data dependencies, whose introduction in the input code the designer has to avoid as much as possible, the pass tries to accomplish the different iterations belonging to the loop in parallel. The

number of resources required to implement the algorithm increases in favour of a reduction of the execution time.
Through this strategy, the parallel capabilities of Logic-in-Memory architectures can be best highlighted.

7.3 The back-end

Most of the work around the Octantis project has been focused on the definition of the back-end, the component which effectively translates the input algorithm from LLVM Intermediate Representation to an architecture described through the syntax of DExIMA files. The complexity of the labour and the will to refer to the modularity principle highly invoked by LLVM developers, led to a hierarchical organization of the code.

The general structure of Octantis' back-end is based on the traditional scheme of that of the compiler, the same one typically characterizing the High-Level Synthesis tools formerly examined in Section 4.2.1.

Many classes have been defined, each with a specific purpose. However, they all depend on a unique *pass*, named "OctantisPass", which coordinates them to properly lead the code during the whole conversion phases. OctantisPass acquires the input LLVM IR code, that has been previously optimized, and it proceeds to parse the present functions. For each of them, it reads the contained basic blocks, feeding the *scheduler* with the related instructions. During this process, both the *allocation* and the *scheduling* operations are performed. Then, a new data structure is organized to gather the derived results.

After completing these analyses of the entire input algorithm, the same pass commissions the *binder* to further elaborate the extracted information for the generation of:

- an *internal representation* of the described LiM architecture;
- a *finite state machine* useful in the organization of its control flow.

At the end, the *code emission* procedure is executed in order to define an appropriate formatted DExIMA configuration file. Inside the latter, both the description of the Logic-in-Memory solution and the needed instructions

for the dynamic simulation of the algorithms that it implements are included.

Each operation accomplished by OctantisPass and by the associated classes is discussed in the following sections, where the main focus remains on the considered algorithms.

7.3.1 The allocation

As previously mentioned, the allocation and the scheduling operations are executed in parallel. The reason behind this choice relies on the fact that the two algorithms need to exchange information in order to perform each its own task.

The current version of Octantis does not provide the possibility of imposing particular constraints on the synthesis process. Therefore, less resources have been employed during the definition of the allocation algorithm, consequently favouring the scheduling one. In particular, the *allocator* collects, from the input configuration file, the information about the word length for the LiM row. The latter then reveals useful during the *code generation* phase.

7.3.2 The scheduling

The input instructions provided by OctantisPass are firstly identified and categorized, according to the following classes:

- *allocation*
- *load*
- *store*
- *logic/binary operation*
- *branch*
- *control flow statement*
- *pointer*
- *return*

In the event that an *unknown* instruction, or a not yet supported one, is acquired, an error message is given back by Octantis in order to adequately inform the designer and the synthesis process is terminated.

Depending on their type, the instructions are then treated consequently and they are assigned a time slot in which they will be executed inside the final LiM architecture. In addition to their identification, during this synthesis step also the recognition of peculiar logic patterns is performed. In

particular, if eventual *negative logic* bit-wise operations are declared through a couple of operators (i.e. *not-and*, *not-or*, *not-xor*), they are substituted with their equivalent operators not directly supported by the standard C language (i.e. *nand*, *nor*, *xnor*).

Octantis implements an *As Soon As Possible (ASAP)* scheduling algorithm[2], whose details are discussed in the following section. At the end of the scheduling process, the instructions, enriched in these additional information, are stored inside a proper data structure, called *Instruction Table*.

The scheduling algorithm

In order to perform an accurate scheduling of the input instructions, their classification is essential. In particular, considering the structure and the properties of the LLVM IR code, it is remarkable that:

- any variable belonging to the program has to be *allocated* once inside the stack region of the memory;
- any time a variable is considered for a subsequent elaboration, a *load* operation is performed first, to store it inside an internal register, whose name follows the SSA convention;
- when a result has to be saved inside the stack, a *store* operation is executed.

As a Logic-in-Memory architecture is essentially a *memory*, there is no need for the implementation of a stack region. Consequently, none of these operations do make sense. However, they are useful to the scheduler to keep track of the evolution of information inside the algorithm and so to organize efficiently their computation.

In particular, *store* instructions are considered to know when an elaborated data is made available after its processing. They are of utmost importance to define the data dependencies present within the input algorithm and to identify the proper time intervals in which executing the following instructions. Instead, *load* operations are treated as they are: definitions of new variables inside the architecture, in this case the LiM array. The scheduler

[2]The ASAP scheduling algorithm, as the name suggests, states that any operation inside an algorithm will be executed *as soon as possible*.

assigns to them the briefest allocation time and it stores them directly inside the Instruction Table.

Considering all the other types of instructions, the ASAP scheduling algorithm becomes relevant. The various *data dependencies*, characterizing the information, are taken into account and the Instruction Table is filled accordingly. An example of the entire scheduling process, where the content of the generated data structure has been made clear, is depicted in Figure 7.3.

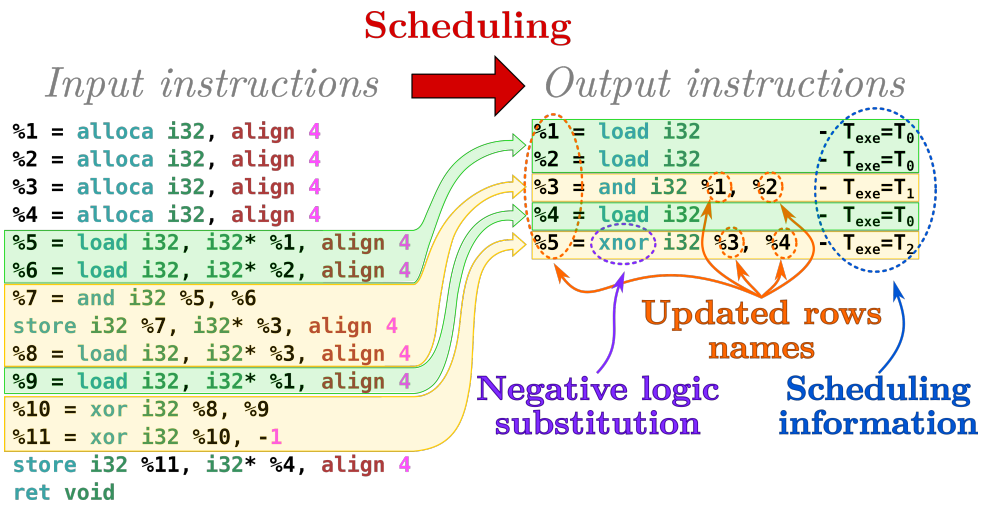

Figure 7.3: An example of the results derived from the application of the scheduling algorithm onto an input LLVM code; the performed operations by the code are: *A = (B and C), D = (A xnor B).*

7.3.3 The binding

After establishing the execution order of the different operations belonging to the IR code, the effective Logic-in-Memory architecture has to be arranged during the binding phase. In particular, the control and data flows are separately analyzed and two different data structures are derived: from the former, a *finite state machine*, while from the latter, a *LiM Unit*. The whole process is performed by *"LimCompiler"*, invoked, at the appropriate time, by OctantisPass.

However, before the deepening of the presented operations, the reference structure of a Logic-in-Memory Unit needs to be discussed.

The general Logic-in-Memory implementation structure

Octantis is given the possibility to compile architectures whose structure shall comply with the most recent researches about Logic-in-Memory carried out within the VLSI Laboratory at Politecnico di Torino. In particular, the basic scheme of the implementable devices is depicted in Figure 7.4.

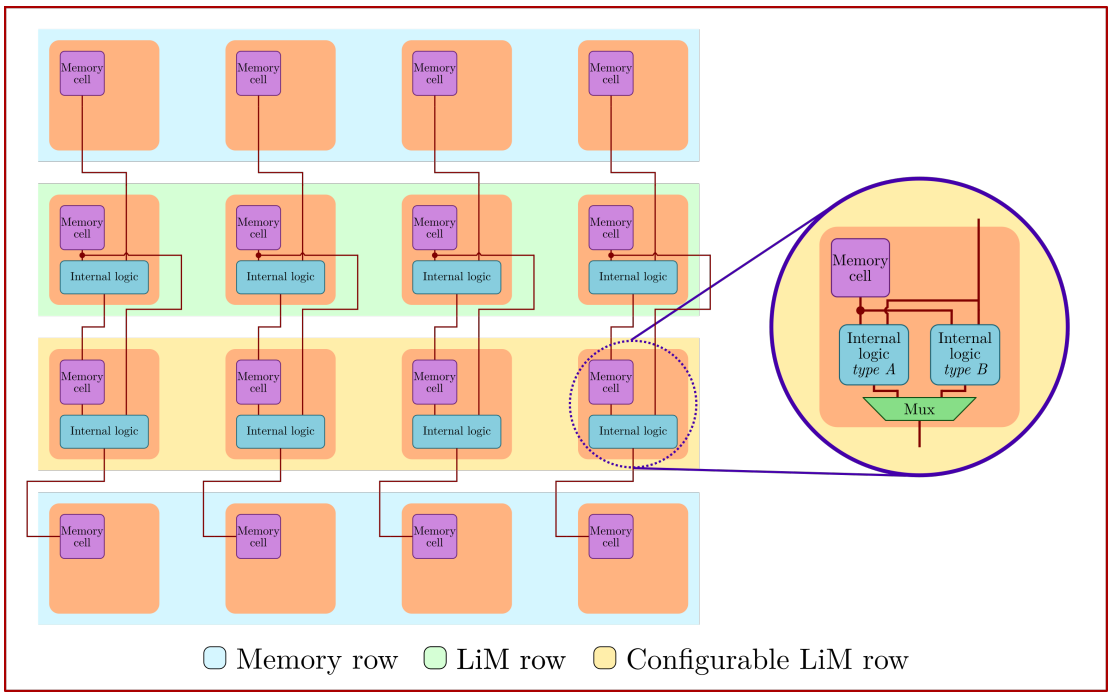

Figure 7.4: An example of the general structure of the LiM Unit that Octantis is able to synthesize. The output signals, like the control ones, are not represented to limit the complexity of the scheme.

As can be observed, the LiM array is regular and composed of rows with constant length. The cells belonging to a single row are uniform and characterized by the same internal logic, if any. In fact, not all the memory cells integrated inside a LiM Unit are enhanced with additional logic ports and it all depends on the specific input algorithm.

The internal organization of the LiM cells is the same that has been presented in Section 1.2.1 and, in particular, in Figure 1.2. Moreover, also configurable cells can be defined, referring to the stricture to the *CLIM-Architectures* briefly described in Section 1.1.

Therefore, form Octantis' point of view, the considered interfaces of a LiM row are: *two input connections* (one at the input of the memory cell and one at the input of the internal logic) and *one output connection*.

From an algorithmic perspective, each operation that has to be implemented requires two memory rows, for the input operands, and an additional one, for storing the result of the computation. Moreover, the cells representing one of the source row must include the needed logic ports to accomplish the same operation.

Regarding the interconnections between the different rows, maximum flexibility is guaranteed to the output connections. In this way, the information generated inside a LiM row can be considered for a subsequent elaboration though the integrated logic belonging to any other row[3]. Another argument applies to the incoming connections, as they are limited to a maximum of two inputs, without considering the access port to the memory cell. This decision is intended to keep the complexity of the array low and so, to maintain the performance of the Logic-in-Memory device adequate. In fact, the more logic ports are considered per unit of LiM cell, the more the *critical path* is and, consequently, performance reduces. Moreover, limiting the number of the input connections has, at least, two additional positive implications:

- The management of the interconnections themselves remains easy for many aspects, among which the data integrity, the related power consumption and the routing process.
- The control system of the LiM architecture is easier too, not having to handle abundant variable connections.

A final clarification should be made on how the LiM array is conceived. In particular, how the operations are implemented inside it. The structure is intended as a register-based one and each memory row belonging to these architectures represents a temporary register of a more complex computational device. Therefore, an external *control unit* is responsible for the correct timing of the registers and of the elements useful for proper routing of information (i.e. multiplexers). The algorithmic steps are then punctuated by a

[3]This is valid only for limited fan-outs and when the data locality principle is preserved. Otherwise, data should be replicated multiple times inside the memory array as far as the previous conditions are satisfied again.

timing signal, whose length is defined taking into account the *critical path* of the LiM Unit.

LiM architectures serve as a compromise between a memory device and an *hardware accelerator*. They flank a traditional processing unit (e.g. a CPU) in order to reduce the computational effort during the execution of data-intensive algorithms. Part of the overall elaborations are performed inside the same memory ad the derived intermediate results are made available to a processor.

The organization of the Control Flow

The information derived from the scheduling of the input operations are considered for the definition of a *Finite State Machine (FSM)*, necessary during the simulation phase of the synthesized architecture. The execution of the algorithm is organized in a series of discrete time intervals punctuated by a timing signal, as described above.

In particular, the active memory rows are declared for each time unit constituting the timeline. In this way the operating cells, during the various algorithmic steps, are possible to be known. The derived results are useful for estimating the *dynamic power consumption* of the designed LiM architecture, by means of DExIMA.

The organization of the needed hardware components

From the information present inside the Instruction Table, arranged by the scheduler, a virtual representation of the LiM array is organized. The different instructions are not directly mapped into LiM rows and some optimizations are applied to generate a memory with a more compact structure.
In particular, considering the generic organization of the LiM Unit previously described, the objective of this procedure consists in reducing, as much as possible, the number of memory rows devoid of internal logic. This results in a more effective use of the data in them contained.

The array is formed, row by row, during the analysis of the set of instructions. Since the latter is organized considering the execution order of the various operations, the mapping can proceed without particular difficulty. Again, the kind of instruction is discerned and an appropriate behavior is

expected for each of them. For sake of simplicity, they are gathered in two macro-classes: instructions for the *allocation* of memory rows, straightway associated to traditional storage cells devoid of any logic, and for the implementation of *logical and arithmetic operations*, which provides the execution of more complex tasks instead, detailed below.

First of all, the source operands of the instructions are considered in order to determine their type. In fact, for each elaboration of the data stored inside the LiM array, one of the two source rows must include the necessary logic to perform it. Different cases can be encountered and they will be described below together with the decisions that are taken on the definition of the array:

A. *One of the two source operands is enclosed inside a traditional memory row*

   ⇒ The binder inserts in the associated memory cells the needed logic ports for the implementation of the operation and it connects their free input to the other source operand.

B. *One of the two source operands is constituted by the same LiM cells required by the operation that has to be implemented*

   ⇒ Firstly, the binder checks the number of the input connections of the logic and, if the latter considers only one operand from another row[4], it introduces to them the additional connection with the cells belonging to the second operand row.

C. *All the other cases*

   ⇒ If both the source LiM rows are occupied by different logic ports with respect to the ones required by the considered instruction, the information stored inside one of the two operands is duplicated inside a new row where properly LiM cells are inserted. Therefore, the binder set the other operand as input for the logic.

Once the arrangement of the two rows of the source operands with the related connections has been carried out, the binder allocates an additional row

[4]Remember the imposed constraint on the number of input connections for the internal logic in order to reduce the complexity of the LiM array.

for storing the result of the computation. This new element of the array is composed of traditional memory cells, as they need only to acquire an input data. A proper connection is then defined to link them with the output ports of the source operand that embeds the logic useful for the execution of the current instruction.

In each of the presented cases the Finite State Machine is properly updated with the information useful to identify the active elements during the time interval assigned for the completion of the specific operation considered.

7.3.4 The code emission

The last part of the compilation process consists, as repeatedly pointed out, in the code emission. The information gathered during the previous elaboration steps is here summarized to generate the DExIMA configuration file. The latter describes both the synthesized LiM architecture and the instructions useful for its dynamic simulation.

The specific class with this aim is called *"PrintDexFile"*, however, its name should not mislead. In fact, the functions belonging to it do not only take the input information and print them out, but they also further elaborate them to produce a *consistent* result.

State-of-the-art, DExIMA configuration file is composed of many sections, some of which destined for the description and the simulation of Out-Of-Memory logic, others for the same purposes but considering Logic-in-Memory architectures. As Octantis produces at its output a LiM device which can be simulated, only the procedure for defining the structure of the latter component through DExIMA's syntax is discussed in the following. For a more detailed description about how to describe a configuration file for DExIMA simulator, please refer to the related documentation.

At the outset, an accurate description of the LiM array is performed. In particular, the memory structure is defined as a traditional RTL circuit. The first information that has to be declared is the *overall dimension* of the memory array. Subsequently, a characterization of the content of the different cells is performed. To do this, the kind of LiM cells has to be identified and the associated code sequence printed on the file. In parallel with the definition of the various elements composing the array, also the *interconnections* have

to be detailed, both *intra* and *inter cells*. It is interesting to note that they can vary depending on the specific logic integrated inside the rows. A prime example of this different organization of the interconnections is represented by the comparison between rows implementing bit-wise operations and rows intended to perform additions[5]. In fact, while the former is composed of cells which lead the information in a "vertical" way, the latter needs to propagate the carry bit horizontally and additional interconnections have to be defined to make this propagation possible, as depicted in Figure 7.5.

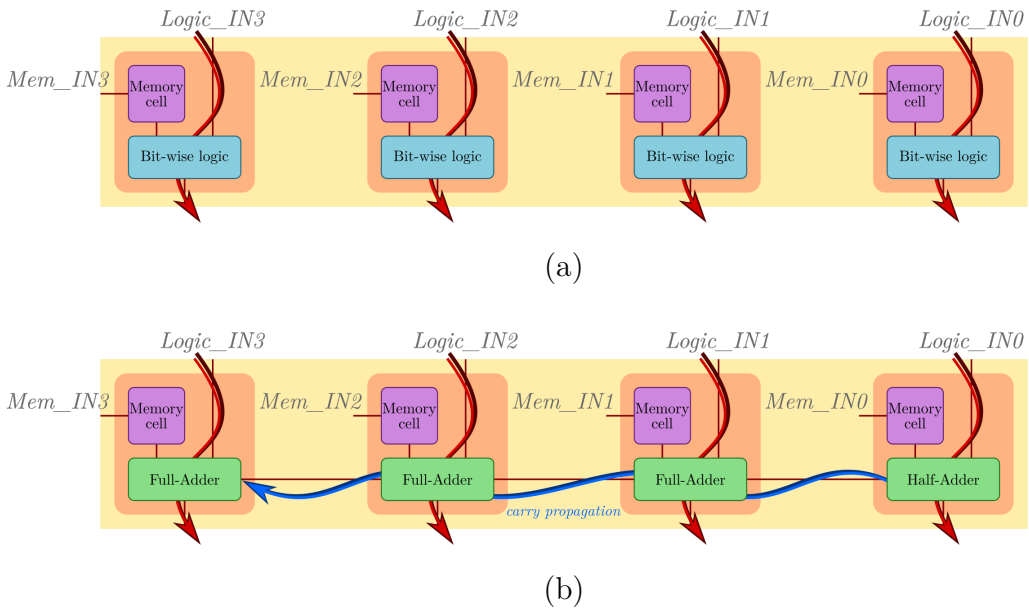

Figure 7.5: Particular of the characteristic interconnections of two LiM rows performing a bit-wise operation (a) and an addition (b).

During the definition of the interconnections, also the needed *multiplexers* have to be properly described. They are inserted whenever the logic inside a memory cell owns two different inputs or they are configurable.

It is important to highlight that the current version of DExIMA does not allow a simulation considering the variation of control signals. The performed simulation is, in fact, *virtual* and it requests the user to declare, for each time interval, which cells are active in order to compute an estimate of the dynamic power consumption of the designed LiM architecture. As the details on the performed simulation are under study, at the time of writing, it has

[5]The architecture embedded in the Logic-in-Memory array for the execution of the addition is the *Ripple Carry Adder* one.

been decided not to print this section of the configuration file. In its place, for an easy debugging of the solutions generated by Octantis, a list of the active rows per unit of time are printed out. The required information is collected considering the Finite State Machine previously organized by the binder.

Examples of the code produced by Octantis are reported in the Appendix, while their description is discusses in the following chapter. There, some test cases, inclusive of the obtained results, are discussed.

Chapter 8

Test on Octantis

The effective functionality of Octantis has been proved through the analysis of the results deriving from the synthesis process of many input algorithms. Among them, the most significant ones have considered some of the latest research works on Logic-in-Memory architectures, carried on by the researches of the VLSI Laboratory. In particular, two of them have been discussed during the opening chapter of the present dissertation in Section 1.1, when the LiM concept has been introduced for the first time. They are the *CLiMA* units implemented for the definition of a *Convolutional Neural Network (CNN)*[1] and for the implementation of the *Bitmap Indexing Algorithm*[2]. In addition to these works, another one has been taken into account which considers again a CNN architecture but it is based on a *binary approximation* of the convolution operation[41].

The synthesis process of the cited works is described in the following along with the derived results. Since an important *reconstructive work* on the simulator engine DExIMA is underway, many of its functionalities are not available. Hence, the results discussed along the following pages have been tested completely *by-hand*. The verification procedure has not turned out to be particularly complicated, due to the sufficiently abstract constitution of the derived Logic-in-Memory architecture (i.e. *Register Transfer Level*).

8.1 Synthesis of the XNor Net for an approximated CNN

The considered work focused on the implementation of a *Binarized Convolutional Neural Network*, an artificial neural network, for the execution of algorithms suitable for pattern recognition and classification. The process consists in the *discretization* of an input information (e.g. an image as a composition of pixels) and a subsequent *convolution*, in order to extract the needed features. These results are then properly classified to provide the external user with the investigated parameters (e.g. the outcome of the analysis of the input image).

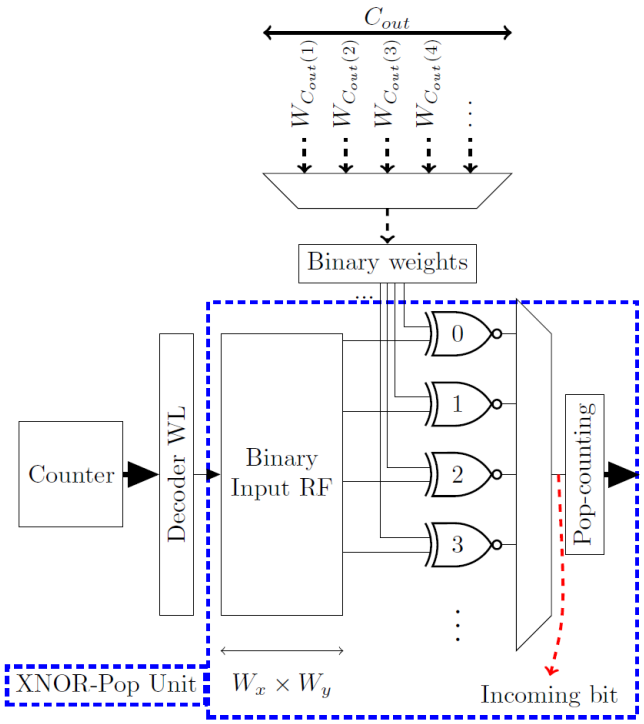

Figure 8.1: The LiM array implementing the XNor Net. *Courtesy of Andrea Coluccio from Article [41]; p. 10, Figure 7.*

The Logic-in-Memory array has been specifically adopted to improve the efficiency of the convolution. Traditionally, this operation is performed through a series of multiplications and accumulations. However, as the actual Logic-in-Memory architectures are effective in the execution of simple logic functions, the multiplications have been substituted by logic operations. The

introduced approximation on the output results gives the implemented solution the epithet of *binarized*. In particular, both the input data and the relative weights have been *discretized* and two hardware units have become necessary: a *XNOR Net*, for the execution of the binary multiplication of the data, followed by a *Pop-counter*[1], for their accumulation. The specific architecture proposed by the cited work is depicted in Figure 8.1.

Since the Pop-counter is actually an Out-Of-Memory circuit, it has not been described in the input C algorithm for the synthesis of the Logic-in-Memory architecture. Thus, the latter has come down to a simple array of XNor LiM cells. The Octantis' configuration file has been set up to produce a final memory with a word length of five bits, the same dimension declared inside the article and the input algorithm, illustrated in Figure 8.2, has been defined.

```
1  //Code for the implementation of a XNor Net
2  void XNor_Net(){
3
4      //Allocation of the weight
5          unsigned weight;
6      //Allocation of the matrix for the input data
7          unsigned dataMatrix[5];
8      //Allocation of the rows for the output results
9          unsigned outData[5];
10
11     //Execution of the Xor operations on the data
12     for(int i=0;i<4;++i)
13                 outData[i]=~(weight^dataMatrix[i]);
14
15 }
```

Figure 8.2: Tested input code for the LiM implementation of the *XNOR Net*.

The deriving architecture has revealed fully comparable and equivalent to the solution proposed inside the article, so as the timing information. The

[1]A Pop-counter is a hardware unit useful to determine the difference between the number of ones and the number of zeros constituting an input data.

synthesis process required few milliseconds to complete, showing also the effectiveness of Octantis as an agile exploration tool.

The obtained results have been gathered and proposed in a *readable-friendly* format inside Appendix A, for completeness.

8.2 Synthesis of a Bitmap Indexing algorithm implementation on CLiMA

The Bitmap Indexing algorithm represents a valid approach in the classification of information coming from different sources in parallel. The proposed solution for its implementation is a configurable Logic-in-Memory architecture inspired by the concept of CLiM-Arrays. The algorithm can operate two kinds of operations on the data stored inside the LiM array:

- $Out = A \ and \ B$
- $Out = A \ and \ (B \ or \ C)$

where A, B and C serve as a typology of information. Also their negated values are made available inside each LiM cell.

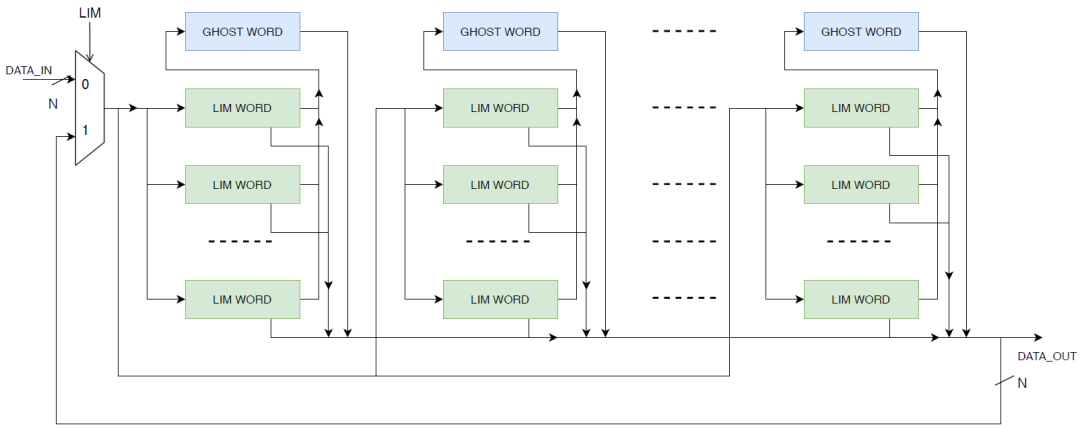

Figure 8.3: The CLiM-Array implementing the Bitmap Indexing algorithm. *Courtesy of Milena Andrighetti from her M.Sc. Thesis [42]; p. 78, Figure 3.12.*

The overall circuit is complex, especially as regards the control and the techniques useful to the routing of information, all external to the actual

Logic-in-Memory array. Therefore, the current synthesis test has considered only the sub-portion of the project which is more suitable to the process. In particular, the architecture depicted in Figure 8.3 has been taken into account.

The present test case is of particular interest because it emphasizes the breadth of Octantis' scope, in particular for the range of abstraction it is able to manage. While in the previous example a completely high-level description has been considered for the definition of the logic-in-Memory array, here the *elevation* is reduced and a *behavioral description* has to be adopted[2]. This kind of design reveals essential to describe a configurable logic as it needs a *granularity* of detail that otherwise could not be achieved. In fact, each LiM cell is composed of the components depicted in Figure 8.4.

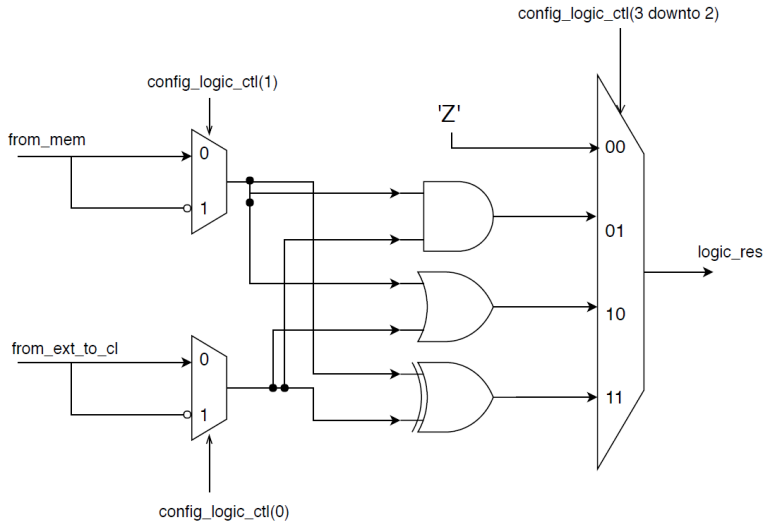

Figure 8.4: Details of the logic integrated inside the CLiM cells. *Courtesy of Milena Andrighetti from her M.Sc. Thesis [42]; p. 74, Figure 3.8.*

In particular, as the architecture implemented is regular, only one "column" of the the entire array has been synthesized, also devoid of the input multiplexer since it belongs to the Out-Of-Memory logic. Therefore, the word length of the memory has been imposed to 16-bits inside Octantis' configuration file, as the reference architecture, and the code describing the architecture has been defined. The latter is reported in Figure 8.5.

[2]It should be recalled that the behavioral synthesis is an ancestor of modern high-level synthesizers and that it is partially integrated inside the available EDA tools.

```
//Code for the implementation of the Bitmap Indexing algorithm
//  NOTE: the return statement is the common output line while
//        the passed parameter represents the common input line
int BitmapIndex (int commonInLine){

    //Allocation of the memory "ghost" row
    int ghostRow;
    //Definition of the configuration signal for CLiM cells
    int configuration;
    //Allocation of the set of data
    int sourceRow[8];

    for(int i=0; i<8; i++){

        //Definition of the configurable logic inside each CLiM cell
        switch(configuration){
                        case 0:
                            ghostRow='Z';
                            return sourceRow[i];
                        case 1:
                            ghostRow=sourceRow[i] & commonInLine;
                            return sourceRow[i];
                        case 2:
                            ghostRow=sourceRow[i] | commonInLine;
                            return sourceRow[i];
                        case 3:
                            ghostRow=sourceRow[i] ^ commonInLine;
                            return sourceRow[i];
                        default:
                            break;
                }
        }
    //Definition of the output connection
        return ghostRow;
}
```

Figure 8.5: Tested input code for the implementation of the *Bitmap Indexing algorithm* through a CLIM-Architecture.

Also in this case, the results of the synthesis process have revealed in compliance with the original architecture proposed by the author of the article. However, it should be clarified that the information obtained for the simulation of the architecture is meaningless. A drawback of behavioral synthesis consists in the loss of details about the algorithm an architecture has to implement. As for a traditional RTL design, a test-bench has to be arranged in order to verify the correct behavior of the developed solution, also in this context the implemented algorithm should be declared to Octantis. Only in this way, the synthesizer can produce also the needed information for a proper simulation of the generated device. State-of-the-art, this feature has not been implemented yet but it will be in the next future.

A copy of the results obtained through Octantis, in the implementation of the discussed algorithm, is attached in Appendix B.

8.3 Synthesis of a CLiMA CNN

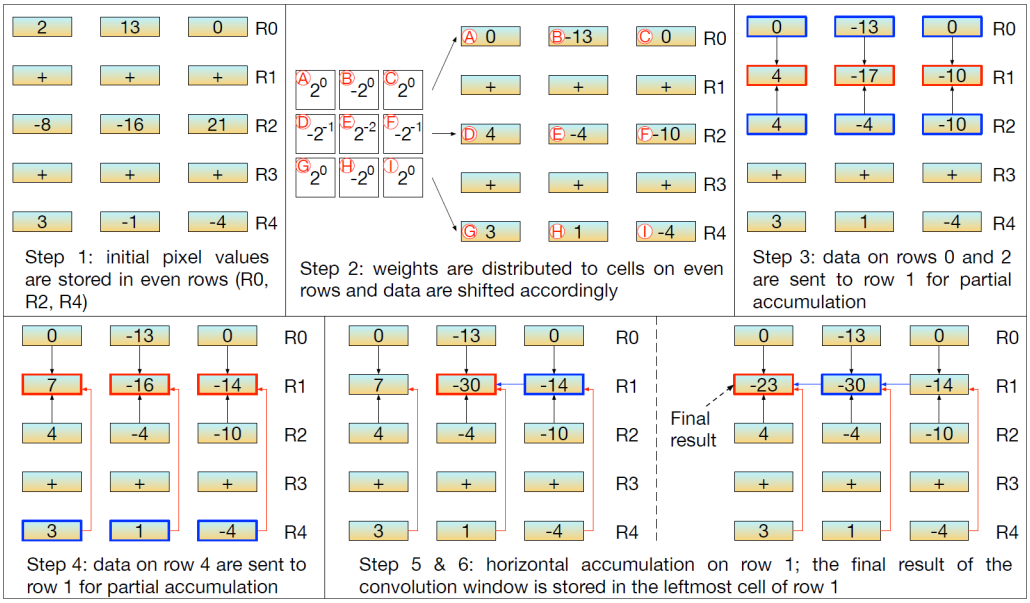

Figure 8.6: The process of convolution in the CLiM-Array. *Courtesy of Giulia Santoro from Article [1]; p. 12, Figure 13.*

The cited work consisted in the implementation of another *Quantized Convolutional Neural Network*. However, in this case the *discretization* of the information does not affect anymore the data and the weights but only the

results. In fact, the convolution has been obtained through the approximation of the multiplications in shift operations and a subsequent phase of accumulation.

A CLiM-Array has been considered for the implementation of the designed neural network and the derived architecture, proposed by the article, is detailed in Figure 8.6. Inside the same figure, also a description of the applied algorithm is given by the author.

The synthesized circuit refers precisely to the array barely mentioned, with an additional adjustments. Firstly, the array is disposed following a vector configuration. In fact, DExIMA is able to test Logic-in-Memory architectures composed of uniform and continuous rows. Future implementations of the simulator will support also these more complex structures, surely.

The configuration file for Octantis has been set up so as to impose a word length of 8-bits and the input code, depicted in Figure 8.7, has been defined.

The produced architecture has resulted fully equivalent from an algorithmic point of view. However, it has an *hardware overhead* which, although limited, makes the obtained solution different from the reference one. These variations are quantified inside Table 8.1.

Integrated Logic	Original implementation	Octantis' solution
Shift	9	9
RCA	6	8

Table 8.1: Overhead of the architecture produced by Octantis over the reference one.

As can be observed, the responsibility for the increment of the needed hardware for the implementation of the code is to be attributed to the operation of *accumulation*. In fact, Octantis relies on a specific algorithm to schedule this kind of repetitive tasks which operates following a binary tree like flow in order to reduce, step by step, the input operands. At every step of the computation, additional adders are allocated to generate partial results and the re-usability of these components is not considered. The accumulation so defined is faster than a serial execution, but resource consuming.

```
//Code for the implementation of the CNN algorithm
void CNN(){

    //Allocation of the LiM rows for the pixels
    int pixels[9];
    //Allocation of the map for the input weights
    int weights[9];
    //Allocation of the vector for the partial results
    int partial[9];
    //Allocation of the row for the result of accumulation
    int result=0;

    //Definition of the shift operations and computation
    //of the partial results
    for(int i=0; i<9; ++i)
        partial[i]=pixels[i] >> weights[i];

    //Accumulation of the generated partial results
    for(int j=0; j<9; ++j)
        result+=partial[i];
}
```

Figure 8.7: Tested input code for the implementation of a *Convolutional Neural Network* through a CLIM-Architecture.

The decision to consider such an algorithm lies on the principle of reducing the overall complexity. Firstly, it is important to note that the operation of accumulation could be executed in parallel with other ones and managing limited resources with the allocation of different operations represents an optimum problem, not so easy to resolve. Secondly, both the interconnections and the control become even more complicated.

Nevertheless, the need for more advanced optimizations during the synthesis process becomes clear and more sophisticated algorithms could be adopted in the future in that direction.

In conclusion, the detailed results provided by Octantis have been gathered inside Appendix C. Also in this case, the information about the timing

of the architecture is meaningless since the shift operations are applied by the control as a sequence of one-right-shift per unit of time. The number of shift operations varies depending on the input data and, without any additional details on the simulation procedure, Octantis is not able to provide accurate test-benches. As previously discussed, the compatibility of Octantis to perform simulations with customized input test vectors will be implemented within future updates.

Chapter 9

Conclusions and future works

Octantis presents itself as an agile tool in support of the exploration and the design of Logic-in-Memory architectures. As repeatedly pointed out, the aim of this first version of the program is to provide a designer with quick information on how an algorithm can benefit from the implementation on such an architecture.

For all intents and purposes, Octantis represents an High-Level Synthesizer and it receives in input an abstract algorithm, described through one of the most diffused programming languages, to define a proper architectural solution. This high-level approach during the design phase allows a designer to focus on the quality and the efficiency of an algorithm, without being distracted by the many implementation details and technicalities that only a Logic-in-Memory designer should really know. Thereby enabling an easy access to an electronic designer to LiM world is ensured.

Therefore, the main advantages of Octantis can be summarized as follows. It's fast during the synthesis process, embodying the purpose of the exploration of alternative implementation solutions. The whole project has been designed so as to guarantee the *modularity* and the *maintenance* of the code. For what concerns the maintenance, a detailed documentation is provided together with Octantis in order to prepare the designer to work on it in a short time. The modularity is then guaranteed to allow the same developers to extend the capability of the tool during time. Moreover, it is integrated

inside a *lively* development framework, the LLVM one, which is firstly open-source and that it is rich in tools and algorithms for the adoption of new and advanced functionalities. Last but not least, Octantis is capable of managing both the definition of an *hardware architecture* and the *relative control flow*, starting from simple and self-consistent[1] input algorithms. Hence, also a test-bench for the proposed solution is generated.

State-of-the-art, many other features can be introduced, in the view of an increasing complexity and completeness of the program. First of all, the definition of more advanced optimization techniques and that are able to overstep the actual *ASAP scheduling algorithm*. Then, the introduction of support for more accurate test-benches that could accept test vectors in input to properly simulate the produced Logic-in-Memory unit. Finally, wider customization properties inside the configuration file can be developed so that they may lead the synthesis process for the generation of more specific and optimized architectures. Time of writing, many general purpose Logic-in-Memory devices are under evolution and which are reaching the needed maturity to establish themselves as a reference for specific algorithmic applications. The latter may be supported by Octantis, in its future versions.

An additional future work goal could be the analysis of a generic C source code to define, directly when Octantis starts, which parts of the whole algorithm could benefit of a Logic-in-Memory implementation and which would not. In this way, only a portion of the source code would be mapped on a Logic-in-Memory architecture while the remaining part could be rearranged to be provided to a generic processor back-end. This new version of Octantis would be able to collect information on how a more complex system, composed of a LiM unit associated to a modern processor, would behave.

These are few of the possible future works which could be realized around Octantis.

In conclusion, the hope is to have realized an instrument capable of giving light to the many Logic-In-Memory researches that have been carried out so far by the researchers of the VLSI Laboratory. Octantis aims to be a *valid guide* during the exploratory phase of LiM architectures, a promising solution for how electronics could evolve in the coming years.

[1]Self-consistent code is defined as an algorithm which proceeds by design in its execution, without the need of any external information, like test-vectors.

Appendix A

XNor Net on LiM

In the following the results of the synthesis process on the *XNor Net* have been gathered. The information associated to the scheduling time assumes that the content of the Logic-in-Memory unit has been initialized before the execution of any operation.

Row Number	Address	Word Length	Integrated Logic	Input Connections
1	000	5	—	—
2	001	5	XNor	0000
3	010	5	XNor	0000
4	011	5	XNor	0000
5	100	5	XNor	0000
6	101	5	XNor	0000

Table A.1: Information about the LiM structure.

Row Number	Address	Scheduling Time
1	000	0
2	001	0
3	010	0
4	011	0
5	100	0
6	101	0

Table A.2: Information about the scheduling of the operations.

Appendix B

Bitmap Indexing algorithm on CLiMA

In the following the results of the synthesis process of architecture implementing the Bitmap Indexing have been gathered. Since the input test vectors can't be provided to Octantis in order to let it organize a proper simulation, the information associated to the scheduling time is meaningless and it is not reported.

Row Number	Address	Word Length	Integrated Logic
1	0000	16	—
2	0001	16	And/Or/Xor
3	0010	16	And/Or/Xor
4	0011	16	And/Or/Xor
5	0100	16	And/Or/Xor
6	0101	16	And/Or/Xor
7	0110	16	And/Or/Xor
8	0111	16	And/Or/Xor
9	1000	16	And/Or/Xor

Table B.1: Information about the LiM structure *(Part 1)*.

Row Number	Address	Input Connections	Additional Notes
1	0000	All the other lines	—
2	0001	Input Line	Configurable logic
3	0010	Input Line	Configurable logic
4	0011	Input Line	Configurable logic
5	0100	Input Line	Configurable logic
6	0101	Input Line	Configurable logic
7	0110	Input Line	Configurable logic
8	0111	Input Line	Configurable logic
9	1000	Input Line	Configurable logic

Table B.2: Information about the LiM structure *(Part 2)*.

Appendix C

CLiMA CNN

In the following the results of the synthesis process of architecture implementing the Quantized CNN have been gathered. Since the input test vectors can't be provided to Octantis in order to let it organize a proper simulation, the information associated to the scheduling time is meaningless and it is not reported.

Row Number	Address	Word Length	Integrated Logic	Input Connections	Additional Notes
1	00000	8	—	—	Internal shift operation
2	00001	8	Full/Half-Adder	00000	Internal shift operation
3	00010	8	—	00001	—
4	00011	8	—	—	Internal shift operation
5	00100	8	Full/Half-Adder	00011	Internal shift operation
6	00101	8	Full/Half-Adder	00100, 00010	—
7	00110	8	—	—	—
8	00111	8	—	—	Internal shift operation
9	01000	8	Full/Half-Adder	00111	Internal shift operation
10	01001	8	—	01000	—
11	01010	8	—	—	Internal shift operation
12	01011	8	Full/Half-Adder	01010	Internal shift operation
13	01100	8	Full/Half-Adder	01011, 01001	—
14	01101	8	Full/Half-Adder	01100, 00110	—
15	01110	8	—	01101	—
16	01111	8	Full/Half-Adder	01110	Internal shift operation
17	10000	8	—	01111	—

Table C.1: Information about the LiM structure

Nomenclature

ADG Algorithm Dependence Graph

API Application Programming Interface

ASAP As Soon As Possible

ASIC Application Specific Integrated Circuit

CFG Control Flow Graph

CLiMA Configurable Logic-in-Memory Array

CMOS Complementary Metal-Oxide Semiconductor

CNN Convolutional Neural Network

CPU Central Processing Unit

DAG Directed Acyclic Graph

DSP Digital Signal Processor

EDA Electronic Design Automation

ESL Electronic System-Level

FPGA Field Programmable Gate Array

FSM Finite State Machine

GPGPU General Purpose Graphical Processing Unit

GPU Graphical Processing Unit

HDL Hardware Description Language

HLS High-Level Synthesis

IoT Internet of Things

IP Intellectual Property

IR Intermediate Representation

LiM Logic-in-Memory

NN Neural Network

$RISC$ Reduced Instruction Set Computing

RTL Register Transfer Level

$SIMD$ Single Instruction Multiple Data

SSA Static Single Assignment

Bibliography

[1] Giulia Santoro, Giovanna Turvani, and Mariagrazia Graziano. "New Logic-In-Memory Paradigms: An Architectural and Technological Perspective". In: *Micromachines* 10.6 (May 2019), p. 368. ISSN: 2072-666X. DOI: 10.3390/mi10060368.

[2] Milena Andrighetti et al. "Data Processing and Information Classification—An In-Memory Approach". In: *Sensors* 20.6 (Mar. 2020), p. 1681. ISSN: 1424-8220. DOI: 10.3390/s20061681.

[3] N. Piano. "DExIMA: a Design Explorer for In-Memory Architectures". MA thesis. Politecnico di Torino, 2019. URL: https://webthesis.biblio.polito.it/12547/.

[4] F. Riente et al. "Understanding CMOS Technology Through TAM-TAMS Web". In: *IEEE Transactions on Emerging Topics in Computing* 4.3 (2016), pp. 392–403. ISSN: 2168-6750. DOI: 10.1109/TETC.2015.2488899.

[5] Shyamkumar Thoziyoor et al. *CACTI 5.1*. Technical Report. Palo Alto: HP Laboratories, 2008. URL: https://www.hpl.hp.com/techreports/2008/HPL-2008-20.html.

[6] M. Niemier and al. "Nanomagnet logic: progress toward system-level integration". In: *J. Phys.: Condens. Matter* 23 (Nov. 2011), p. 34. DOI: 10.1088/0953-8984.

[7] G. Turvani et al. "A pNML Compact Model Enabling the Exploration of Three-Dimensional Architectures". In: *IEEE Transactions on Nanotechnology* 16.3 (May 2017), pp. 431–438. ISSN: 1941-0085. DOI: 10.1109/TNANO.2017.2657822.

[8] *2018 Edition of International Roadmap for Devices and Systems (IRDS)*. 2018. URL: https://irds.ieee.org/editions/2018.

[9] Fayez Gebali. *Algorithms and Parallel Computing*. John Wiley & Sons Inc, 2011. ISBN: 9780470902103.

[10] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach*. 6^{th}. Morgan Kaufmann, Nov. 2017, p. 936. ISBN: 9780128119051.

[11] M. D. Hill and M. R. Marty. "Amdahl's Law in the Multicore Era". In: *Computer* 41.7 (July 2008), pp. 33–38. ISSN: 1558-0814. DOI: 10.1109/MC.2008.209.

[12] J. E. Volder. "The CORDIC Trigonometric Computing Technique". In: *IRE Transactions on Electronic Computers* EC-8.3 (Sept. 1959), pp. 330–334. ISSN: 0367-9950. DOI: 10.1109/TEC.1959.5222693.

[13] S. W. Keckler et al. "GPUs and the Future of Parallel Computing". In: *IEEE Micro* 31.5 (Sept. 2011), pp. 7–17. ISSN: 1937-4143. DOI: 10.1109/MM.2011.89.

[14] OpenMP ARB, ed. *OpenMP Application Programming Interface*. v. 5.0. 2018. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

[15] Khronos Group, ed. *OpenCL 3.0 Reference Guide*. v. 3.0. 2020. URL: https://www.khronos.org/files/opencl30-reference-guide.pdf.

[16] Advanced Micro Devices, ed. *Introduction to OpenCL Programming*. 2010. URL: http://developer.amd.com/wordpress/media/2013/01/Introduction_to_OpenCL_Programming-Training_Guide-201005.pdf.

[17] Nvidia, ed. *CUDA C++ Programming Guide*. 2020. URL: https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf.

[18] M. J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* C-21.9 (Sept. 1972), pp. 948–960. ISSN: 1557-9956. DOI: 10.1109/TC.1972.5009071.

[19] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[20] Kasliwal Vishal and Vladimirov Andrey. *A Performance-Based Comparison of C/C++ Compilers*. Tech. rep. Colfax International, Nov. 2017. URL: https://colfaxresearch.com/compiler-comparison/.

[21] A. Sangiovanni-Vincentelli. "The Tides of EDA". In: *IEEE Design Test of Computers* 20.06 (Nov. 2003), pp. 59–75. ISSN: 1558-1918. DOI: 10.1109/MDT.2003.1246165.

[22] G. Martin and G. Smith. "High-Level Synthesis: Past, Present, and Future". In: *IEEE Design Test of Computers* 26.4 (July 2009), pp. 18–25. ISSN: 1558-1918. DOI: 10.1109/MDT.2009.83.

[23] J. J. Rodríguez-Andina, M. D. Valdés-Peña, and M. J. Moure. "Advanced Features and Industrial Applications of FPGAs—A Review". In: *IEEE Transactions on Industrial Informatics* 11.4 (Aug. 2015), pp. 853–864. ISSN: 1941-0050. DOI: 10.1109/TII.2015.2431223.

[24] S. M. Trimberger. "Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology". In: *Proceedings of the IEEE* 103.3 (Mar. 2015), pp. 318–331. ISSN: 1558-2256. DOI: 10.1109/JPROC.2015.2392104.

[25] A. Takach. "High-Level Synthesis: Status, Trends, and Future Directions". In: *IEEE Design Test* 33.3 (June 2016), pp. 116–124. ISSN: 2168-2364. DOI: 10.1109/MDAT.2016.2544850.

[26] S. Lahti et al. "Are We There Yet? A Study on the State of High-Level Synthesis". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.5 (May 2019), pp. 898–911. ISSN: 1937-4151. DOI: 10.1109/TCAD.2018.2834439.

[27] Yuanbin Guo et al. "Rapid Industrial Prototyping and SoC Design of 3G/4G Wireless Systems Using an HLS Methodology". In: *EURASIP Journal on Embedded Systems* 2006 (Jan. 2006). DOI: 10.1155/ES/2006/14952.

[28] J. Cong et al. "High-Level Synthesis for FPGAs: From Prototyping to Deployment". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.4 (Apr. 2011), pp. 473–491. ISSN: 1937-4151. DOI: 10.1109/TCAD.2011.2110592.

[29] R. Nane et al. "A Survey and Evaluation of FPGA High-Level Synthesis Tools". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.10 (Oct. 2016), pp. 1591–1604. ISSN: 1937-4151. DOI: 10.1109/TCAD.2015.2513673.

[30] Andrew Canis et al. "LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems". In: *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, CA, USA: Association for Computing Machinery, 2011, pp. 33–36. ISBN: 9781450305549. DOI: 10.1145/1950413.1950423.

[31] C. Pilato and F. Ferrandi. "Bambu: A modular framework for the high level synthesis of memory-intensive applications". In: *2013 23rd International Conference on Field programmable Logic and Applications*. Sept. 2013, pp. 1–4. DOI: 10.1109/FPL.2013.6645550.

[32] Bruno Cardoso Lopes and Rafael Auler. *Getting Started with LLVM Core Libraries*. Packt Publishing, 2014. ISBN: 1782166920.

[33] LLVM Developer Group, ed. *LLVM Language Reference Manual*. URL: https://llvm.org/docs/LangRef.html.

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

[35] LLVM Developer Group, ed. *The LLVM Target-Independent Code Generator*. URL: https://llvm.org/docs/CodeGenerator.html.

[36] Chen Chung-Shu. *Tutorial: Creating an LLVM Backend for the Cpu0 Architecture*. Release 3.9.1. May 2020. URL: https://jonathan2251.github.io/lbd/.

[37] LLVM Developer Group, ed. *Writing an LLVM Backend*. URL: https://llvm.org/docs/WritingAnLLVMBackend.html.

[38] LLVM Developer Group, ed. *Clang: a C language family frontend for LLVM*. URL: https://clang.llvm.org/index.html.

[39] Louis-Noël Pouchet et al. "Polly-polyhedral optimization in LLVM". In: vol. 2011. Jan. 2011.

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

[41] Andrea Coluccio, Marco Vacca, and Giovanna Turvani. "Logic-in-Memory Computation: Is It Worth It? A Binary Neural Network Case Study". In: *Journal of Low Power Electronics and Applications* 10.1 (Feb. 2020), p. 7. ISSN: 2079-9268. DOI: 10.3390/jlpea10010007.

[42] M. Andrighetti. "Parallel architectures for Processing-in-Memory". MA thesis. Politecnico di Torino, 2018. URL: https://webthesis.biblio.polito.it/9501/.