Received 26 July 2024, accepted 22 August 2024, date of publication 26 August 2024, date of current version 5 September 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3450105

RESEARCH ARTICLE

Beyond Von Neumann Architectures: Exploring Algorithmic Opportunities via Octantis

ANDREA MARCHESIN[1], (Member, IEEE), ALESSIO NACLERIO[1], (Graduate Student Member, IEEE), FABRIZIO RIENTE[1], (Member, IEEE), AND MARIAGRAZIA GRAZIANO[2]
[1]Department of Electronics and Telecommunications, Politecnico di Torino, 10129 Turin, Italy
[2]Department of Applied Science and Technology, Politecnico di Torino, 10129 Turin, Italy

Corresponding author: Andrea Marchesin (andrea.marchesin@polito.it)

The work of Andrea Marchesin was supported by TIM S.p.A. through the Ph.D. Scholarship.

ABSTRACT Today, one of the problems the scientific community is called upon to tackle is the well-known *von Neumann bottleneck*, which concerns the limitation in the bandwidth between the CPU and memory in a digital electronic system. Among the various solutions under study, the concept of Logic-in-Memory (LiM) has been proposed: a memory device that embeds simple computational elements between the different cells to define a distributed processing system. The present work introduces an extended version of Octantis, a novel open-source software useful for exploring LiM architectures. To achieve this goal, the internal structure of Octantis takes inspiration from the one of standard High-Level Synthesis tools, distinguishing itself from them for the target topology addressed. It analyses user-defined standard-C algorithms and determines which LiM architecture would be best suited to implement it. At its output, the tool provides a VHDL description of the synthesised circuit along with a custom test-bench. The earlier version of Octantis efficiently synthesised rather simple user-defined C algorithms. The version discussed here has been improved by extending the allowed complexity of input C-codes, like addressing nested loops and non-trivial data dependencies, and introducing hardware-specific optimisations to meet resource constraints. Several case studies have been considered to validate the newly implemented techniques and to analyse the capabilities of the tool in implementing data-intensive algorithms. The results demonstrate that Octantis can produce architectures that comply with the LiM topology while significantly reducing the exploration space to meet specific hardware requirements, such as memory dimensions and maximum logic integration. This methodology provides initial insights into potential LiM units that can be adopted in customised designs, making it a valuable tool in researching alternative electronic devices.

INDEX TERMS Algorithmic-level explorations, circuit design, Logic-in-Memory (LiM), von Neumann bottleneck.

I. INTRODUCTION

The great technological achievements of the Semiconductor Industry that have characterised the last few decades have enabled the development of even more powerful and compact electronic devices. Moore's law has been a reference for the scaling trend of the technological node (i.e., the minimum size that can be reached in the manufacturing process of integrated circuits), with all the benefits that have been there for all to see. However, as the transistors got smaller, many problems arose that began to harm the prosperous performance increase. Therefore, electronic designers had to strive in order to find solutions to let the benefits prevail over emerging critical issues [1].

Considering the traditional structure of modern electronic devices, two elements of great importance are always present: the CPU and the memory. It is their combination that enables computation on data sets. Such systems implement a reference architecture, known as *"von Neumann"* [2]. However, the continuous communication between processing

The associate editor coordinating the review of this manuscript and approving it for publication was Rahim Rahmani.

units and memory significantly weighs in terms of latency and power consumption. The increasing popularity of data-intensive applications, such as AI workloads, make this drawback clearer and more significant. Furthermore, the technological disparity between processing and memory units has culminated in system architectures where the cost of data transfer significantly surpasses that of data elaboration in terms of energy consumption [3]. This critical condition is known in the electronic field as *"von Neumann bottleneck"* [4].

In order to address this problem, research is active on several fronts, and many proposals have been presented over the years. Among them, there are Logic-in-Memory (LiM) architectures [5], [6], [7], [8], [9], [10], [11] that try to overcome the limitation in bandwidth by reorganising the processing system itself. In particular, the LiM principle proposes integrating simple logic elements within the memory array to pre-process the information stored therein so that the CPU only executes the most complex operations. It decreases the data amount transferred between the memory and the CPU while increasing the concurrency in information processing. As a result, the overall execution time is drastically reduced, as well as the power consumption of the whole system [12]. Promising results have also been obtained by implementing LiM architectures on beyond-CMOS solutions [13].

However, LiM architecture design is not straightforward, and it may become challenging when addressing complex algorithms. Furthermore, the design space to be explored may be vast, thus offering many possible LiM architectures implementing the same algorithm. In order to help designers tackle these challenges, an open-source C++ software named Octantis has been developed. The tool helps the designer understand if a C-described input algorithm could benefit from a Logic-in-Memory implementation. By implementing the standard flow usually adopted by High-Level Synthesis tools, the software automatically designs a LiM architecture that is tailored to implement the user-provided input C code, characterised at the gate-level using VHDL. It is intended to remain independent of any specific technology, thereby broadening the scope of research and development opportunities. The program additionally produces a test bench that enables verification of the circuit behaviour through external simulators. Octantis is accessible to anyone interested [14] with the aim of sharing its details and opening its development to external contributors. It's hoped that the program can inspire the definition of other similar tools for high-level exploration of alternative electronic systems for data processing.

The LiM topology taken as reference by Octantis comprises Application Specific Integrated Circuits (ASICs) composed of regular arrays of memory cells enriched with the logic gates necessary to execute a definite sequence of logic and arithmetic operations. This type of memory can be compared to a register-based architecture in digital electronics, where each memory row represents a temporary register of a more complex computational device. An example of a synthesized LiM architecture is provided in Figure 1. The architecture here depicted can implement a portion of an XNor-Net belonging to a more complex *Binarized Convolutional Neural Network*, as proposed in the literature [15]. In the context of the cited research work, multiplication is approximated by applying the XNor operation between the data stored in the memory and a fixed weight during the convolution process. The unit consists of six rows of memory cells, each associated with an address and five bits wide. The free inputs of the integrated logic gates are all connected to the data stored in the first row, thus implementing the approximated multiplication operation. The algorithm executes in parallel during a single clock cycle.

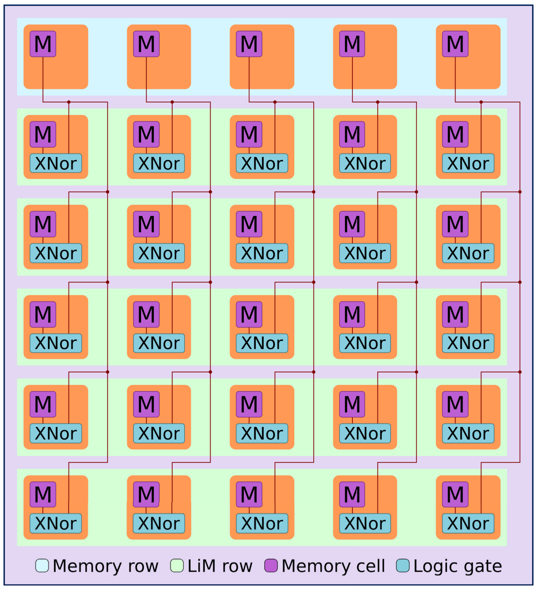


FIGURE 1. Abstract representation of a portion of an XNor-Net implemented through the LiM topology. The scheme depicts only the main elements composing the array and the logical connections to track the information flow throughout the array.

The overall circuit is suitably controlled by a control unit that manages memory locations for reading and writing data. The unit also manages information flow through the available processing elements to execute the necessary in-memory operations.

The LiM design represents a compromise architecture between a super-pipelined hardware accelerator and a memory device, which supports the primary processing unit to reduce computational efforts when running data-intensive algorithms. Representative examples of this topology can be found in [13], [15], [16], and [17].

The purpose of this article is to discuss the advancements of the Octantis reference structure, whose preliminary version has been introduced in a previously published work [18], and to present it as the first stable release made available on a public repository. The present work expands what has been presented in [18]. In particular, the current work introduces:

1) enhancements enabling processing more complex C-described algorithm;

2) new optimisation techniques introduced to reduce the resources required for a LiM unit.

Nonetheless, for sake of completeness, a thorough presentation and explanation of the inner functioning of Octantis is also provided. The paper is organised as follows. Section II presents the research background on open-source HLS tools and design automation tools for processing in memory systems that have already been proposed in literature. Section III introduces the methodology employed by Octantis' design flow and outlines how it explores LiM implementations starting from general C-described algorithms. Here, the newly developed expansions to the tool's functionalities are also examined. Section IV presents validation tests conducted to demonstrate the effectiveness of Octantis in translating the design of some LiM architectures present in the literature at the algorithmic level. Furthermore, this section discusses practical test-cases in the field of Image Processing (ImP), which are known to be data-intensive applications. The examples here illustrate the tool's effectiveness and how the new optimisations introduced impact the results. Finally, the conclusion and future perspectives are provided in Section V.

II. RESEARCH BACKGROUND AND RELATED WORK

The concept of High-Level Synthesis, also known as behavioural synthesis, aims to translate a behavioural model into hardware implementations like ASIC or FPGA designs leveraging advanced optimisation techniques. Manual implementation of traditional optimisation strategies is expensive, as are complete verification procedures [19].

Since the early 2000s, the modern Electronic Design Automation (EDA) Industry has expressed a growing interest in the field of HLS, looking upon it as a way to accelerate and improve the design process of integrated circuits. The development history of these tools has been marked by alternating fortune [20] but today, many commercial HLS tools are available on the market (e.g. *Xilinx' "Vitis"* [21], *Mentor Graphics' "Catapult"* [22] and *Intel HLS Compiler* [23]). They adopt a new paradigm of automated synthesis in which the project of an optimised integrated circuit is always accompanied by a verification infrastructure, reducing the design time and increasing the productivity of designers. Moreover, the refinement of behavioural models has contributed to the achievement of more throughput- and energy-efficient HLS designs [24].

To be fair, it is important to point out that the actual HLS tools prove practical to tackle certain kinds of applications where the solution space is suitably limited. In particular, they are now widely considered for designing circuits with a regular structure, such as memories and Intellectual Property blocks [19], whose synthesis can be customised by end-users through a set of parameters. Other HLS tools are also of interest for prototyping purposes, especially aimed at FPGAs [25], [26]. Great examples of past applications of this software can be found in the design of wireless communication networks to develop chips compliant with 3G and 4G standards [27]. Nonetheless, although manual designs take more time, they still offer superior solutions compared to HLS-based ones.

Considering that the structure of LiM architectures is regular and that the associated design constraints foresee specific optimisation techniques, the development of a tool inspired by the HLS methodology seemed a good choice for designing LiM prototypes and exploring new algorithmic opportunities. Current commercial tools are protected by patents and many of the adopted implementation choices are not accessible. Therefore, open-source projects have been considered as a reference in the definition of Octantis. In particular, LegUp [28] and Bambu [29].

LegUp is a High-Level Synthesizer, developed by University of Toronto researchers since 2011, intended to design custom FPGA accelerators. Formally, it started as a C-to-Verilog HLS, based on the LLVM compiler infrastructure. Along with the input algorithm to be synthesised, it is also necessary to provide the tool with a configuration file to define all the design constraints. The synthesis process produces a complete project of a hardware accelerator that can be implemented inside SoC boards equipped with FPGA modules. Specific support for commercial boards is given. The tool also generates a testbench, useful to verify the correct behavior of the accelerator.

Bambu is a C-to-HDL tool developed at the Politecnico di Milano since 2013, whose purpose is to produce an ASIC optimised for implementing an input algorithm. Also, this tool produces a complete architecture and a test-bench to verify the functional correctness of the project. Moreover, Bambu can produce two different versions of an output design, compliant with VHDL or Verilog standards.

Octantis implements the general structure of traditional high-level synthesisers [30], from which it inherits many techniques. However, differently from state-of-the-art tools, it particularly addresses the design of LiM architectures, adopting advanced optimisation strategies specific to their topology, presented in the Introduction. Hence, the obtained design, described at an architectural level, can be further optimised and implemented considering traditional electronic devices but also alternative technologies, opening up the possibility of delving into beyond-CMOS solutions. This can be made possible by finalising the design with CAD programs developed for such applications, as the one discussed in [31]. In this regard, the program is useful for exploring, from a research perspective, possible alternatives for future electronic devices.

Considering the literature associated with the automated design of processing in memory systems, to which the LiM principle belongs, several tools have been proposed over the years [32], [33], [34]. These tools mainly focus on logic synthesis from Boolean functions, optimising translation into a sequence of instructions for execution on memristive memory arrays [35]. While these tools have proven valuable for mapping operations on a reference memory architecture, Octantis stands out for its exploration of implementations that refer to a different topology of processing-in-memory systems. Octantis is capable of accepting higher-level algorithms as input, which are typically more complex and expressive than previous software can manage. Moreover it returns a complete LiM architecture description that is technology-independent. As previously introduced, Octantis is a first-analysis support tool to identify opportunities for LiM implementation at the algorithmic level, providing solutions that can be potentially applied to different target technologies. For this reason, the discussion in the following sections intends to present the exploration capabilities of Octantis at the *architectural level*. Therefore, the final implementation of the produced solutions in specific technologies will not be detailed, as they are not part of this work. For an insight on how these architectures can be mapped on beyond C-MOS technologies and, in particular, on the pNML (i.e., perpendicular Nano Magnetic Logic) one, please refer to the example in [13].

The tool has been developed as C++ software built using the LLVM compiler infrastructure, with some of its components derived from it. The core of the program is implemented as an LLVM Pass named *"OctantisPass"*, and comprises several modules that enable the C code provided by the user to be transformed into the final architecture in a step-by-step process.

At the beginning, the input C algorithm is fed to Clang, the front-end of the LLVM compiler framework. Clang translates the input C code to a byte code translatable to assembly-like language known as LLVM Intermediate Representation (IR) [36]. Then, it is processed by specific passes already integrated within the LLVM framework. After that, *OctantisPass* is invoked, and after undergoing some further algorithmic optimisations, the resulting LLVM IR is mapped onto a LiM array using the standard 4-staged LiM synthesis flow that includes *allocation*, *scheduling*, *binding* and *code generation*. These stages are fundamental building blocks of any traditional high-level synthesiser [30], but their implementation on Octantis targets the generation of a LiM architecture that complies with the description provided above.

The overall structure of Octantis, illustrated in Figure 2, highlights all the main modules of the program by making explicit their interrelationships and how they relate to the *LLVM environment*. The first version of Octantis [18] implemented the primary design flow for generating simple but complete LiM architectures from a reduced set of input C instructions. It established the tool's workflow's central elements and was designed to be modular, allowing for easier code maintenance and expanding its capabilities through incremental updates.

In order to provide a comprehensive analysis, the rest of the article will initially offer a general outline of Octantis, emphasising the algorithmic decisions taken in defining its modules. Then, the discussion will focus on the expansions of the functionalities introduced in this work and the related details will be thoroughly argued.

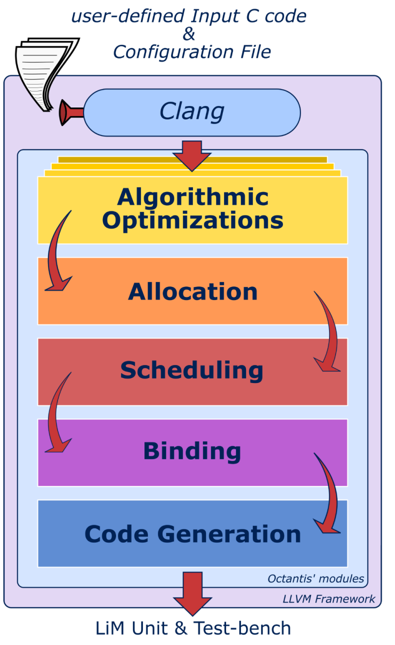


FIGURE 2. Graphical representation of Octantis structure and working flow. It highlights the grouping of the main modules developed and incorporated into the program within the blue rectangle. The wider purple box denotes that the tool has been defined within the LLVM framework, adopting its formalism and inheriting generic compilation strategies.

III. OCTANTIS STRUCTURE

The aim of this section is to present an in-depth analysis of Octantis primary modules in order to clarify the complete design flow and methodology adopted by the tool. Although some of the discussed features were already available in the first version of the tool presented in [18], they are examined in detail for the sake of completeness. Additionally, the novel implementations in this work are showcased. It is important to notice that the internal organization of the tool remains unchanged, and the new strategies have been implemented by expanding the capabilities of some pre-existing modules.

A. INPUT C CODE SPECIFICATIONS

As previously introduced, the user must provide Octantis with a C-described algorithm that must consist of a single function that implements the algorithm to be mapped onto a LiM array. However, to make it more appropriate for hardware design, the description of the input algorithm allows limited expressiveness, similar to that of established HLS tools. In particular, the constraints on the user-defined input code can be summarised in the following statements:

- Dynamic data allocation and management are not allowed.
    - The input algorithm will run on a LiM unit, a hardware component with a defined amount of physical resources.
- Recursive function calls are not allowed either.
    - The input code should avoid complexities that would make the synthesiser work more difficult, which, in some cases, would be unable to optimise the mapping properly in a LiM architecture.

- The input data must be defined through integer variables.
  - The architectures to be synthesised are capable of efficiently processing information expressed as integers and floating-point data is not yet supported.

In addition to these, one suggestion more related to the reference topology of LiM architectures is also introduced:

- Multiplications and divisions should be avoided.
  - The complexity of the hardware components required to implement these operations is known, which makes it difficult to introduce them into a memory array. As will be detailed in the following sections, these operations can be forcibly integrated into the synthesised LiM unit only with approximations.

The tool also requires a configuration file to be provided. It specifies information on the design constraints and the optimisations to be adopted during the exploration process.

B. PRE-PROCESSING OF THE C ALGORITHM THROUGH CLANG AND GENERIC LLVM OPTIMIZATIONS

The general structure of a high-level synthesiser, common to all compilers, consists of two main elements: the front-end and the back-end. The former represents the input interface of the program, and it has to deal with a very abstract representation of the algorithm. It performs lexical and syntactic analyses to ensure the formal correctness of the provided code, and ultimately elaborates it into a standard format, with a lower abstraction level and independent of both the input source and the output target. The latter considers this intermediate code and produces an optimised "translation" into the format compatible with the destination architecture. The separation between these two components refers to the re-usability principle, keeping constant the front-end component for a specific source code language and the back-end component for each target architecture.

As the front-end performs standard analyses on the input code, it has been decided to integrate an existing one into Octantis. The choice fell on Clang [37], which is the front-end compiler of the LLVM framework, one of the most popular open-source compilation tools available.

After Clang's verification checks, the algorithm provided by the user is translated into LLVM intermediate representation, where it undergoes processing via specific tools integrated within the LLVM framework (e.g., mem2reg, simplifycfg, licm, and loop-simplify passes), which aim to implement high-level optimisations. These strategies are designed to simplify the algorithm without interfering with the subsequent phases of code processing. The application of these optimisations is defined within the input configuration file.

The resulting code is fed to *OctantisPass*, where advanced optimisations are applied. These custom optimisations are specifically beneficial for the LiM architecture exploration process.

C. OCTANTIS ALGORITHMIC OPTIMIZATIONS PERFORMED ON THE USER-DEFINED C CODE

The input C-code provided by the user may require processing through algorithmic optimisation techniques in order to identify opportunities for parallelisation, thereby enabling LiM implementations. Specifically, Octantis implements strategies that are aimed at identifying and optimising loops.

Loops play a significant role in algorithms as they are commonly used to express parallel and compact codes. They can be classified based on the data dependencies that relate to the variables involved. More specifically, a loop can be categorised as either an *independent* or *dependent* loop. The previous version of Octantis was only capable of handling single independent loops. The work presented in this paper significantly expands the algorithmic optimisations stage of the tool. In its current version, Octantis can identify *nested loop structures* within the C-described algorithm and distinguish their respective category. Then, specific strategies are introduced to handle and optimise both dependent and independent loops. As a result, the enhancement of this module allows the tool to support the analysis of more complex algorithmic constructs. The details of the adopted optimisations will be provided below.

1) INDEPENDENT LOOPS OPTIMIZATIONS

An independent loop is comprised of iterations that are not correlated with each other. As a result, each iteration's execution can be easily mapped to parallel processing units, and out-of-order processing is also allowed. In this favourable scenario, the maximum execution speed can be expressed as:

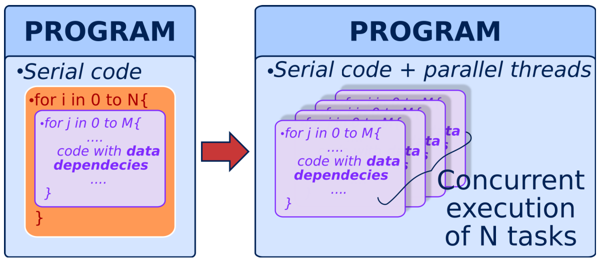
$$Max\ speed-up : S(I,N) = \frac{I}{\lceil I/N \rceil} \tag{1}$$

where the variable I represents the number of iterations that characterise the cycle, and N is the number of parallel processing units available.

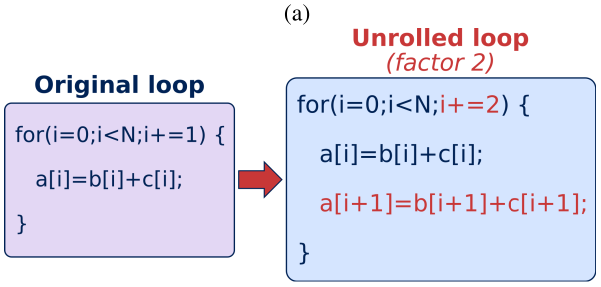
In the case of independent loops, Octantis tries to parallelise the execution of the given algorithm by applying *Loop Spreading* and *Loop Unrolling* techniques concurrently. The former targets nested loops and aims to allocate them to parallel processing units, while the latter tries to condense multiple iterations to reduce the total number of iterations in a loop. The diagram depicted in Figure 3 serves as a conceptual representation of these optimisation strategies. It is important to note that their effectiveness is contingent upon the dependencies in the input code. Therefore, it is the responsibility of the end-user to provide a high-quality input C code that is arranged in a manner that avoids dependent loops to the greatest extent possible. In doing so, the aforementioned techniques can be applied, and instructions executed in parallel, resulting in the full exploitation of the potential of a LiM implementation.

2) DEPENDENT LOOPS OPTIMIZATIONS

A dependent loop is characterised by the presence of dependencies among different iterations of the same loop.


PROGRAM
•Serial code
•for i in 0 to N{
•for j in 0 to M{
....
code with data dependecies
....
}
}
PROGRAM
•Serial code + parallel threads
•for j in 0 to M{
....
code with data dependecies
....
}
Concurrent execution of N tasks

(a)



```
for(i=0;i<N;i+=1) {
  a[i]=b[i]+c[i];
}
```

```
for(i=0;i<N;i+=2) {
  a[i]=b[i]+c[i];
  a[i+1]=b[i+1]+c[i+1];
}
```

(b)

FIGURE 3. Intuitive examples of the two algorithmic strategies implemented in Octantis to optimise loops execution: in (a) *Loop Spreading* and in (b) *Loop Unrolling*.

These dependencies are commonly known as *Loop-Carried Dependencies* and represent a significant obstacle to the parallelisation opportunities of the input algorithm. The violation of logical relations imposed by these dependencies can lead to incorrect results, emphasising the need for caution when handling them. However, addressing loop-carried dependencies is a complex task.

Octantis, in its current version, supports the detection of a particular kind of dependency that arises from *accumulations*. These operations involve adding together all elements of a set, usually an array, and storing the resulting sum in a single variable. This results in dependencies between loop iterations, preventing the ability to run different iterations in parallel. Octantis has been designed to identify and denote its presence for further processing. While accumulations cannot be entirely parallelised, they can still be optimised by mapping them efficiently onto a LiM array. This optimisation is implemented during the Binding section, as it is closely linked to the available hardware. More information regarding this optimisation will be provided in the corresponding section.

D. ALLOCATION

Octantis parses the information provided by the configuration file to define all the characteristics required for the LiM Unit to be properly designed. In particular, the *size of the memory* and the *word-length* of the data to be stored and processed within the architecture. This information will prove essential during the *code generation phase*.

E. SCHEDULING

One of the most important contributions to the entire exploration process is attributable to the scheduling phase. The derived results, in fact, have a strong impact on all three figures of merits characterising the final LiM architectures: *latency*, *occupied area* and *static* and *dynamic power consumption*.

The implemented scheduling algorithm analyses the sequence of instructions of the optimised input code and defines their allocation to build a complete *Data Flow Graph (DFG)* of the overall algorithm. The chosen strategy is the *As Soon As Possible (ASAP)* approach which prioritises the performance of the explored architectures. As the name suggests, this algorithm executes each operation immediately upon the availability of its input operands.

Algorithm 1 Pseudo-code of the DFG building process

```
**DFG**: data-flow graph containing all detected instructions useful for the implementation of the input C code
**instNd**: instruction node already present inside the DFG
**newInstNd**: new instruction node to be inserted in the DFG
**instList**: list of LLVM IR instructions
**destOp**: destination operand of an instruction node in the DFG
**srcOp**: source operands of an instruction node in the DFG

**Input**: list of LLVM IR instructions
**Output**: DFG whose nodes contain useful information about the LLVM IR instructions they correspond to

function dfgBuilder(instructionList)
    for all inst ∈ instList do
        if inst is valid then
            Check if inst is related to an accumulation
            Analyze Aliases of operands used by inst
            Create node newInstNd containing information about inst
            for all instNd ∈ DFG do
                if newInstNd srcOp == instNd destOp then
                    if checkForNegLogic(instNd, newInstNd) then
                        Change instNd operator to its negative (i.e. xor → xnor)
                    else
                        Insert newInstNd in DFG
                        Insert edge connecting instNd and newInstNd
                    end if
                else
                    Insert newInstNd in DFG
                end if
            end for
        end if
    end for
end function
```

In order to work correctly, the scheduler needs to understand the semantics of the different LLVM IR instructions, evaluate the logical relationships between them, perform optimisations and produce the final DFG. As for addressing the increasing complexity, DFG building process now incorporates suitable analyses that support more complex algorithmic structures. The DFG building process is described with the pseudo-code presented in Algorithm 1, and it takes into consideration four key aspects:

- **Analysis of the allowed instructions**
  - As previously discussed, Octantis can consider only a subset of C instructions, the ones that make sense for the synthesis of a LiM unit. If an unsupported instruction is identified in the LLVM IR code, the scheduler provides an error message to the user describing the details of the incompatibility found and stops the synthesis process. Multiplications and divisions lead to generating an information message, as they will be implemented through hardware shifters, which introduce approximations of the final results. Valid instructions are inserted into the DFG as a node with relevant information. A special case is accumulations, which are recognised by the algorithmic optimisation stage to inform the scheduler about its presence. As a consequence, the DFG building process can condense the accumulation-related information into a single node, referred to as *accumulation node*.
- **Alias analyses**
  - This operation is directly due to LLVM intermediate representation language. Its formalism, in fact, allows the presence of multiple aliases for a single data stored in memory. To optimise the exploration process, the scheduler must keep track of these variables to reduce redundancies and simplify the subsequent processing steps of Octantis.
- **Analysis of the data dependencies**
  - In order to effectively implement the ASAP algorithm, the scheduler must consider the dependencies between instructions and determine the *degree of mobility* of each operator. This refers to the specific time intervals in which an operation can be executed without violating the logical sequence of the algorithm. By analysing these factors, the scheduler can allocate the execution of instructions to maintain the algorithm's logical flow.
- **Advanced logic substitution**
  - ANSI C language and LLVM intermediate representation do not support the full set of Boolean operators, unlike *Hardware Description Languages (HDLs)*. For instance, the NAND operation is not available, despite being a commonly used logical operation. Instead, it is implemented by exploiting a sequence of other instructions. In LLVM intermediate representation, a NAND operation is achieved by performing two subsequent AND, as outlined below:

LLVM IR Code	**Octantis operation**
%9 = and i32 %7, %8	%10 = %7 nand %8
%10 = and i32 %9, −1	

Therefore, the scheduler analyses the algorithm provided by the user to identify typical patterns for describing specifically negative logic operators (i.e. NAND, NOR, XNOR). Upon detecting the related patterns, the scheduler replaces the corresponding instructions with a unique and equivalent negative logic operation. In Algorithm 2, the *checkForNegLogic* function implements the explained analysis.

Algorithm 2 Pseudo-code of the negative logic substitution analysis

```
instNd1: a DFG instruction node
InstNd2: a DFG instruction node whose first source operand is equal to the destination operand of instNd1

Input: two DFG nodes, namely instNd1 and instNd2. The first source operand of instNd is equal to the destination operand newInstNd.
Output: boolean value indicating if the operations related to the two input nodes represents a negative logic bitwise operation pattern

function checkForNegLogic(instNd1, instNd2)
    if instNd1 operation is bitwise then
        if instNd1 operation == InstNd2 operation then
            if second source operand of InstNd2 == -1 then
                Return True
            end if
        end if
    end if
end function
```

Upon completion of the building process of the DFG, each node within it represents an LLVM IR instruction (or an accumulation that will be later mapped onto a LiM Array). Moreover, if a data dependency is detected between two instructions, the corresponding nodes are linked by an edge.

After the construction of the DFG, the data structure is traversed to assign each node with an appropriate starting time, according to the ASAP algorithm. The pseudo-code of this process is provided in Algorithm 3.

Algorithm 3 Pseudo-code of the ASAP Scheduling process

```
DFG: data-flow graph containing all detected instructions useful for the implementation of the input C code
instNd: DFG node related to an LLVM IR instruction
sTime: starting time of a node in the DFG
maxParentStartTime: max starting time among all parent nodes of instNd
parentNds: list of parent nodes of instNd
```

```
Input: DFG produced by the dfgBuilder function
Output: DFG whose nodes have been annotated with the
starting times devised by the ASAP scheduling algorithm

function asapScheduler(DFG)
    for all instNd ∈ DFG do
        if all parentNds have an assigned sTime then
            sTime of instNd ← maxParentStartTime + 1
        end if
    end for
end function
```

F. BINDING

The primary function of the binder is to execute the mapping of input operators, each assigned to specific time slots, to hardware units that are capable of executing them. To achieve this, it analyses the DFG generated by the scheduler and organises the information into two distinct data structures, namely the LiM Unit and the Finite State Machine:

- The *LiM Unit* is equipped with the necessary memory rows and logic elements that are properly connected to enable the execution of the user-defined algorithm.
- The *Finite State Machine (FSM)* regulates the behaviour of the designed architecture over time, providing all the signals required to time the operations.

The obtained solution complies with the reference topology of LiM architectures, discussed in the Introduction. However, with respect to the previous work [18], the binder is enhanced with the introduction of several hardware-dependent optimisation strategies. These optimisations are applied to reduce the complexity of the system and its required resources. Given the crucial role of these strategies in achieving favourable exploration results, the following section will provide detailed information regarding the choices adopted in their implementation.

1) HARDWARE-DEPENDENT OPTIMIZATION STRATEGIES

The binder mapping process involves traversing the DFG generated by the scheduler. Each node in the DFG contains information regarding the corresponding LLVM IR instruction, including its source operands, the type of operation to be performed, and its starting time.

Algorithm 4 Pseudo-code of the Binder Mapping Process

```
Input: DFG produced by the dfgBuilder function and
annotated with starting times by the asapScheduler function
Output: LiM Array that implements the algorithm defined
by the user
function MappingProcess(DFG)
    for all node ∈ DFG do
        if operation of node is load then
            Insert memory row in LiM Array
        else if operation of node is accumulation then
            Accumulate(node)
        else
            HdOpt(node)
        end if
    end for
end function
```

As presented in Algorithm 4, during the through passing, different strategies are employed according to the type of operation associated with the node. The nodes visited first usually represent *load* operations involving the algorithm input operands. Therefore, the binder starts to directly populate the LiM Array with the necessary memory rows for storing these input data. Subsequently, the process goes over the nodes that entail actual calculations. The type and starting time of each node are obtained, and the LiM Array rows containing the two source operands, referred to as source LiM rows, are identified. Here, a specific function handles the definition of additional hardware operators required to implement the operation, attempting to produce the most compact, yet performing, LiM Array. The pseudo-code of the optimisation procedure is reported in Algorithm 5. The primary objective of the algorithm is to analyse the two source LiM rows and evaluate two specific conditions:

- Check if either of the source LiM rows already integrates the operator required by the DFG node instruction. In this case, if the operator is not used by any other instruction at the same time, the corresponding LiM row is chosen to perform the instruction indicated by the node.
- If the previous condition is not met, the algorithm checks if at least one source LiM row integrates fewer operators than a pre-defined amount, identified as *MAX_OP* in Algorithm 5. If this is the case, the selected source LiM row is equipped with the necessary operator.

If neither of the above conditions is met, a copy of one of the two source LiM rows with the required operator is inserted inside the LiM Array. In all cases, the LiM row that contains the needed operator will receive the other source LiM row's content as input.

The optimisation strategy discussed is used to achieve three purposes:

1) The number of memory rows must be minimised while ensuring the correct implementation of the input algorithm.
2) It is crucial to set boundaries to the number of logic elements within the single memory cells, as it is a challenging aspect to implement from a technological perspective, as observed in the literature presented in the Introduction section.
3) In order to contain interconnections complexity, data locality per subsequent operations is preserved. Indeed, interconnections represent a delicate design aspect as they directly impact data integrity, power consumption, and overall performance.

Algorithm 5 Pseudo-code of the Hardware-Dependent Optimisation Strategy

```
MAX_OP: maximum number of operators that can be integrated into a single LiM Row

srcRow1: LiM row storing the first source operand
srcRow2: LiM row storing the second source operand
destRow: destination LiM Row
op: logic element to be integrated (or already present) within the cells of one of the two source rows
sTime: starting time for the operation retrieved from the scheduled DFG node

Input: a node of the DFG
Output: The function handles the mapping of the input node operation onto the LiM Array that is being synthesised. It also returns the destination LiM row

function HdOpt(node)
    Get operation source operands from node
    Get srcRow1 and srcRow2 storing the source operands
    Get op and sTime from node
    destRow ← newly generated LiM row
    if srcRow1 contains op inactive at sTime then
        Exploit op of srcRow1
        Connect srcRow2 to op of srcRow1
        Connect output of srcRow1 op to destRow
    else if srcRow2 contains op inactive at sTime then
        Exploit op of srcRow2
        Connect srcRow1 to op of srcRow2
        Connect output of srcRow2 op to destRow
    else if srcRow1 has less operators than MAX_OP then
        Integrate op into srcRow1
        Connect srcRow2 to op of srcRow1
        Connect output of srcRow1 op to destRow
    else if srcRow2 has less ops than MAX_OP then
        Integrate op into srcRow2
        Connect srcRow1 to op of srcRow2
        Connect output of srcRow2 op to destRow
    else
        Create srcRow1Copy, copy of srcRow1
        Integrate op into srcRow1Copy
        Connect srcRow2 to op of srcRow1Copy
        Connect output of srcRow1Copy op to destRow
    end if
    Return destRow
end function
```

In addition, a specific optimisation strategy has been integrated into the Binder to handle accumulations, as introduced in Section III-C. This strategy is designed to identify the set of operands that constitute the so-called *Initial Accumulation Set*, which is the group of operands that must be summed together, and their corresponding LiM source rows. To obtain the final result, the content of these rows must be added in a *Reduction Tree* manner, as illustrated in Figure 4. With reference to the example in Figure 4, the reduction process begins with the initial accumulation set, whose operands are stored in the rows indicated by the leftmost nodes. Within this group of source rows, four couples are identified and the associated additions are mapped onto the array using the function already presented in Algorithm 4. Consequently, four destination LiM rows are inserted in the LiM Array to store the output of these operations. The destination rows are assigned a level equal to 1, indicating intermediate results of the accumulation. It is noteworthy that the two source LiM rows may possess different levels. In such cases, the resulting destination LiM row will have a level equal to the higher of the two source LiM row levels incremented by one. The set of all destination LiM rows undergoes the same reduction process, which is executed until the final result is obtained.

The Reduction Tree mapping strategy allows to speed up the overall execution time of this kind of operation, which is equal to $\log_2 N$, where N is the number of elements of the initial accumulation set.

Algorithmic details on the handling of accumulations can be found in Algorithm 6.

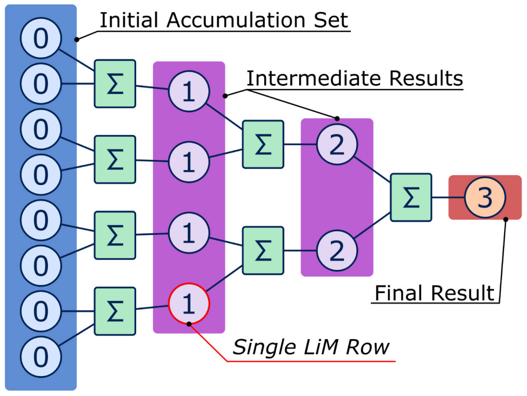


FIGURE 4. Reduction tree employed in the mapping of accumulations onto a LiM Array. Each node represents a different memory row, with the number within denoting its level in the Reduction Tree. These memory rows may contain an initial accumulation set operand, an intermediate result, or the final result.

Furthermore, Octantis implements an additional optimisation technique to further improve the mapping of accumulations. The technique is presented in Algorithm 7 and it involves identifying redundant information in the accumulation set and specifically recognising when pairs of memory rows have already been summed. This strategy avoids the insertion of unnecessary operations in the LiM Array, and their result is shared among the involved sum operations. The LiM rows belonging to the final set are accumulated using the Reduction Tree strategy described before.

Algorithm 6 Pseudo-code of the Accumulation Mapping Process

```
initAccSet: set of source LiM rows storing the initial accumulation set operands
```

```
**redAccSet**: set of source LiM rows storing the reduced accumulation set operands
**sTime**: starting time of an addition within the reduction tree
**coupleRowsList**: list composed of couples of LiM rows whose addition has already been mapped onto the LiM Array
**lSrc1**: level associated to srcRow1 in the reduction tree
**lSrc2**: level associated to srcRow2 in the reduction tree
**lDest**: level associated to destRow in the reduction tree
**lvl**: level associated to a row in the reduction tree

**Input**: a DFG node whose operation is accumulation
**Output**: the input accumulation node is mapped onto the LiM Array

function Accumulate(node)
    Get the initial accumulation set operands.
    Get source LiM rows storing the initial accumulation set operands
    Add source LiM rows to initAccSet
    redAccSet ← AccumulationOpt(initAccSet, coupleRowsList)
    while redAccSet is not empty do
        (srcRow1, srcRow2) ← couple of LiM rows in redAccSet with the lowest level
        sTime ← max(lSrc1, lSrc2)
        tmpAccNode ← temporary addition node with srcRow1 and srcRow2 as source LiM rows and starting time equal to sTime
        Insert couple (srcRow1, srcRow2) in coupleRowsList
        destRow ← HdOpt(tmpAccNode)
        lDest ← max(lSrc1, lSrc2) + 1
        Insert destRow in redAccSet
        Remove srcRow1 from redAccSet
        Remove srcRow2 from redAccSet
    end while
end function
```

Algorithm 7 Pseudo-code of the Optimisation Strategy for Accumulations

```
**initAccSet**: set of source LiM rows storing the initial accumulation set operands
**redAccSet**: set of source LiM rows storing the reduced accumulation set operands
**destRow**: LiM row storing the result of an already mapped addition between a couple of source LiM rows

**Input**: coupleRowsList (defined in Algorithm 6) and the set of LiM rows storing the operands of the initial accumulation set
**Output**: reduced set of LiM rows

function AccumulationOpt(initAccSet)
    existCouples ← true
    redAccSet ← initAccSet
    while existCouples do
        existCouples ← false
        for all couple ∈ redAccSet do
            if couple ∈ coupleRowsList then
                Get destRow related to couple
                Remove couple from redAccSet
                Insert destRow in redAccSet
                existCouple ← true
            end if
        end for
    end while
    Return redAccSet
end function
```

The application of the described optimisation is most beneficial whenever Octantis has to synthesise algorithms exploiting many accumulation sets that share several operands. Indeed, in these cases, the probability of finding additions that have already been mapped is much higher. The implementation of this optimisation technique ensures that the execution time for the accumulation is kept in the order of $\log_2 N$, but with a reduction of the number of source operands N, while reducing the LiM Array area occupation.

In conclusion, the binding operation yields a LiM array and its corresponding FSM, effectively concluding the design process. These two units are further elaborated in the final module of Octantis, enabling them to be represented in the desired file format.

G. CODE GENERATION

The final phase of the exploration process incorporates all the information gathered in the preceding steps to generate the required set of files in VDHL. The code generator produces a *LiM array*, a *Control Unit*, and a *test-bench* for verifying and characterising the behaviour of the solution. The test-bench is available as a complete functional test template in VHDL file format, and the user is only required to provide details regarding the data vectors to be applied.

More details of the final LiM Unit are defined during this last stage. Specifically, the interconnections between memory cells are defined, and multiplexers are introduced to ensure the correct dispatching of various signals.

Upon completing the output files production, it is possible to further evaluate the solution generated by Octantis with other EDA tools and implement it in a specific target technology. This will provide relevant information about the circuit's performance, spatial utilisation, and power consumption. The adoption of a standard HDL representation for the output architectures enhances the interoperability between Octantis and tools capable of synthesising the solutions at a circuit level. In the current version of the work, the responsibility for these implementation choices lies with the end-users

IV. RESULT

In order to validate the proposed methodology, several tests have been conducted. They can be divided into two categories. The former encompasses primary tests have been performed to validate the resulting architectures with the LiM topology of reference. The latter considers more complex algorithms that have been taken as examples to look at the potential of this exploration process. Since Image Processing (ImP) algorithms have been already demonstrated highly compatible with LiM implementation [38], they have been chosen as test cases. For each test, a specific C code has been defined, along with an associated configuration file to specify the word-length of data and to adopt the most suitable optimisations available to improve the final results.

After running Octantis, the LiM architectures obtained have been characterised in terms of *memory composition*, *memory dimension* and *execution time*. As previously pointed out, the LiM unit synthesised by Octantis is technology-independent, and these metrics are extracted by considering the final architecture as a classical register-based architecture. The LiM design produced by Octantis is generally composed of different types of rows. The term memory composition refers to an analysis of the different types of LiM rows synthesised by the tool. For instance, a memory row where XOR gates are integrated is referred to as "Xor Row". A memory row without additional logic is called "Simple Memory Row". The memory dimension is calculated as the product of the number of LiM rows and the bit-width of a single row. The total execution time is equivalent to the number of clock cycles necessary to perform a logic simulation of the algorithm. During the binding phase, this metric is extracted and it is exploited for the definition of the FSM. It is indicated with the T_{clk} notation in the following text and tables. To ensure reproducibility, the materials used to conduct the experiments have been made available in the *online open repository* [14] along with Octantis source code.

A. VALIDATION OF THE PROPOSED METHODOLOGY

Regarding the validation procedure, three algorithms have been taken as a reference, particularly those proposed in [13], [15], and [16]. Reference C codes and proper configuration files have been defined. An extract of the results obtained by Octantis is collected in Table 1, expressed in terms of integrated logic in the memory array. They are compared with the characteristics of the architectures proposed in the articles.

All the produced architectures have resulted completely equivalent to the reference ones, and two out of three are the same also in terms of integrated hardware. The last test case, the one considering [13], has shown a limited overhead as the architecture proposed by the authors had been manually designed with the adoption of customised optimisation techniques, not implemented within Octantis. It is important to highlight that the optimisation phases introduced in the new version of Octantis did not further improve the results obtained for these three algorithms with respect to [18]. This was expected, however, as the complexity of these algorithms is relatively low, thus allowing the first version of the tool to already reach an optimal solution. In conclusion, the validation through these preliminary tests revealed success.

B. IMAGE PROCESSING ALGORITHMS: EXPLORING LIM IMPLEMENTATIONS

The field of ImP is a branch of Computer Vision, which involves the execution of specific operations on digital images, according to given algorithms, with the aim of modifying them to improve their quality or extract meaningful information. However, Image Processing algorithms are known to be computationally intensive, and their execution time can be significantly important if appropriate parallelisation techniques are not employed.

In this regard, Logic-in-Memory architectures may represent an alternative solution to address the performance issues, thanks to their intrinsic parallel computation capabilities, as pointed out in [38]. A subset of typical operations present in ImP algorithms has been identified to be effectively mapped onto a LiM architecture. In particular, *accumulations* and *bit-wise operations* have been recognised as critical processing elements in a vast number of algorithms. They are used to modify the initial image directly (e.g., for the implementation of local filters), generate intermediate data structures needed for further elaboration stages or extract regions of interest using *masks*. Bit-wise operations are also commonly employed in image encryption, where XOR and XNOR operators are primarily considered.

With the purpose of highlighting the potential benefits deriving from a Logic-in-Memory implementation, three algorithms [39], [40], [41] have been considered to be run on customised LiM architectures designed via Octantis. After a thorough analysis of the algorithms, it has been determined that only a portion of the first two algorithms was suitable for LiM implementation, which has been subsequently processed through Octantis. Instead, the third algorithm was entirely implemented using LiM architecture. These algorithms have been selected to evaluate the exploration capabilities of Octantis and its ability to benefit from dedicated and efficient hardware acceleration. The optimisations that have been implemented in the tool have been thoroughly examined to determine their effectiveness, and the derived results will be detailed in the following.

The first algorithm investigated is the *multi-image encryption algorithm* presented in [39] by *Huang, Z.J et all.* The last stage of the algorithm has been considered for LiM implementation, which involves performing the XOR operation between two intermediate 256×256 images to obtain the final output cyphertext image. The XOR operation is employed in the proposed encryption scheme

TABLE 1. Results of the preliminary tests conducted on Octantis to validate the proposed architectures compared to the LiM topology.

Implemented Algorithm	Architecture designed by Octantis			Reference Architecture		
	Memory dimensions	Integrated Logic	Exec. time	Memory dimensions	Integrated Logic	Exec. time
X-NOR Net [15]	25 bits	XNor: 25	1 T_{clk}	25 bits	XNor: 25	1 T_{clk}
Bitmap Indexing [16]	144 bits	And: 128 Or: 128 Xor: 128 Mux 3-to-1 16 bits: 8	T_{exe} (a)	144 bits	And: 128 Or: 128 Xor: 128 Mux 3-to-1 16 bit: 8	T_{exe} (a)
Convolutional Neural Network [13]	136 bits	Full/Half-Adder: 64 Mux 2-to-1 8 bits: 9	T_{exe} (a)	120 bits	Full/Half-Adder: 48 Mux 2-to-1 8 bits: 9	T_{exe} (a)

(a) As the reference architectures are configurable and so many algorithms can be implemented, the execution time is expressed in a parametric way.

to improve the robustness against chosen-plaintext attacks. In the reference work, tests have been conducted considering four 256 × 256 grayscale images as inputs. According to the algorithm's specifications, the size of the two intermediate images on which the XOR operation must be performed is the same as the input ones.

To implement this algorithm on LiM architecture, two matrices representing the intermediate images have been declared in the C code given in input to Octantis. Since grayscale images have been considered, Octantis configuration file has been characterised to generate a memory with a word-length of 8 bits, and loops optimisations have been enabled. Two nested for-loops have been exploited to visit the two matrices in row-first order and perform the XOR operation between their elements with position (i, j). Several syntheses through Octantis have been run with different matrices size, namely 2 × 2, 4 × 4, 8 × 8, 16 × 16, 32 × 32, 64 × 64, 128 × 128 and 256 × 256.

As expected, regardless of the size of the images, Octantis design process has recognised the opportunity of unrolling the two nested loops, allowing the concurrent execution of all needed XOR operations. This parallel implementation results in a very low execution time of just one clock period for all the size cases. The produced memory size is such that it contains both the input images and the output image, and it increases along with the matrices size. The memory rows are classified based on the type of integrated logic they feature. "Simple memory" rows do not present additional logic, while "Xor" ones feature an XOR gate. The memory rows related to one of the two input images integrate XOR logic gates to perform the logic operation, and they are classified as "Xor" rows. Octantis exploits the optimisation strategy explained in Algorithm 5 to equip already-instantiated memory rows with XOR gates. Figure 5 shows, on a semi-logarithmic scale, how the number of "Simple memory" and "Xor" rows, as well as the total amount of memory rows, increases by changing the matrices size. As expected, it can be easily noticed that they all present the same exponential behaviour. Although this algorithm allows to get a glimpse of the benefits enabled by the optimisation strategy presented in Algorithm 5, the next benchmark examines them in more detail.

In reference to the second test, a proposed *multi-image encryption algorithm* presented in [40] *Li, X et all* has been

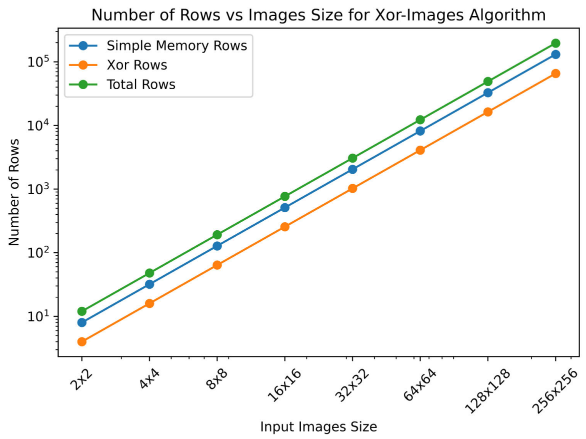


FIGURE 5. Number of "Simple memory" and "Xor" rows for each Octantis' synthesis run with a different images size.

considered. The algorithm involves the generation of the "XOR-Image" as a central operation. This is achieved by applying the XOR operator to a set of images in order to obtain the respective *"scrambled"* versions. The operation can be expressed using Equation 2,

$$XOR_{Image}(i,j) = IMG_0(i,j) \oplus IMG_1(i,j) \oplus \dots \\ \dots \oplus IMG_{n-1}(i,j) \oplus IMG_n(i,j) \quad (2)$$

where IMG_k denotes the k-th scrambled image. Usually, images are represented as matrices of pixels, and a specific pixel can be identified using the (i, j) index. As a result, the XOR-Image is obtained by applying the XOR operation to the pixels at the same position of all scrambled images. Meanwhile, an *XOR-Key* must be generated for each scrambled image to decrypt the images. The key for the k-th image is produced by performing the XOR operation on all the scrambled images, except the k-th one, as shown in Equation 3:

$$XOR_{Key_i}(i,j) = IMG_0(i,j) \oplus IMG_{i-1}(i,j) \oplus \dots \\ \dots \oplus IMG_{i+1}(i,j) \oplus IMG_n(i,j) \quad (3)$$

The algorithm has been implemented through a C code provided in input to Octantis along with the configuration file. The word length parameter has been set to 8 bits since grey-scale images were used. However, the sizes of the six input images has been changed with respect to the ones

TABLE 2. Overall memory reduction achieved by the optimisations strategies introduced within Octantis synthesis flow for algorithms proposed in [40] and [41].

Algorithm	Images Size	Memory Reduction
Li, X et all [40]	2x2	7.4%
	4x4	7.4%
	8x8	7.4%
	16x16	7.4%
Viola, P. and Jones, M. [41]	2x2	7.7%
	4x4	56.4%
	8x8	82.0%
	16x16	93.2%

TABLE 3. Memory composition of the resulting LiM design produced by Octantis for the multi-image encryption algorithm proposed by Li, X et all [40].

Algorithm	Images Size	Simple Memory Rows	Xor Rows	Xor Rows with 2-to-1 mux	Total Memory Rows
non-optimised SI	2x2	32	76	0	108
	4x4	128	304	0	432
	8x8	512	1216	0	1728
	16x16	2048	4864	0	6912
optimised SI	2x2	32	60	8	100
	4x4	128	240	32	400
	8x8	512	960	128	1600
	16x16	2048	3840	512	6400

TABLE 4. Memory dimension, LiM density and total execution time of the LiM design produced by Octantis for the multi-image encryption algorithm proposed by Li, X et all [40].

Algorithm	Images Size	Memory Dimension [bits]	LiM Density	Execution Time [T_{clk}]
non-optimised SI	2x2	864	70.4%	5
	4x4	3456	70.4%	5
	8x8	13824	70.4%	5
	16x16	55296	70.4%	5
optimised SI	2x2	800	68,0%	5
	4x4	3200	68,0%	5
	8x8	12800	68,0%	5
	16x16	51200	68,0%	5

indicated in the reference paper. In order to observe the effects of the optimisation strategy described in Algorithm 5, multiple syntheses have been run with different images sizes, namely 2×2, 4×4, 8×8 and 16×16. For each test case, two syntheses have been performed. The first has been run disabling the mentioned optimisation, generating designs referenced as *non-optimised SI* LiM architectures, where *SI* stands for *Scrambled Images*. The second has been issued enabling the optimisation strategy of reference, and the synthesised designs are referenced as *optimised SI* LiM architectures. Hence, 8 LiM design have been generated by Octantis. It is worth noting that, regardless of the sizes, each image's processing is independent of the others, allowing for parallel execution and efficient utilisation of a Logic-in-Memory implementation.

Results in Table 2 show that the optimisation technique allowed achieving a 7.4% reduction in the total amount of memory rows by integrating 2-to-1 multiplexers for all input images size cases. In Table 3, a detailed breakdown of the memory composition for both *optimised SI* and *non-optimised SI* LiM architectures is provided. Similarly to the previous algorithm, the memory rows are classified based on the type of integrated logic they feature. In this benchmark, "Xor with 2-to-1 mux" rows are synthesised, and they feature an XOR gate with one input from a 2-to-1 1-bit multiplexer. The optimisation strategy was capable of substituting several "Xor" LiM rows with "Xor with 2-to-1 mux" ones, thus enabling their reuse. As a consequence, the number of memory rows equipped with logic decreased, as well as the total amount of memory rows.Table 4 also reports memory dimension and LiM density metrics. The latter is an indicator of the produced architecture's compactness, and it is calculated as the ratio between logic-equipped rows and simple memory ones. As shown in Table 4, a slight decrease in LiM density has occurred for *optimised SI* designs. However, the overall memory dimension reduction is more significant. As regards the overall execution time, for all the different test cases, it is equal to 5 T_{clk}, due to the need for five XOR operations to calculate the (i, j) pixel of the XOR-Image. Furthermore, it is interesting to notice that the same memory reduction is achieved for all the different images size cases. This consistency can be attributed to the optimisation process identifying the same subset of "Xor" rows to be equipped with 2-to-1 multiplexers in each case. Hence, the linear growth of this subset throughout the cases led to the same reduction for all of them. As a result, measures adopted by Octantis synthesis flow have proved to provide an optimal solution regarding the area

TABLE 5. Memory composition of the LiM design produced by Octantis for the summed area table algorithm [41].

Algorithm	Images Size	Simple Memory Rows	Add Rows	Total Memory Rows
non-optimised SAT	2x2	8	5	13
	4x4	65	84	149
	8x8	705	1232	1937
	16x16	9473	18240	27713
optimised SAT	2x2	8	4	12
	4x4	32	33	65
	8x8	133	217	350
	16x16	564	1313	1877

TABLE 6. Memory dimension, LiM density and total execution time of the LiM design produced by Octantis for the summed area table algorithm [41].

Algorithm	Images Size	Memory Dimension [bits]	LiM Density	Execution Time [T_{clk}]
non-optimised SAT	2x2	104	38.5%	2
	4x4	1192	56.4%	5
	8x8	15496	63.6%	7
	16x16	221704	65.8%	9
optimised SAT	2x2	96	33.3%	2
	4x4	520	50.8%	5
	8x8	2800	62.0%	7
	16x16	10504	70.0%	9

occupation while keeping complexity and execution time contained.

The last algorithm to be addressed is the *Summed Area Table (SAT)* algorithm. It is a pre-processing technique used to generate the *Integral Image*, which is a data structure where each pixel $P_{IM}(i, j)$ corresponds to the sum of all the pixels above and to the left of the same pixel in the input image $P_{input}(i, j)$. This algorithm has gained popularity due to its prominent use in the *Viola-Jones* object detection framework [41]. Since implementing the SAT algorithm needs many accumulations, it represents a good test case to evaluate the potential of Octantis optimisation on reduction trees.

An appropriate C code has been defined, and Octantis configuration file has been set with a parallelism of *8 bits*, considering that each pixel value is represented on 8 bits. Similar to the previous benchmark, various sizes for the input images have been selected, namely 2×2, 4×4, 8×8 and 16×16. Two syntheses have been issued for each of them to evaluate the effects of Octantis optimisation described in Algorithm 7. Results reported in Table 2 prove the effectiveness of the optimisation technique. It allowed achieving a reduction up to 93,2% in the total amount of memory rows. The memory composition details of the resulting 8 LiM architectures are shown in Table 5. As it can be noticed, the introduced optimisation has enabled significant reductions in the number of both simple and logic-equipped rows. At the same time, as reported in Table 6, it also allowed for increasing the LiM density, leading to a more compact design. As a consequence, a relevant reduction in the overall memory size has occurred. The overall execution time of the algorithm remains unchanged, as the optimisation technique aims to identify memory rows whose insertion can be avoided while keeping the execution time almost unaltered. Therefore, Octantis' optimisation techniques have revealed effective in improving the implementation of the SAT algorithm on a LiM architecture, without compromising its execution time.

Upon analysing the results obtained from the preliminary tests conducted on Image Processing applications, it can be concluded that the intrinsic characteristics of LiM are highly compatible with this type of algorithm, particularly due to its highly parallelisable nature. Octantis synthesis flow effectively recognises parallelisation opportunities, and combines them with the newly introduced hardware-oriented optimisation techniques. As highlighted throughout this section, the enhanced synthesis capabilities of the tool allow achieving significant reductions in area occupation while keeping the same overall execution time. This demonstrates the potential of Octantis in effectively handling Image Processing applications and, more in general, data-intensive algorithms.

C. SYNTHESIS OF OCTANTIS-GENERATED LIM ARCHITECTURES WITH SYNOPSYS DESIGN COMPILER

As previously discussed, Octantis synthesizes a technology-independent LiM architecture, providing designers with a VHDL description of it. Ultimately, a specific target technology is required to effectively implement these LiM units. Beyond-CMOS technologies are particularly promising for LiM architectures due to their intrinsic logic and memory capabilities. However, as discussed in section II, design automation tools for in-memory architecture available in literature mainly focus on logic synthesis for memristive arrays. Hence, tools capable of extracting area, power, and latency metrics for beyond-CMOS-based LiM architectures are currently lacking.

While Octantis aims to integrate with such tools in the future, commercial EDA synthesis tools can be used in the interim. This allows for the implementation of LiM

TABLE 7. Power and area metrics obtained with synopsys design compiler for the Octantis-generated VHDL design of the multi-image encryption algorithm in [39].

Algorithm	Images Size	Total Area $[um^2]$	Total Power $[uW]$
Huang, Z.J et all	2x2	9.2e+02	1.0e+02
	4x4	3.5e+03	3.8e+02
	8x8	1.4e+04	1.5e+03
	16x16	5.5e+04	5.9e+03
	32x32	2.2e+05	2.3e+04

TABLE 8. Power and area metrics obtained with synopsys design compiler for the Octantis-generated VHDL design of the multi-image encryption algorithm in [40].

Algorithm	Images Size	Total Area $[um^2]$	Total Power $[uW]$
non-optimised SI	2x2	8.8e+03	9.6e+02
	4x4	3.1e+04	3.3e+03
	8x8	1.2e+05	1.3e+04
	16x16	4.8e+05	5.1e+04
optimised SI	2x2	8.4e+03	9.2e+02
	4x4	3.0e+04	3.1e+03
	8x8	1.1e+05	1.2e+04
	16x16	4.5e+05	4.8e+04

TABLE 9. Power and area metrics obtained with synopsys design compiler for the Octantis-generated VHDL design of the summed area table algorithm [41].

Algorithm	Images Size	Total Area $[um^2]$	Total Power $[uW]$
non-optimised SAT	2x2	9.8e+02	1.0e+02
	4x4	1.2e+04	1.1e+03
	8x8	1.7e+05	1.4e+04
optimised SAT	2x2	8.8e+02	9.6e+01
	4x4	5.0e+03	4.8e+02
	8x8	2.8e+04	2.6e+03

architectures using standard CMOS technology, thereby enabling a preliminary evaluation of Octantis's synthesis capabilities. Consequently, the VHDL descriptions generated by Octantis for the algorithms have been synthesized using Synopsys' Design Compiler with the 45 nm Nangate open-cell library.

For the *multi-image encryption algorithm* showcased in [39], Octantis generated a VHDL description for each LiM design of image sizes ranging from 2×2 to 256×256 pixels. However, only the ones corresponding to sizes from 2×2 up to 32×32 were synthesized by imposing a 100 MHz clock frequency, extracting power and area metrics. As expected, both area and power increase with larger input images, as reported in Table 7.

As regards the *multi-image encryption algorithm* presented in [40], the VHDL designs produced by Octantis for both *non-optimised SI* and *optimised SI* architectures have been fed to Synopsys Design Compiler imposing a 100 MHz clock frequency. Power and area metrics have been obtained and they are reported in Table 8. Moreover, Table 10 displays percentage reductions obtained in these metrics. Results show that the optimised SI LiM architectures demonstrates up to 6% savings in both area and power compared to the non-optimised counterparts. Moreover, it is interesting to highlight how area and power percentage savings remain nearly unchanged for all input images sizes, as happened for memory reduction in the previously discussed Table 4. Hence, this confirms that the effectiveness of the introduced optimisation strategy does not depend on the input image size.

The same procedure has been applied to the *Summed Area Table (SAT)* algorithm. Octantis generated VHDL designs for both *non-optimised SAT* and *optimised SAT* LiM architectures with input image sizes 2×2, 4×4 and 8×8. Designs have been synthesized using Synopsys Design Compiler with a 100 MHz clock frequency, and the extracted area and power metrics are presented in Table 9. Due to the complexity of the synthesis process, results for larger image sizes were omitted as they do not influence the aims of this discussion. The objective of these tests is not to demonstrate the effectiveness of a LiM implementation using standard CMOS technology but to evaluate the impact of the optimisation strategy adopted by Octantis. The application of the accumulation optimisation strategy has resulted in significant savings, achieving up to 82,2% reduction in area and 82,0% in power consumption, as it can be noticed in Table 2. Differently from the previous benchmark, the strength of the introduced optimisation increased along with the dimension of the input image.

The evaluation conducted in this section demonstrates the capability of Octantis to produce a VHDL description for the synthesized LiM architectures that is suitable for standard CMOS technology implementation. Exploiting Synopsys' Design Compiler to synthesise Octantis-generated LiM designs made it possible to prove the effectiveness of the

TABLE 10. Power and area reduction enabled by the optimisations strategies introduced within Octantis synthesis flow for algorithms proposed in [40] and [41].

Algorithm	Images Size	Area Reduction	Power Reduction
Li, X et all [40]	2x2	4.8%	4.7%
	4x4	6.2%	6.2%
	8x8	5.6%	5.9%
	16x16	6.0%	5.8%
Viola, P. and Jones, M. [41]	2x2	10.3%	7.7%
	4x4	58.5%	56.5%
	8x8	82.8%	82.0%

introduced optimisation strategies. These allowed to achieve significant reductions in both area occupation and power consumption. In conclusion, the obtained results allowed assessing the quality of Octantis synthesis flow and its optimisation strategies.

V. DISCUSSION AND CONCLUSION

The tests conducted are aimed at verifying the methodology adopted by Octantis and its recent expansions. The results described in this paper are necessary to validate the produced architectures with respect to the reference LiM topology and, at the same time, demonstrate the validity of the exploratory approach implemented on specific application fields. As previously mentioned, the LiM principle addresses performance enhancement when running data-intensive algorithms. The more an algorithm is parallelisable and composed of binary operations, the more it can benefit from a LiM implementation. Among these, some ImP algorithms exhibit characteristics that suggest a profitable execution through computational units so conformed.

Octantis synthesis capabilities have proved effective in identifying parallelisation opportunities and applying hardware-oriented optimisation techniques. In particular, these optimisation strategies allowed for achieving better LiM designs in terms of area occupation without compromising execution performance. As Octantis provides a VHDL description of the generated LiM architectures, syntheses using Synopsys' Design Compiler with the 45nm Nangate open-cell library have been successfully carried out to further validate the effectiveness of Octantis. The generated LiM architectures exhibited significant reductions in both area occupation and power consumption, thus highlighting the importance of the newly introduced optimisation techniques. As pointed out during the discussion, these findings represent preliminary results, as the tool focuses on exploring innovative implementation technologies, and further analyses will be conducted in this direction in the future.

The results obtained have proved to be promising and of interest for further investigation. It is important to highlight how the quality of the solutions provided by Octantis must be assessed in a subsequent phase of the exploration process, as well as the choice of the target technology to be considered, which broadens the exploration prospects to above-CMOS solutions. To emphasise the importance of this approach at the design stage, it should be noted that it is possible to configure the behaviour of Octantis and introduce several constraints to guide the synthesis process.

The tool has been made available in its first release to share all the details about the implementation choices adopted to develop the LiM explorer and open up the possibility for external contributions. Nonetheless, the same modules can be considered to derive other similar applications.

Octantis is constantly growing, experimenting with more complex design strategies and advanced optimisations. For instance, polyhedral analyses and optimisation techniques enabled by Polly [42] within the LLVM framework are being considered. Octantis parallelisation and synthesis capabilities could highly benefit from the related analysis and optimisation strategies. The program allows a designer to explore beyond von Neumann's solutions to tackle everyday problems and seek a glimpse into a possible future for electronic computing devices.

REFERENCES

[1] (2018). *2018 Edition of International Roadmap for Devices and Systems (IRDS)*. [Online]. Available: https://irds.ieee.org/editions/2018

[2] M. D. Godfrey and D. F. Hendry, "The computer as von Neumann planned it," *IEEE Ann. Hist. Comput.*, vol. 15, no. 1, pp. 11–21, Aug. 1993.

[3] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "Processing data where it makes sense: Enabling in-memory computation," *Microprocessors Microsyst.*, vol. 67, pp. 28–41, Jun. 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933118302291

[4] J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," *Commun. ACM*, vol. 21, no. 8, pp. 613–641, Aug. 1978, doi: 10.1145/359576.359579.

[5] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "TOP-PIM: Throughput-oriented programmable processing in memory," in *Proc. 23rd Int. Symp. High-Perform. Parallel Distrib. Comput. (HPDC)*. New York, NY, USA: Association for Computing Machinery, Jun. 2014, pp. 85–98, doi: 10.1145/2600212.2600213.

[6] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proc. ACM/IEEE 42nd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2015, pp. 105–117, doi: 10.1145/2749469.2750386.

[7] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *Proc. 53rd ACM/EDAC/IEEE Design Autom. Conf. (DAC)*. New York, NY, USA: Association for Computing Machinery, Jun. 2016, pp. 1–6, doi: 10.1145/2897937.2898064.

[8] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, in -50 '17. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 273–287.

[9] M. Imani, S. Gupta, and T. Rosing, "Ultra-efficient processing in-memory for data intensive applications," in *Proc. 54th Annu. Design Autom. Conf. (DAC)*. New York, NY, USA: Association for Computing Machinery, Jun. 2017, pp. 1–6, doi: 10.1145/3061639.3062337.

[10] S. Angizi, Z. He, F. Parveen, and D. Fan, "RIMPA: A new reconfigurable dual-mode in-memory processing architecture with spin Hall effect-driven domain wall motion device," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, Jul. 2017, pp. 45–50.

[11] J. Chen, W. Zhao, Y. Wang, Y. Shu, W. Jiang, and Y. Ha, "A reliable 8T SRAM for high-speed searching and logic-in-memory operations," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 30, no. 6, pp. 769–780, Jun. 2022.

[12] D. Fan and S. Angizi, "Energy efficient in-memory binary deep neural network accelerator with dual-mode SOT-MRAM," in *Proc. IEEE Int. Conf. Comput. Design (ICCD)*, Nov. 2017, pp. 609–612.

[13] G. Santoro, G. Turvani, and M. Graziano, "New logic-in-memory paradigms: An architectural and technological perspective," *Micromachines*, vol. 10, no. 6, p. 368, May 2019, doi: 10.3390/mi10060368.

[14] A. Marchesin and A. Naclerio, *VLSI-Nanocomputing/Octantis: Version 1.0.0*. Genève, Switzerland: Zenodo, 2023, doi: 10.5281/zenodo.10017506.

[15] A. Coluccio, M. Vacca, and G. Turvani, "Logic-in-memory computation: Is it worth it? A binary neural network case study," *J. Low Power Electron. Appl.*, vol. 10, no. 1, p. 7, Feb. 2020.

[16] M. Andrighetti, G. Turvani, G. Santoro, M. Vacca, A. Marchesin, F. Ottati, M. Ruo Roch, M. Graziano, and M. Zamboni, "Data processing and information classification—An in-memory approach," *Sensors*, vol. 20, no. 6, p. 1681, Mar. 2020, doi: 10.3390/s20061681.

[17] A. Coluccio, U. Casale, A. Guastamacchia, G. Turvani, M. Vacca, M. R. Roch, M. Zamboni, and M. Graziano, "Hybrid-SIMD: A modular and reconfigurable approach to beyond von Neumann computing," *IEEE Trans. Comput.*, vol. 71, no. 9, pp. 2287–2299, Sep. 2022.

[18] A. Marchesin, G. Turvani, A. Coluccio, F. Riente, M. Vacca, M. R. Roch, M. Graziano, and M. Zamboni, "Octantis: An exploration tool for beyond von Neumann architectures," in *Proc. 16th Int. Conf. Design Technol. Integr. Syst. Nanosc. Era (DTIS)*, Jun. 2021, pp. 1–5.

[19] A. Takach, "High-level synthesis: Status, trends, and future directions," *IEEE Des. Test.*, vol. 33, no. 3, pp. 116–124, Jun. 2016.

[20] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Design Test Comput.*, vol. 26, no. 4, pp. 18–25, Jul. 2009, doi: 10.1109/MDT.2009.83.

[21] Xilinx. (2024). *Vitis Software Platform*. [Online]. Available: https://www.xilinx.com/products/design-tools/vitis.html

[22] S. D. I. Softw. (2022). *Catapult High-level Synthesis and Verification*. [Online]. Available: https://eda.sw.siemens.com/en-U.S./ic/catapult-high-level-synthesis

[23] Intel. (2024). *Intel High Level Synthesis Compiler*. [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html

[24] H. Almorin, B. Le Gal, J. Crenne, C. Jego, and V. Kissel, "High-throughput FFT architectures using HLS tools," in *Proc. 29th IEEE Int. Conf. Electron., Circuits Syst. (ICECS)*, Oct. 2022, pp. 1–4.

[25] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 4, pp. 473–491, Apr. 2011.

[26] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 10, pp. 1591–1604, Oct. 2016.

[27] Y. Guo, D. McCain, J. R. Cavallaro, and A. Takach, "Rapid industrial prototyping and SoC design of 3G/4G wireless systems using an HLS methodology," *EURASIP J. Embedded Syst.*, vol. 2006, pp. 1–25, Dec. 2006.

[28] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," *ACM Trans. Embedded Comput. Syst.*, vol. 13, no. 2, pp. 1–27, Sep. 2013.

[29] C. Pilato and F. Ferrandi, "Bambu: A modular framework for the high level synthesis of memory-intensive applications," in *Proc. 23rd Int. Conf. Field Program. Log. Appl.*, Sep. 2013, pp. 1–4.

[30] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Design Test Comput.*, vol. 26, no. 4, pp. 8–17, Jul. 2009.

[31] F. Riente, U. Garlando, G. Turvani, M. Vacca, M. Ruo Roch, and M. Graziano, "MagCAD: Tool for the design of 3-D magnetic circuits," *IEEE J. Explor. Solid-State Comput. Devices Circuits*, vol. 3, pp. 65–73, 2017.

[32] F. Wang, G. Luo, G. Sun, J. Zhang, J. Kang, Y. Wang, D. Niu, and H. Zheng, "STAR: Synthesis of stateful logic in RRAM targeting high area utilization," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 40, no. 5, pp. 864–877, May 2021.

[33] R. Ben-Hur, R. Ronen, A. Haj-Ali, D. Bhattacharjee, A. Eliahu, N. Peled, and S. Kvatinsky, "SIMPLER MAGIC: Synthesis and mapping of in-memory logic executed in a single row to improve throughput," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 10, pp. 2434–2447, Oct. 2020.

[34] D. Bhattacharjee, L. Amaru, and A. Chattopadhyay, "Technology-aware logic synthesis for ReRAM based in-memory computing," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 1435–1440.

[35] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "MAGIC—Memristor-aided logic," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 61, no. 11, pp. 895–899, Nov. 2014.

[36] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, Mar. 2004, pp. 75–86.

[37] L. D. Group. *Clang: A C Language Family Frontend for LLVM*. Accessed: Mar. 2024. [Online]. Available: https://clang.llvm.org/index.html

[38] M. Cofano, M. Vacca, G. Santoro, G. Causapruno, G. Turvani, and M. Graziano, "Exploiting the logic-in-memory paradigm for speeding-up data-intensive algorithms," *Integration*, vol. 66, pp. 153–163, May 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016792601830556X

[39] Z.-J. Huang, S. Cheng, L.-H. Gong, and N.-R. Zhou, "Nonlinear optical multi-image encryption scheme with two-dimensional linear canonical transform," *Opt. Lasers Eng.*, vol. 124, Jan. 2020, Art. no. 105821. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0143816619305597

[40] X. Li, X. Meng, X. Yang, Y. Wang, Y. Yin, X. Peng, W. He, G. Dong, and H. Chen, "Multiple-image encryption via lifting wavelet transform and XOR operation based on compressive ghost imaging scheme," *Opt. Lasers Eng.*, vol. 102, pp. 106–111, Mar. 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0143816617307832

[41] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. (CVPR)*, vol. 1, Dec. 2001, pp. 1–11.

[42] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly—Performing polyhedral optimizations on a low-level intermediate representation," *Parallel Process. Lett.*, vol. 22, no. 4, Dec. 2012, Art. no. 1250010, doi: 10.1142/s0129626412500107.

ANDREA MARCHESIN (Member, IEEE) received the B.Sc. and M.Sc. degrees in electronic engineering from Politecnico di Torino, Turin, Italy, in 2018 and 2020, respectively, where he is currently pursuing the Ph.D. degree in electrical, electronics and communications engineering.

He is involved in research projects on both quantum computing algorithms and platforms and advanced logic-in-memory architectures. His research interests include quantum world, digital designs, and CAD tools development for the exploration of innovative electronic systems.

ALESSIO NACLERIO (Graduate Student Member, IEEE) received the B.Sc. and M.Sc. degrees in electronic engineering from Politecnico di Torino, Turin, Italy, in 2019 and 2022, respectively, where he is currently pursuing the Ph.D. degree in electrical, electronics and communications engineering.

His research interests include the investigation of beyond Von-Neumann architectures and the development of techniques for automating the mapping of suitable algorithms on such architectures.

FABRIZIO RIENTE (Member, IEEE) received the M.Sc. degree (magna cum laude) in electronic engineering and the Ph.D. degree from Politecnico di Torino, in 2012 and 2016, respectively.

He was a Postdoctoral Research Associate with the Technical University of Munich, in 2016. He is currently a Postdoctoral Research Associate with the Politecnico di Torino. His primary research interests include device modeling and circuit design for nano-computing, with a particular interest in magnetic QCA. His interests also cover the development of EDA tools for beyond-CMOS technologies, with the main focus on physical designs.

MARIAGRAZIA GRAZIANO received the D.Eng. and Ph.D. degrees in electronics engineering from Politecnico di Torino, Italy, in 1997 and 2001, respectively.

Since 2002, she has been an Assistant Professor with Politecnico di Torino. Since 2008, she has been an Adjunct Faculty Member with the University of Illinois at Chicago (UFL), Chicago, IL, USA. Since 2014, she has also been a Marie-Curie Fellow with London Centre for Nanotechnology. She works beyond CMOS devices, circuits, and architectures for traditional and quantum processing systems.

Open Access funding provided by ‘Politecnico di Torino’ within the CRUI CARE Agreement