POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Modeling and Exploration of AI Accelerators based on Digital In-Memory-Computing

Tesi di Laurea Magistrale in
Computer Science and Engineering
Ingegneria Informatica

Author: **Valeria de Gennaro**

Student ID: 251450
Advisor: Prof. Cristina Silvano
Co-advisors: Marco Ronzani, Cristian Zambelli
Academic Year: 2024-2025

Abstract

The design and optimization of AI hardware accelerators are becoming increasingly challenging. Modern deep learning models are growing larger and more complex, requiring massive computational resources. At the same time, edge devices impose strict constraints on energy consumption, latency, and physical resources. These opposing factors create a growing need for innovative solutions that can balance model complexity with the limitations of edge hardware.

Among the emerging solutions, In-Memory Computing (IMC) has gained significant attention thanks to its intrinsic compatibility with convolution operations, which enables efficient multiply–accumulate (MAC) acceleration. By performing computations directly within the memory arrays, IMC effectively mitigates the data transfer bottleneck between memory and processing units — a major limitation of Von Neumann architectures — while exploiting the high data reuse opportunities of convolutional workloads. Modeling and evaluating such accelerators to minimize energy consumption and latency while staying within strict memory and computational resource budgets remains a key research problem addressed by recent state-of-the-art studies.

This thesis proposes a methodology for design space exploration (DSE) of neural network accelerators based on IMC technology, with a specific focus on convolutional workloads. The proposed framework integrates and explores existing algorithms and methodologies — enhanced with dedicated optimizations — to efficiently identify a set of Pareto-optimal design configurations that balance performance, energy, and resource utilization across multiple workloads under given system constraints. Experimental evaluations reveal an effective evolutionary process that identifies configurations with an optimal energy–latency balance relative to other comparable architectures.

Keywords: design space exploration, in-memory computing, convolutional neural networks, hardware accelerators

Abstract in lingua italiana

La progettazione e l'ottimizzazione degli acceleratori per AI stanno diventando sempre più complesse. I moderni modelli di deep learning stanno infatti crescendo in dimensioni e complessità, richiedendo risorse computazionali sempre più elevate. Allo stesso tempo, i dispositivi edge impongono vincoli stringenti in termini di consumo energetico, latenza e risorse fisiche disponibili. Questi fattori contrastanti generano una crescente necessità di soluzioni innovative in grado di bilanciare la complessità dei modelli con le limitazioni dell'hardware edge.

Tra le soluzioni emergenti, l'In-Memory Computing (IMC) ha attirato particolare attenzione grazie alla sua intrinseca compatibilità con le operazioni convoluzionali, che consente un'efficiente accelerazione delle operazioni di multiply–accumulate (MAC). Eseguendo i calcoli direttamente all'interno delle matrici di memoria, l'IMC riduce in modo significativo il collo di bottiglia legato al trasferimento dei dati tra memoria e unità di calcolo — una delle principali limitazioni delle architetture di tipo Von Neumann — sfruttando al contempo le elevate opportunità di data reuse proprie dei workload convoluzionali.

La modellazione e la valutazione di tali acceleratori, con l'obiettivo di minimizzare il consumo energetico e la latenza rispettando vincoli rigorosi in termini di memoria e risorse computazionali, rappresentano tuttora una sfida di ricerca centrale affrontata dagli studi più recenti state-of-the-art.

Questa tesi propone una metodologia di Design Space Exploration (DSE) per acceleratori di reti neurali basati su tecnologia IMC, con un focus specifico sui workload convoluzionali. Il framework proposto integra e adatta algoritmi e metodologie esistenti — arricchiti da ottimizzazioni dedicate — per identificare in modo efficiente un insieme di configurazioni Pareto-optimal in grado di bilanciare performance, consumo energetico e utilizzo delle risorse su molteplici workload, nel rispetto dei vincoli imposti dal sistema. Le valutazioni sperimentali mostrano un processo evolutivo efficace, capace di individuare configurazioni con un bilanciamento ottimale energy–latency rispetto ad altre architetture comparabili.

Parole chiave: esplorazione dello spazio di design, computazione in memoria, reti neurali convoluzionali, acceleratori hardware

Contents

Abstract **i**

Abstract in lingua italiana **iii**

Contents **v**

1 Introduction **1**
1.1 Motivations 1
1.2 Objectives 2
1.3 Organization 3

2 Background **5**
2.1 Deep Neural Networks 5
2.1.1 Convolutional Neural Networks 5
2.2 In Memory Computing 9
2.2.1 Motivations for Reducing Data Movement 9
2.2.2 Digital In-Memory Computing 11
2.3 Design Space Exploration 13
2.3.1 The concept of Pareto curve 14
2.3.2 Design of Experiment 14
2.3.3 Design and Analysis of Computer Experiments 15
2.3.4 Techniques for Exploration 19
2.3.5 NSGA-II: an elitist multiobjective EA 23

3 Related works **31**
3.1 Modeling Tools 32
3.1.1 Digital In-Memory Computing (DIMC) Modules 32
3.1.2 CimLoop 35
3.2 Mapping Tools 38

3.2.1 Timeloop 38
3.2.2 FactorFlow 39
3.3 Exploration Tools 41
3.3.1 DOSA 41
3.3.2 Di Gamma 42

4 Proposed In-Memory Computing-based Methodology 45
4.1 Overview 45
4.2 Template Architectures 45
4.2.1 IMC Modeling and Characterization. 47
4.3 Design Space Exploration Process 55
4.3.1 Problem Formulation and Initial Setup 55
4.3.2 Parameter Space Definition 57
4.3.3 Sampling Strategy: Latin Hypercube 60
4.3.4 Architecture Configurations and Initial Population 61
4.3.5 GPR Model Training 62
4.4 Multi-Objective Evolutionary Exploration Algorithm 65
4.4.1 Evolutionary Process 67
4.4.2 Iteration and Termination 71
4.4.3 Optional Optimizations 72
4.5 Implementation 75
4.5.1 Programming Environment and Framework Integration 76
4.5.2 Code Architecture and Workflow 76
4.5.3 Configuration Management: The `Settings` Class 78
4.5.4 Design Choices and Multiprocessing Strategy 78
4.5.5 Conclusions and Limitations 79

5 Experimental Results 81
5.1 Experimental Setup 81
5.1.1 Comparison Algorithms 81
5.1.2 Exploration Metrics 83
5.1.3 Parameter Configurations 87
5.2 Results 89
5.2.1 Basic Homogeneous Exploration 89
5.2.2 Exploitation vs Exploration 98
5.2.3 Re-Training Optimization 104
5.3 Quantitative Evaluation of Improvements Against the Baseline 115

6 Conclusions **117**
6.1 Summary 117
6.2 Future Work 118

Bibliography **121**

List of Figures **127**

List of Tables **129**

List of Symbols **131**

Acknowledgements **133**

1 | Introduction

1.1. Motivations

In recent years, the rapid expansion of Artificial Intelligence (AI) has profoundly transformed numerous application domains, including computer vision, natural language processing, and autonomous systems. Among the various AI paradigms, deep learning has emerged as one of the most powerful approaches, enabling models to perform complex tasks previously considered beyond the capabilities of traditional computing systems. In particular, the rise of Generative AI and Transformer-based architectures has further accelerated progress by enabling models to generate, understand, and manipulate complex data with unprecedented accuracy [10, 32]. A key milestone in this revolution has been the development of image recognition models, which allow artificial systems to learn visual patterns directly from raw pixel data. These models are capable of identifying and extracting relevant features from images, forming the foundation for tasks such as classification, detection, and segmentation. At the core of such models lies the process of feature extraction, which is efficiently realized through the use of convolutional operations [9]. This process can hierarchically capture and represent features across multiple layers.

The fundamental computational unit underlying deep learning is the multiply–accumulate (MAC) operation. In convolutional neural networks, these operations exhibit a high degree of data reuse, as the same convolution kernel is applied repeatedly across the input feature maps. However, despite this inherent regularity, the acceleration of such workloads remains a significant challenge in hardware design. The primary limitation of conventional von Neumann architectures lies in the memory–computation bottleneck [20], where the frequent data transfers between memory and processing units significantly impact both energy consumption and latency. This issue becomes even more critical in convolutional workloads, which repeatedly access the same operands, amplifying the cost of data movement.

Recently, In-Memory Computing (IMC) has emerged as a promising paradigm to overcome this bottleneck [7, 13, 31]. By enabling computation directly within memory arrays,

IMC architectures minimize data movement and exploit the intrinsic data reuse of MAC operations. Each IMC cell is capable of storing a given operand while simultaneously participating in the computation with the corresponding input data. This architectural innovation redefines the design of neural accelerators, offering substantial improvements in energy efficiency and computational throughput. Recent studies have demonstrated both analog and digital IMC implementations, highlighting trade-offs in accuracy, scalability, and energy efficiency across different workloads [31]. In particular, digital SRAM-based IMC architectures have demonstrated scalability and compatibility with modern semiconductor technologies, enabling on-chip integration in neural network accelerators, achieving impressive energy efficiency. [7].

Nevertheless, the design of IMC-based accelerators introduces challenges related to architectural optimization. Even under constrained design conditions—where memory, energy, and computational budgets must be respected—the exploration of possible design configurations remains a vast and complex task. It is often impractical to perform an exhaustive search of all configurations due to the exponential growth of the design space. For this reason, methodologies such as Design Space Exploration (DSE) and Design of Experiments (DoE) have been introduced to guide the search process efficiently. These methods aim to identify Pareto-optimal trade-offs among conflicting objectives, such as minimizing energy consumption while reducing latency and respecting hardware constraints. Tools such as CiMLoop [2] have recently emerged to support accurate and scalable modeling of IMC-based systems, highlighting the growing importance of systematic exploration approaches in this field.

Exploring novel DSE methodologies tailored to IMC-based accelerators, particularly for convolutional workloads, therefore represents a valuable research challenge. Such an approach not only enhances the understanding of IMC architectural behavior but also provides practical strategies for the efficient design of next-generation AI hardware accelerators.

1.2. Objectives

This thesis focuses on integrating IMC-based accelerators within an existing evaluation and mapping framework, followed by the development of an efficient methodology for DSE in this domain, with a specific emphasis on convolutional workloads. The thesis revisits existing mono-objective and multi-objective evolutionary approaches, adapting and extending them to this specific context by incorporating optimization and randomization strategies.

The main contributions of this thesis can be summarized as follows:

1. Literature review and analysis of the current state of the art on accelerators based on In-Memory Computing technology and related Design Space Exploration methodologies, identifying their strengths, limitations, and open challenges relevant to AI accelerator design.
2. Modeling and characterization of a configurable digital SRAM-based IMC-module suitable for AI accelerators.
3. Integration of the SRAM-based IMC-module in a new template architecture for the FactorFlow existing mapping framework, ensuring seamless interoperability and consistency with the original spatial architecture.
4. Definition of the design space parameters of the template architecture to be explored and related optimization metrics.
5. Development of a multi-objective evolutionary algorithm (MOEA)-based exploration approach, leveraging genetic algorithms (GAs) and DoE techniques to identify Pareto-optimal design solutions within constrained design spaces. The proposed approach encompasses several optimization and adaptation mechanisms designed to improve the exploration-exploitation balance and accelerate convergence toward high-quality solutions.
6. Experimental comparison of different algorithmic configurations and optimization strategies, including randomized and baseline approaches.
7. Experimental evaluation to demonstrate the effectiveness of the proposed exploration methodology to identify Pareto-optimal configurations of the IMC-based template architecture to execute Deep Learning workloads. The results quantify the improvements to conventional IMC-baseline designs in terms of energy efficiency and latency.

1.3. Organization

- **Chapter 1** (this chapter) presents the motivations and objectives behind the work, along with an overview of its organization.
- **Chapter 2** introduces the necessary background to understand the concepts of CNNs, IMC, and DSE, discussing their main characteristics, challenges, and goals. It also describes the algorithms that serve as the foundation of the proposed methodology, including GAs and MOEAs.

- **Chapter 3** reviews key existing works related to mapping, modeling, and Design Space Exploration problems, providing a comprehensive overview of the current state of the art.
- **Chapter 4** details the proposed In-Memory Computing-based DSE methodology. It begins with the description of a new IMC-based template architecture and its integration in the FactorFlow existing mapping framework, followed by the modeling and characterization of the SRAM-based IMC-module used for the described architecture. It continues with the description of the overall methodological setup and the evolutionary process at the core of the approach. Each stage of the process, along with the corresponding evaluation metrics, is thoroughly analyzed.
- **Chapter 5** presents the experimental comparison and evaluation of the proposed approach, including various configurations using the previously defined metrics and the method's performance in comparison to other algorithms, highlighting its effectiveness.
- **Chapter 6** concludes the thesis by summarizing the main contributions, discussing the key results, and outlining potential directions for future developments and optimizations.

2 | Background

2.1. Deep Neural Networks

At this point in history, the use of Artificial Intelligence (AI) has become widespread, driven by the rapid development and diffusion of numerous models across different domains, architectures, and purposes. As is well known, deep learning systems are based on neural networks (NNs) —a revolutionary concept in computer science that enables machines to learn from data and, to some extent, emulate human cognitive behavior.

It all began with the development of Machine Learning (ML) models — a class of algorithms designed to enable computers to learn patterns from data and make predictions or decisions without being explicitly programmed for specific tasks. The fundamental goal of ML is to discover a functional mapping between input data and the desired output, allowing the model to generalize from examples and perform accurately on unseen data. Over time, various learning paradigms such as supervised, unsupervised, and reinforcement learning have been proposed, each addressing different types of problems and data availability scenarios.

The growing complexity of tasks and the increasing availability of large datasets have gradually led to the emergence of Deep Learning (DL), a subfield of ML that leverages deep neural networks (DNNs) composed of multiple layers of interconnected neurons. These architectures are capable of automatically extracting hierarchical features from raw data, significantly improving performance in domains such as computer vision, natural language processing, and speech recognition. The success of deep learning has been largely driven by advances in hardware acceleration, algorithmic optimization, and the availability of massive amounts of data. For this reason, the optimization and acceleration of DNNs remains active areas of research and technological innovation.

2.1.1. Convolutional Neural Networks

One of the most prominent application domains of AI, and consequently DL, involves image-related tasks — including image modification, processing, recognition, and even

generation.

In the context of image processing, Convolutional Neural Networks (CNNs) represent one of the most widely adopted architectures within deep learning. Given a raw image as input, the network processes it to extract relevant features necessary for both training and inference through specialized layers known as convolutional layers.

The convolution is a mathematical operation that, given two functions, produces a third function as the integral of the product of the two functions after one is reflected about the y-axis and shifted. In deep learning, this concept is implemented by sliding a small window —called a **kernel** or **filter**— across the input data and computing the scalar (dot) product at each position. The result of this process is a **feature map**, which highlights specific local patterns in the input, such as edges, textures, or color gradients. This approach allows the network to extract increasingly complex and abstract features at deeper levels. Early layers tend to capture simple patterns, while deeper layers encode higher-level semantic information. The convolutions are followed by normal Neural Networks layers, performing the usual tasks.

From a computational perspective, convolutional operations are particularly demanding because they involve a large number of Multiply-and-Accumulate (MAC) operations and frequent memory accesses. An interesting aspect is that this computational cost can be directly estimated from the dimensions of the input and the convolutional kernels. Most commonly, CNNs are designed to process RGB images, which consist of three input channels corresponding to the red, green, and blue color components. Each convolutional kernel must therefore operate across all input channels simultaneously.

For this purpose, let's define the main dimensions involved in a convolutional operation:

- **Input Feature Map (IFM)**: represented with height **H**, width **W**, and number of channels **C**. For example, an RGB image has $C = 3$.
- **Filter (Kernel)**: there are typically **M** filters, each characterized by a height **R**, width **S**, and the same number of channels as the input (**C**).
- **Output Feature Map (OFM)**: the output consists of **M** feature maps, one per filter. Each output feature map has a height **E** and width **F**, which depend on the input dimensions, the filter size, and the stride **U**.

Assuming a stride of U and no padding, the spatial dimensions of the output feature map can be computed as:

$$E = \frac{H - R}{U} + 1, \qquad F = \frac{W - S}{U} + 1$$

If padding P is applied, the equations generalize to:

$$E = \frac{H - R + 2P}{U} + 1, \qquad F = \frac{W - S + 2P}{U} + 1$$

It is now straightforward to compute the total number of **Multiply-and-Accumulate (MAC)** operations required for a single convolutional layer:

$$\text{MACs} = E \cdot F \cdot R \cdot S \cdot C \cdot M$$

This equation shows that the computational complexity grows linearly with the number of filters (M), the spatial dimensions of the output feature map (E and F), the kernel size ($R \times S$), and the number of input channels (C).

Estimating the number of MAC operations is fundamental to understanding the computational workload of a CNN layer, as well as the number of memory reads and writes required during execution. This estimation becomes particularly important when analyzing or designing hardware accelerators, since data movement and access patterns are directly influenced by these parameters.

To validate the effectiveness and generality of the proposed exploration methodology, the evaluation phase employs two of the most widely recognized CNNs architectures in literature: **ResNet-18** and **VGG-16**. Both models are benchmark references in the field of image classification and have been extensively used to assess the performance of hardware accelerators, thanks to their distinctive architectural characteristics and diverse computational requirements.

ResNet-18. Early attempts to improve accuracy by simply increasing network depth often led to diminishing returns due to vanishing gradients and overfitting, where deeper models failed to outperform their shallower counterparts. The introduction of the **Residual Network (ResNet)** architecture [9] addressed these limitations through a novel concept known as *residual connections*.

Residual connections enable the model to learn not the direct mapping between inputs and outputs, but the *residuals*—that is, the difference between the desired output and the input itself. This mechanism allows information and gradients to flow more effectively through the network by introducing *skip connections* that bypass one or more layers. As a result, deeper architectures can be trained efficiently without the degradation problems typically associated with increasing network depth.

ResNet-18, a member of the ResNet family, is composed of 18 layers, including convolutional, ReLU activation, and fully connected layers. Its design relies on a sequence of lightweight *building blocks*, each containing two convolutional layers and a shortcut connection, as illustrated in Figure 2.1. These residual blocks preserve feature information across layers and promote stable gradient propagation during both forward and backward passes. Although ResNet-18 is one of the smaller members of the ResNet family, it effectively demonstrates the advantages of residual learning while remaining computationally efficient and well-suited for hardware-based experimentation.

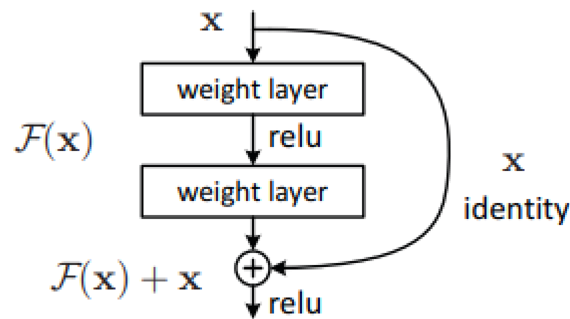


Figure 2.1: Example of a ResNet building block with residual (skip) connection.

VGG-16. Another landmark CNN architecture employed in this thesis is **VGG-16**, introduced by the Visual Geometry Group (VGG) at the University of Oxford [29]. VGG-16 is a deep CNN composed of 16 layers—13 convolutional and three fully connected—and was designed with an emphasis on simplicity and uniformity. Unlike many later architectures, VGG-16 employs small 3×3 convolutional kernels throughout the network, combined with max-pooling layers to reduce spatial dimensions while increasing feature depth progressively.

Despite its relatively straightforward structure, VGG-16 has demonstrated remarkable performance across various image classification tasks and continues to serve as a foundational model in computer vision research. Its consistent layer configuration and dense convolutional structure make it an ideal benchmark for evaluating accelerator architectures, as it imposes a substantial computational workload while maintaining predictable and regular dataflow patterns.

2.2. In Memory Computing

One of the major challenges in modern computing systems is the significant time and energy consumed by data transfers between memory and processing units. As the gap between processor speed and memory access latency continues to widen — a phenomenon often referred to as the memory wall [20] — the cost of moving data has become a dominant factor in overall system performance and efficiency. Traditional von Neumann architectures, which physically separate processing and memory units, exacerbate this issue due to the constant data movement across the memory hierarchy [13]. **In-memory computing (IMC)** emerges as a promising paradigm to mitigate this bottleneck by performing computations directly within or near the memory arrays, thereby drastically reducing data movement and improving both performance and energy efficiency.

2.2.1. Motivations for Reducing Data Movement

Hardware platforms designed to accelerate deep-learning workloads traditionally pursue a primary objective: maximizing computational speed. However, despite substantial progress in specialized accelerators, a critical bottleneck continues to undermine overall system efficiency—**the energy cost of data movement**. As discussed in [24], and effectively illustrated in Table2.1, the dominant contributor to total system energy is not the arithmetic computation itself, but rather the repeated transfer of data between processing units and off-chip memory.

This issue becomes particularly severe in deep-learning workloads, where tensor operations require frequent and often large-scale memory accesses. Even when computation is highly optimized or parallelized, each off-chip memory transaction incurs a significant energy penalty. As a result, it is the memory subsystem, rather than the compute engine, that becomes the primary driver of energy consumption.

Furthermore, modern neural-network layers, especially convolutional ones, exacerbate this imbalance due to their extensive reuse of weights and feature maps. Although this reuse improves computational efficiency, it simultaneously increases the pressure on the memory hierarchy. If data cannot be kept close to the compute units, the architecture must repeatedly fetch it from off-chip DRAM, dramatically increasing energy consumption. While hardware accelerators continue to advance in terms of raw computational speed, the **energy cost associated with memory transfers remains the dominant limiting factor**, motivating alternative paradigms.

Table 2.1: Energy per operation with a 45nm and 7nm technology node.
Credits to [12, 14].

Operation		PicoJoules per Operation	
		45nm	7nm
+	Int 8	0.03	0.007
	Int 32	0.1	0.03
	BFloat 16	–	0.11
	IEEE FP 16	0.4	0.16
	IEEE FP 32	0.9	0.38
×	Int 8	0.2	0.07
	Int 32	3.1	1.48
	BFloat 16	–	0.21
	IEEE FP 16	1.1	0.34
	IEEE FP 32	3.7	1.31
SRAM	64 bit Reg.	6.0	–
	8 KB SRAM	10	7.5
	32 KB SRAM	20	8.5
	1 MB SRAM	100	14
DRAM	DDR3/4	1300	1300
	HBM2	–	250-450
	GDDR6	–	350-480

The first classical computational paradigm found in the literature is the conventional **von Neumann architecture**. This foundational model comprises a processing element and an off-chip memory, connected via a high-bandwidth bus. It is inherently *processor-centric*: the central processing unit (CPU) acts as the master of the system and is traditionally regarded as the sole component responsible for executing computational tasks [20]. All auxiliary subsystems—such as memory and communication units—are treated as passive entities whose role is limited to storing or transporting data, requiring continuous transfers between memory and the processor. As a consequence, system performance and energy efficiency are strongly constrained by the memory–computation bottleneck.

Building upon this classical model, several heterogeneous architectures have been proposed to improve performance–power trade-offs, based on a host processor coupled with a hardware accelerator. As illustrated in Fig. 2.2[24], an early step beyond the von Neumann design introduces *near-memory computing* solutions, where non-volatile memory

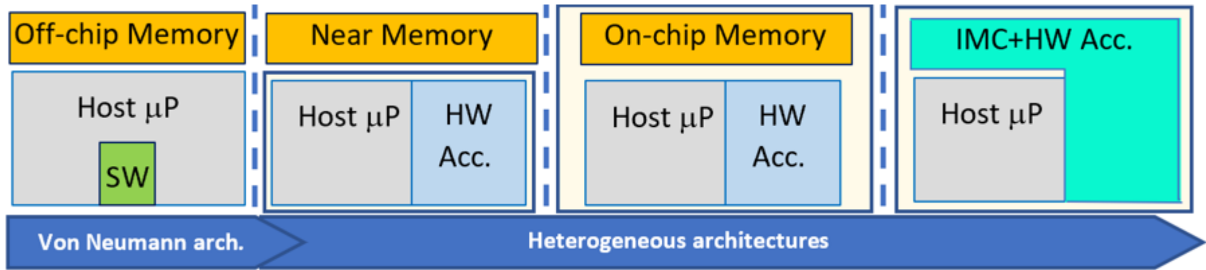


Figure 2.2: Evolution of memory–compute integration strategies. From left to right: (1) traditional von Neumann architectures with off-chip memory access, (2) near-memory computing using closely coupled DRAM/NVM modules, (3) on-chip memory integration with SRAM or NVM arrays, and (4) true In-Memory Computing (IMC), where computation is performed directly inside the memory array, minimizing data movement.

(NVM) modules are placed in proximity to the processing elements to reduce transfer overheads, enabling the storage of even MBytes of model parameters (weights and activations) closely coupled to the processing units. A further and more aggressive refinement is achieved through *on-chip memory integration*, typically based on SRAM or NVM arrays connected via low-latency, low-energy interconnect fabrics. This progression naturally leads to the emergence of **In-Memory Computing (IMC)** architectures—both analog and digital—which push computation **directly into the memory arrays** themselves. These IMC systems provide computational capabilities within the memory fabric, offering substantially higher energy efficiency compared to traditional and near-memory alternatives, which drastically reduces, and in some cases eliminates, data transfers to and from memory. In addition to reducing energy and latency, IMC architectures are particularly well-suited for data-intensive applications such as neural network inference and signal processing, where large volumes of data are repeatedly accessed and transformed.

For these reasons, the area where IMC technologies are most widely applied is machine learning and deep neural network (DNN) acceleration [17]. IMC architectures can exploit the inherent parallelism of memory arrays to perform MAC operations efficiently, allowing a large number of computations to be executed simultaneously. This high degree of parallelism enables IMC systems to achieve significant improvements in both performance and energy efficiency, particularly in data-intensive operations.

2.2.2. Digital In-Memory Computing

Over time, two main approaches to in-memory computing have been explored: analog and digital [31]. **Analog In-Memory Computing (AIMC)** exploits the high compu-

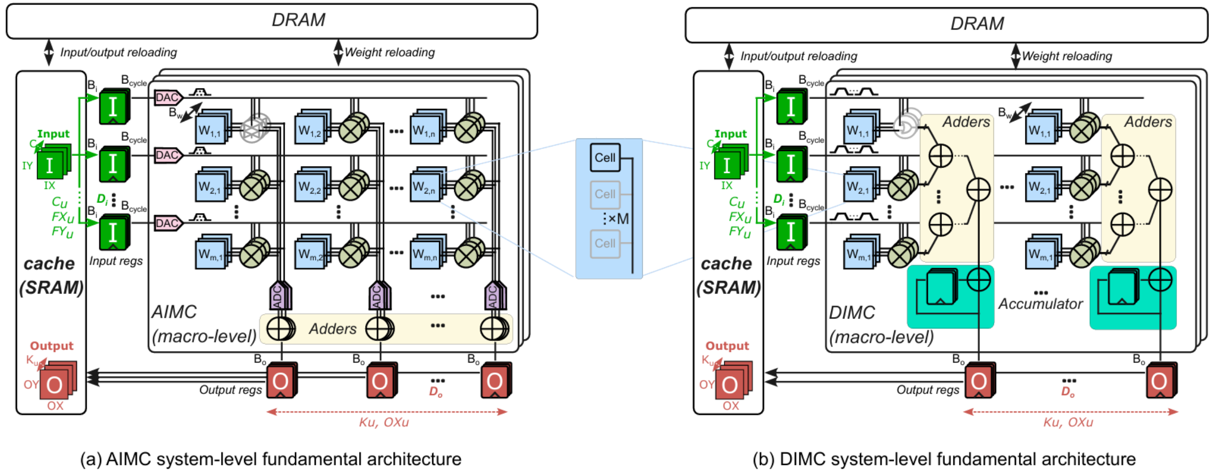
tational density and speed enabled by performing operations directly through physical quantities such as charge or current. This paradigm achieves extremely high energy efficiency and massive parallelism, making it well-suited for workloads dominated by vector-matrix multiplications. However, the analog nature of computation introduces several trade-offs: analog circuits are inherently sensitive to noise, process variations, and device mismatches, which can degrade accuracy and lead to unreliable outputs. Moreover, AIMC architectures typically require additional peripheral components—such as ADCs and DACs, as illustrated in Figure 2.3a—to convert between analog and digital domains. These conversion units introduce non-negligible energy and latency overheads, often limiting the practical efficiency gains of AIMC in real-world systems.

For these reasons, and despite a potential reduction in peak energy efficiency, fully digital implementations are often preferred. **Digital In-Memory Computing (DIMC)** removes the need for analog–digital signal conversion, enabling noise-free computation, deterministic behavior, and greater flexibility in spatial mapping and precision control. Its compatibility with standard digital design flows makes it particularly attractive for large-scale hardware accelerators [7, 16].

Digital IMC architectures, as described in Figure2.3b, typically rely on enhanced SRAM bit-cells—ranging from 6 to 12 transistors—that natively support element-wise multiplication, followed by a digital accumulation tree integrated alongside the memory array. Compared to analog approaches, DIMC offers several advantages:

- full CMOS compatibility and seamless integration into modern System-on-Chip (SoC) technologies, including advanced lithography nodes;
- increased design flexibility and straightforward integration with conventional digital CAD tools;
- predictable, high-precision computation without analog noise or conversion overhead;
- enhanced robustness against device variability, process noise, and thermal effects.

These characteristics have contributed to DIMC becoming the dominant in-memory computing paradigm for practical and scalable DNN acceleration systems.



(a) AIMC system-level fundamental architecture

(b) DIMC system-level fundamental architecture

Figure 2.3: System-level perspective of Digital and Analog IMC architecture.

2.3. Design Space Exploration

Beyond optimizing individual hardware components to handle the size and complexity of deep neural network (DNN) workloads, another major challenge lies in integrating these components into larger, more complex systems. The goal is to design a chip capable of executing the required tasks efficiently, while minimizing power consumption and adhering to limited hardware resources—constraints that are unavoidable in real-world scenarios. In essence, the problem can be formulated as identifying an optimal architecture that satisfies both resource constraints and energy efficiency objectives across different workloads.

A brute-force approach to this goal would involve exploring all possible architectures that meet the given constraints, evaluating each configuration, and selecting the optimal one. To practically execute such a process, several key components are required:

- **Generator:** a tool capable of systematically producing representations of all possible architectural configurations that satisfy the design constraints.
- **Evaluator:** a module responsible for assessing each generated architecture with respect to defined objectives, computing the necessary performance, energy, and resource utilization metrics.
- **Objective Function:** a mathematical function that expresses the design goals, such as minimizing latency or energy consumption, or maximizing throughput or efficiency.
- **Metric:** a quantitative criterion that allows comparison among evaluated architectures, enabling the selection of the best—or at least near-optimal—design according

to the objective function.

Although the brute-force approach described above may appear feasible in theory, the reality is quite different. In most practical scenarios, enumerating and generating all possible combinations of design parameters yields an enormous number of potential solutions—far too many to evaluate within a reasonable timeframe. For this reason, the concept of Design Space Exploration (DSE) emerged.

DSE refers to a collection of methodologies and tools designed to efficiently explore the design space and identify optimal or near-optimal solutions under given constraints. Rather than exhaustively evaluating every configuration, DSE frameworks employ various exploration strategies—such as heuristic, analytical, or machine learning-based approaches—to guide the search process toward the most promising architectural configurations [22, 27].

2.3.1. The concept of Pareto curve

The notion of an *optimal solution* in the context of DSE is not always straightforward. In theory, the optimal solution is the one that best satisfies or minimizes the objective function. However, in practice, design problems often involve multiple objectives—such as reducing energy consumption, latency, and area simultaneously. As a result, there may not be a single unique solution that optimizes all objectives at once. Instead, several solutions may be equally valid, each representing a different trade-off among the design goals.

To address this, DSE relies on the concept of the **Pareto curve** (or **Pareto front**), which represents the final set of optimal solutions according to the principle of *dominance*. A solution is said to be *dominated* by another if it is strictly worse in every objective. Conversely, a solution is considered *Pareto-optimal* if no other solution dominates it. The elements on the Pareto curve thus correspond to all non-dominated solutions, each offering a distinct and optimal trade-off between competing objectives.

2.3.2. Design of Experiment

To achieve the optimization that Design Space Exploration (DSE) aims to provide over traditional brute-force exploration, it is essential to optimize the selection of architectural configurations to be evaluated within the overall design space. The Design of Experiments (DoE) methodology offers several strategies to efficiently sample the design space, allowing the exploration process to consider only a representative subset of architectures instead of exhaustively evaluating every possible configuration [8].

Among the most commonly used DoE strategies are the Random DoE, Full, and Fractional Factorial DoE approaches.

Random DoE. This approach relies on random sampling of configurations within the design space. Candidate architectures are selected randomly, which helps reduce bias and provides a quick, lightweight estimation of performance trends. Random DoE is generally effective when the number of design parameters is limited or when the design space is relatively uniform, meaning that nearby configurations tend to exhibit similar performance characteristics. However, for larger and more complex design spaces, purely random selection may lead to uneven coverage or the omission of critical configurations.

Full Factorial DoE. In contrast to random sampling methods, the Full Factorial approach systematically explores the design space by evaluating every possible combination of parameter values across all relevant levels. This exhaustive exploration ensures complete and uniform coverage of the space, allowing for a comprehensive analysis of the effects and interactions of each design parameter. Full Factorial DoE is particularly valuable when dealing with design problems characterized by a moderate number of parameters, where interactions among factors can significantly influence the final objectives. However, its main limitation lies in the exponential growth of the number of configurations as the number of parameters or levels increases, which makes this method computationally expensive and often impractical for large-scale problems.

Fractional Factorial DoE. Developed as an extension of the Full Factorial approach, the Fractional Factorial Design of Experiments seeks to reduce computational cost by evaluating only a strategically selected subset of all possible combinations—literally, a **fraction** of the whole design space. This subset is chosen intelligently so that the most significant information can still be extracted: the main effects of the parameters and, in many cases, their low-order interactions can be accurately estimated without the need to evaluate every possible configuration. Fractional Factorial DoE thus provides an efficient balance between exploration completeness and computational feasibility, making it especially suitable for high-dimensional problems where exhaustive exploration would be prohibitive.

2.3.3. Design and Analysis of Computer Experiments

The original concepts of Design of Experiments (DoE) were developed for *physical experiments*, typically involving real-world systems affected by various sources of noise and uncertainty. Over time, as computational modeling and simulation became more prevalent, the need to apply DoE principles in the context of computer-based studies led to

the development of the **Design and Analysis of Computer Experiments (DACE)** framework. DACE can be viewed as an adaptation of traditional experimental design methods for computer simulations, where the "experiment" consists of running a deterministic computer model rather than conducting a physical test. Most literature in this area has focused on deterministic simulations—those that yield the same output when executed multiple times with identical inputs.

However, the three foundational principles of classical experimental design—*randomization*, *replication*, and *blocking*—lose much of their relevance in computer experiments. In this context, the proper relationship between the model inputs (the **parameters**) and outputs (the **predictions**) can be highly nonlinear and is often unknown. To approximate this relationship, surrogate models are typically employed. Among these, **Gaussian Process (GP)** predictors are widely used, as they perform well for deterministic functions and relatively small datasets.

Another essential consideration in DACE is the selection of data points used to train such surrogate models. Because prior knowledge about the proper function is usually limited, **space-filling designs** are preferred to ensure a uniform coverage of the input domain. These designs aim to sample the space as evenly as possible, improving the accuracy and generalization capability of the surrogate model. Common examples of space-filling designs include **Latin Hypercube Sampling (LHS)**, **Sobol sequences**, and **Halton sequences** [19].

Latin Hypercube Sampling

In many practical scenarios, space-filling designs must handle a large number of input variables. However, constructing designs that provide uniform coverage across a high-dimensional input space with only a limited number of design points is extremely challenging—if not impossible. A more feasible approach is to ensure good coverage in lower-dimensional projections of the design space.

To achieve this, **Latin Hypercube Sampling (LHS)** is commonly adopted. An LHS design consists of n runs and k factors, represented by an $n \times k$ matrix $\mathbf{D}$. Each column of $\mathbf{D}$ is a random permutation of n equally spaced levels in the interval $[0, 1)$. This guarantees that for each factor, every interval is sampled exactly once, preventing clustering and ensuring uniform coverage along each dimension[19].

The sampling process can be described as follows: for each factor j, the interval $[0, 1)$ is divided into n equally spaced subintervals. Then, one point is randomly selected within each subinterval according to the expression:

$$d_{ij} = \frac{l_{ij} + (n-1)/2 + u_{ij}}{n}, \quad i = 1, \ldots, n, \; j = 1, \ldots, k$$

where l_{ij} is the integer representing the level of the j-th factor for the i-th sample (after random permutation), and $u_{ij} \sim \mathcal{U}(0,1)$ is a random number uniformly distributed between 0 and 1. This formulation ensures that each sample lies within its assigned interval, thereby guaranteeing stratification and uniform coverage of the design space.

The popularity of Latin Hypercube Sampling (LHS) largely stems from its theoretical justification for variance reduction in numerical integration. Consider a function $y = f(\mathbf{x})$, where $\mathbf{x} = (x_1, \ldots, x_k)$ is a k-dimensional vector whose components are uniformly distributed over the unit hypercube $[0,1)^k$, and $y \in \mathbb{R}$. The goal is to estimate the expected value of y, which is expressed as an integral.

$$E[y] = E[f(\mathbf{x})] = \int_{[0,1)^k} f(\mathbf{x})\, d\mathbf{x}.$$

In classical Monte Carlo estimation, this expectation is approximated by averaging the results of n independent samples uniformly drawn from the input space. However, since Monte Carlo samples are generated entirely at random, the resulting distribution of points can be uneven, leading to regions of oversampling and undersampling, which contribute to high variance in the estimate.

LHS mitigates this problem through stratified sampling: each input variable is divided into n equally sized intervals, and precisely one sample is drawn from each interval along every dimension. This ensures that the entire input space is more uniformly covered, thereby avoiding clustering and improving the representativeness of the samples. As a consequence, under mild smoothness or monotonicity conditions on f, LHS provides estimates with a lower variance than those obtained via standard Monte Carlo sampling. This variance reduction property is one of the main reasons for the widespread adoption of LHS in the design and analysis of computer experiments.

Gaussian Processes as Model Predictors

A **Gaussian Process (GP)** is a non-parametric, probabilistic model that defines a distribution over functions [33]. Instead of assuming a fixed functional form (as in linear regression or neural networks), a GP models the function values at any set of input points

as jointly Gaussian distributed:

$$f(\mathbf{x}) \sim \mathcal{GP}(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}'))$$

where $m(\mathbf{x})$ is the mean function (often assumed to be zero) and $k(\mathbf{x}, \mathbf{x}')$ is the **kernel function** or **covariance function**, which defines how correlated the outputs are for different input configurations $\mathbf{x}$ and $\mathbf{x}'$. The kernel thus encodes prior assumptions about the smoothness, periodicity, or complexity of the underlying function.

In regression, the GP framework provides a Bayesian approach for modeling data. Given a set of observations, it computes the posterior mean (expected prediction) and variance (uncertainty estimate) for new unseen points. This uncertainty-aware prediction capability makes GPs especially valuable in optimization contexts such as Design Space Exploration (DSE), where evaluating each configuration is expensive and selecting promising candidates efficiently is crucial.

Role of the Kernel Function. The **kernel** (or covariance function) lies at the heart of a Gaussian Process model, defining the degree of similarity between input points. It effectively controls the shape and smoothness of the predicted function. For example, the **Radial Basis Function (RBF)** kernel models smooth and infinitely differentiable relationships, while the **Matern** family allows rougher, less smooth functions by tuning the parameter ν. The kernel hyperparameters—such as length scale and variance—are optimized during training to maximize the **log marginal likelihood (LML)**, balancing data fit and model complexity. This optimization makes GPs highly flexible and adaptive to the nature of the data being modeled.

In the context of DACE, defining the relationship between inputs and outputs is equivalent to identifying an appropriate predictor that can estimate this relationship. Given the relatively limited amount of initial data typically available in such scenarios, **Gaussian Processes (GPs)** are often considered one of the most suitable choices.

Despite their versatility and accuracy, Gaussian Processes can become computationally expensive when dealing with large datasets or high-dimensional spaces, as many implementations are not sparse and rely on the full covariance matrix for predictions. Consequently, GPs are most effective in scenarios involving relatively small datasets or low-dimensional environments.

2.3.4. Techniques for Exploration

While Design of Experiments (DoE) techniques primarily define and optimize the initial sampling of the design space, the process of *exploration* itself represents a distinct challenge. Once the tools and models for the exploration are in place, a crucial question arises: *in which direction should the search proceed?*

The goal of Design Space Exploration (DSE) is to evaluate as many architectures as possible within the feasible solution space to identify the Pareto front of optimal solutions. Various exploration strategies have been proposed to achieve this objective, all sharing the same underlying challenge: DSE problems are inherently **multi-objective**, meaning that several conflicting objectives must be optimized simultaneously. Consequently, multiple equally valid **Pareto-optimal** solutions exist, each representing a different trade-off among objectives.

Classical optimization methods typically require transforming a multi-objective problem into a single-objective one. This is often achieved by aggregating or weighting objectives and running multiple simulations with different configurations, in the hope of discovering distinct solutions that collectively approximate the Pareto front. However, this approach can be computationally expensive and inefficient.

Over time, more sophisticated techniques have been developed to overcome these limitations. Among them, **Evolutionary Algorithms (EAs)** have proven particularly effective, as they can identify multiple Pareto-optimal solutions within a single simulation run by leveraging population-based search mechanisms inspired by natural evolution [15]. These algorithms can be extended or adapted to explore different regions of the Pareto front more efficiently.

Monte Carlo Methods

In the field of optimization, randomization—or more precisely, pseudo-randomization—plays a central role. Depending on the problem, a pseudo-random approach may outperform a purely deterministic one. While deterministic algorithms aim to find the exact optimal result (which, in the context of DSE, would mean identifying the absolute Pareto-optimal set), randomized methods prioritize speed and scalability, accepting a non-zero probability of suboptimal outcomes.

Surprisingly, in many practical scenarios, such approximations are perfectly acceptable. DSE is a clear example: obtaining the absolute optimal solutions would require exhaustively evaluating the entire design space, which is infeasible due to time and energy con-

straints. Therefore, every exploration algorithm operates on a subset of the possible architectures, meaning that the probability of excluding the true global optimum is always greater than zero.

From this perspective, random selection becomes a reasonable and often effective strategy. Monte Carlo-based exploration embodies this principle: given an initial sampling of the design space, architectures are randomly selected and evaluated, and the resulting Pareto front is approximated based on these sampled solutions. Although simple, this method provides a valuable baseline and serves as the foundation for more advanced stochastic exploration techniques.

Although this approach may initially appear highly effective, it presents a vital drawback that must be taken into consideration. Due to the random nature of the selection process, there is usually a non-negligible probability of converging to a local minimum rather than the global optimum of the objective function. This occurs because the evaluated architectures may belong to a region of the design space that does not contain the true optimum. Nevertheless, in specific applications, this limitation may be acceptable, depending on the problem's characteristics and the required level of precision.

Simulated Annealing

To overcome the drawback of convergence to local minima, which is typical of Monte Carlo–based algorithms, a simple yet powerful metaheuristic method known as Simulated Annealing (SA) can be employed. Introduced in the 1980s, SA was developed to address complex black-box global optimization problems [6]. The method draws its inspiration from the physical process of annealing in metallurgy, where a material is first heated to a high temperature, allowing its particles to move freely, and then gradually cooled so that the system settles into a low-energy crystalline state.

In the context of combinatorial optimization, this physical analogy translates into a probabilistic search strategy, using two equivalences: a state-space point represents the possible solid states, and the function to be minimized represents the energy of the solid. For this purpose, a control parameter *temperature* is introduced, acting, of course, as the temperature of the solid.

Assuming that for each point in the design space there exists a defined neighborhood and a mechanism for generating a new solution within it, the algorithm proceeds by evaluating both the current point and a newly generated neighbor. According to an *acceptance principle*, the new solution may either be accepted or rejected based on its quality relative to the current one.

The acceptance of a new solution depends on the system's temperature and follows a probabilistic criterion. Specifically, the probability of accepting a worse solution is defined as:

$$P = \exp\left(-\frac{\Delta f}{T}\right)$$

where Δf represents the variation in the objective function between the new and the current solution, and T denotes the current temperature. For each evaluated neighbor, a random number $r \in [0, 1]$ is generated and compared against P.

Given these ingredients, the algorithm operates as follows: each new solution is evaluated and may be accepted in one of two ways.

- If the new solution improves the objective function ($\Delta f < 0$), it is always accepted.
- Otherwise, it is accepted with probability P, i.e., if the random number $r < P$.

If a new solution is accepted, a *transition* occurs: the current solution is replaced by the newly accepted one, even if it is worse. This stochastic acceptance enables the algorithm to escape local minima and continue exploring the search space.

After each iteration, the temperature T is gradually decreased according to a predefined cooling schedule. Initially, a high temperature allows for a higher probability of accepting worse solutions, thereby promoting a more exhaustive exploration of the design space. As the temperature decreases, the likelihood of accepting inferior solutions diminishes, and the search becomes more exploitative, focusing on local refinement. The process continues until the temperature reaches a minimum threshold, at which point the algorithm terminates and the best solution found so far is returned as the final result.

As previously discussed, a DSE problem is typically a **multi-objective optimization** problem. Consequently, when applying SA to this domain, a **Multi-Objective Simulated Annealing (MOSA)** approach is required.

MOSA extends the classical SA algorithm to handle multiple, often conflicting, objectives simultaneously. One of the most common strategies for adapting SA to the multi-objective context is based on the use of a **weighted sum** of all objective functions. In this case, the original single-objective function is replaced by a *combined objective function*, defined as:

$$f_{\text{combined}}(x) = \sum_i w_i \, f_i(x)$$

where $f_i(x)$ represents the i-th objective function and w_i is the weight associated with it, satisfying $\sum_i w_i = 1$. Each weight determines the relative importance of its corresponding objective in the combined optimization process.

The MOSA algorithm is typically executed multiple times, each with a different configuration of weights. This allows the exploration of various trade-offs between objectives. The final solutions obtained from these independent runs correspond to other regions of the **Pareto front**. Collectively, these solutions approximate the Pareto-optimal set, representing the optimal trade-offs among the objectives under consideration.

Genetic Algorithms and NSGA-II

With the continuous growth in the complexity of multi-objective optimization problems, traditional methods such as Multi-Objective Simulated Annealing (MOSA) or Multi-Objective Monte Carlo approaches often become insufficient. These methods typically rely on executing the same single-objective algorithm multiple times with different parameter configurations, hoping to find diverse solutions that approximate the Pareto front. However, such approaches do not guarantee full Pareto-optimal coverage and can be computationally expensive.

Over the years, this limitation has motivated the development of **Multi-Objective Evolutionary Algorithms (MOEAs)**, which are specifically designed to handle multiple objectives in a single optimization run. Among the most prominent of these is the **Non-Dominated Sorting Genetic Algorithm (NSGA)**, and in particular its improved version, **NSGA-II** [5].

The strength of NSGA-II derives from the nature of **Genetic Algorithms (GAs)**, which are inspired by the process of natural selection and biological evolution. In this context, potential solutions are treated as individuals in a population that evolves across successive generations through the processes of selection, crossover, and mutation.

A typical GA operates through the following key components:

- **Population:** A population is a set of candidate solutions to the optimization problem. Over time, this population evolves, with each new version representing a new **generation**.
- **Fitness Function:** The fitness function evaluates the quality of each candidate solution. In DSE, this typically corresponds to minimizing performance-related objectives such as energy consumption, latency, or resource utilization.
- **Selection:** Based on fitness values, a subset of individuals is selected as *parents* to produce the next generation. Common selection strategies include **Tournament Selection** and **Roulette Wheel Selection**.
- **Crossover:** The selected parents are combined to create new offspring by exchang-

ing parts of their solution representation. This mimics genetic recombination and introduces new solution variations.

- **Mutation:** To preserve diversity and avoid premature convergence, small random changes are applied to the offspring. This step enables the exploration of new areas of the design space.

Through these iterative operations, the population progressively evolves toward higher-quality solutions, ideally converging to a set of optimal or near-optimal points. In multi-objective contexts, algorithms such as NSGA-II enhance this process by incorporating mechanisms for non-dominated sorting and maintaining solution diversity, enabling them to approximate the Pareto front within a single run effectively.

2.3.5. NSGA-II: an elitist multiobjective EA

NSGA-II introduced a robust framework for managing non-dominated solutions in the Pareto front. However, the algorithm's initial strengths were effectively overshadowed by performance-degrading architectural issues. The primary limitations centered on two aspects: an unacceptable cubic time ($O(MN^3)$) complexity and a non-elitist selection mechanism that failed to guarantee the retention of high-quality solutions. The subsequent introduction of NSGA-II was a direct response to these specific constraints, providing an optimized and computationally efficient foundation for multi-objective search.

Consistent with the Standard Genetic Algorithm (SGA), the core of the NSGA-II framework is based on an iterative process that progresses through generations. However, its distinction lies fundamentally in the generation of the successor population preserved. Unlike its predecessor, NSGA-II incorporates a highly effective mechanism to ensure the preservation of high-quality solutions.

Therefore, the architectural advancements of NSGA-II are twofold. First, it significantly improves computational efficiency by decreasing the complexity of the non-dominated sort (typically to $O(N^2)$). Second, the implementation of elitism and the crowding distance assignment procedure collectively enhance the overall quality and diversity of the final set of non-dominated solutions.

The iterative process of the **Non-dominated Sorting Genetic Algorithm II (NSGA-II)** is predicated on the concept of generations, consistent with the **Standard Genetic Algorithm (SGA)**. However, its distinction lies fundamentally in its highly effective generational transition mechanism, ensuring the preservation of quality and diversity. The transition from the current generation P_t to the successor generation P_{t+1} involves a

structured, two-part operation:

1. **Offspring Generation:** A temporary **offspring population** (Q_t) of size N is generated from the parent population P_t using standard genetic operators, specifically **selection**, **crossover**, and **mutation**. These operators introduce new solutions and ensure the necessary exploration of the search space.

2. **Elitist Selection:** To constitute the new generation, the parent and offspring populations are first merged to form a combined pool, $R_t = P_t \cup Q_t$, of size $2N$. This combined population R_t is then subjected to the core ranking mechanism of NSGA-II:

   - The individuals in R_t are first sorted based on their **non-domination rank** (front assignment).
   - Within each front, solutions are prioritized using the **crowding distance metric**, which ensures the even spread of solutions across the Pareto front.

   The best N individuals are selected sequentially from the non-dominated fronts, using the crowding distance as a tie-breaker, to form the highly-qualified successor population P_{t+1}.

This **elitist strategy** guarantees that the best solutions found across all previous generations are preserved, thereby enhancing convergence while actively maintaining necessary solution diversity.

The efficacy of any multi-objective genetic algorithm is fundamentally dependent on the speed and accuracy of the **Non-Dominated Sorting** process, which classifies the population into distinct Pareto fronts $(F_1, F_2, \dots)$. NSGA-II employs an optimized sorting mechanism to overcome the computational bottleneck inherent in earlier approaches.

Complexity of the Naive Approach A naive, brute-force sorting approach involves comparing every solution in the population P against every other solution in the population. For a population of size N with M objectives, comparing two solutions requires $O(M)$ time. Therefore, comparing a single solution to the entire remaining population requires $O(MN)$ operations.

To identify the first non-dominated front (F_1), this comparison is repeated for all N solutions, resulting in a complexity of $O(MN^2)$. Once F_1 is found, the solutions are temporarily removed, and the process is repeated to find F_2, and so on. In the worst-case scenario, where each front contains only one solution (resulting in N fronts), the total computational complexity scales cubically: $O(N \cdot MN^2) = \mathbf{O(MN^3)}$. Given large popu-

lation sizes (N), which are standard in complex optimization problems, this complexity is computationally prohibitive.

The NSGA-II Fast Sort The NSGA-II approach drastically reduces this complexity by avoiding pairwise comparisons across the entire population at every step. Instead, for each solution p in the population P, two crucial entities are computed through a single pass over P:

1. The domination count (n_p): The number of solutions that dominate p.
2. The set of solutions dominated by p (S_p): A list of all solutions that p dominates.

The computation of these two sets requires $\mathbf{O(MN^2)}$ comparisons across the entire population.

The sorting then proceeds iteratively, leveraging these pre-computed values:

- All solutions belonging to the first non-dominated front (F_1) are identified immediately, as they are the only solutions with a domination count $n_p = 0$.
- For every solution $p \in F_1$, its list of dominated solutions (S_p) is visited. For each member $q \in S_p$, its domination count (n_q) is reduced by one ($n_q \leftarrow n_q - 1$).
- If the reduction causes n_q to become zero, then q is assigned to the next non-dominated front (F_2).

This procedure continues with F_2 to find F_3, and so on, until all fronts are identified. Crucially, each solution is only visited once its domination count reaches zero. The iterative assignment of solutions to their respective fronts requires a total complexity of $\mathbf{O(N^2)}$.

Therefore, the overall computational complexity of the NSGA-II fast non-dominated sort procedure is dominated by the initial calculation of n_p and S_p, resulting in a complexity of $\mathbf{O(MN^2)}$. This significant reduction from $O(MN^3)$ to $O(MN^2)$ makes NSGA-II a viable algorithm for large-scale multi-objective optimization.

Along with the performance limitations of the original NSGA, another critical issue concerns the preservation of *diversity* among solutions.

In the context of design space exploration, maintaining diversity within the set of candidate solutions is essential to ensure a broad and practical exploration of the search space. Without sufficient diversity, the algorithm may prematurely converge to a limited region, potentially missing other promising areas of the Pareto front.

The original NSGA maintained population diversity using a **sharing function**. This

technique required the user to specify and tune a distance-related sharing parameter, σ_{share}, which governed the extent of fitness sharing desired between solutions to encourage uniform distribution across the Pareto front. This approach, however, introduced two primary deficiencies: a significant increase in **computational complexity** and a reliance on a **user-defined parameter** (σ_{share}), which undermined the algorithm's robustness and ease of application. The maintenance of diversity was coupled with an arbitrary, external parameter.

The Crowding Distance Metric NSGA-II resolves this issue by replacing the sharing function with the **crowding distance metric**, a parameter-less method for estimating solution density in the objective space. For any single solution i belonging to a non-dominated front F_k, its crowding distance, d_i, serves as an estimate of the largest hyper-cuboid perimeter that does not contain any other solution in the front. This metric quantifies the average distance of i's two nearest neighbors along each of the M objectives. A larger crowding distance value indicates that the solution i occupies a less crowded region, thereby promoting the necessary diversity.

The calculation of the crowding distance for all solutions within a front F_k (of size $|F_k|$) involves an efficient procedure:

1. Initialize the distance $d_i = 0$ for all solutions in the front.
2. For each objective function $m = 1, \dots, M$:
    - Sort the solutions in F_k based on the ascending values of the m-th objective function.
    - The two boundary solutions (minimum and maximum values) are assigned an infinite distance ($d_{boundary} = \infty$) to ensure their preservation in the population.
    - For all intermediate solutions, the crowding distance d_i is updated by adding the normalized absolute difference between the m-th objective function values of its two adjacent solutions.

Since the process involves M independent sortings on a set of at most N solutions, the total computational complexity for the crowding distance assignment is $\mathbf{O}(\mathbf{MN}\log\mathbf{N})$.

The Crowded-Comparison Operator ($\prec$) The density metric is integrated directly into the selection mechanism via the **crowded-comparison operator** ($\prec$). This operator guides the elitist selection process, allowing the algorithm to simultaneously select for **convergence** (low rank) and **diversity** (high distance). For any two solutions, i and j,

solution i is preferred over solution j ($i \prec j$) if:

$$i \prec j \quad \text{if} \quad (\text{rank}(i) < \text{rank}(j)) \quad \text{or} \quad (\text{rank}(i) = \text{rank}(j) \quad \text{and} \quad d_i > d_j)$$

This definition establishes a formal partial order: solutions with a superior non-domination rank are always selected. When solutions share the same rank (i.e., belong to the same Pareto front), preference is given to the solution located in the less dense region (i.e., the one with the higher crowding distance).

With the architectural components of a fast non-dominated sorting mechanism and a parameter-less diversity preservation strategy in place, we can now construct the main operational loop of the NSGA-II algorithm. The fundamental objective of this loop is to guide it toward the actual Pareto-optimal front with each successive generation. The procedure is designed to be both computationally efficient and robust, ensuring that the quality and diversity of the solutions improve over time.

The transition from a parent population at generation t, denoted as P_t, to the successor population P_{t+1} is not a simple replacement. Instead, it is a structured, two-phase process designed to uphold the principles of elitism and diversity rigorously.

Generation of the Combined Population The first phase begins by creating a temporary offspring population, Q_t, of size N from the parent population P_t. This is accomplished using standard genetic operators: tournament selection guided by the crowded-comparison operator, followed by recombination (crossover) and mutation.

Once the offspring population is generated, the core of NSGA-II's elitist strategy comes into play. The parent and offspring populations are merged to form a combined population, $R_t = P_t \cup Q_t$, which has a total size of $2N$. This critical step embodies the algorithm's elitist principle, ensuring that high-performing individuals from the parent generation are given a direct opportunity to carry over to the next generation, thereby preventing the loss of valuable solutions discovered in previous generations. This combined pool now contains the full breadth of genetic information from both the established parent solutions and the newly explored offspring.

Selection for the Successor Generation The second phase involves a merit-based competition to select the N most fit individuals from the combined population R_t to form the next generation, P_{t+1}. This selection is a carefully orchestrated procedure guided by the twin goals of convergence and diversity.

First, the entire combined population R_t is subjected to the fast non-dominated sorting procedure described previously. This process stratifies the $2N$ solutions into a hierarchy of non-dominated fronts $(F_1, F_2, F_3, \dots)$, where F_1 contains the absolute best solutions in the combined pool.

The new population, P_{t+1}, is then built by adding solutions from these fronts, starting with the very best. All individuals from the front, F_1, are added first. If space permits, all individuals from front F_2 are added, and this process continues sequentially through the fronts. This ensures that elite solutions are always preserved.

The process stops when including an entire front would cause the size of P_{t+1} to exceed the population size N. Let us say this last front is F_l. To fill the remaining slots in the new population, a selection must be made from the individuals within this critical front F_l. It is at this stage that the crowding distance becomes the decisive tie-breaker. The solutions within front F_l are sorted in descending order based on their crowding distance values. The individuals from the less-crowded regions of the objective space (i.e., those with the highest crowding distances) are selected one by one until the new population P_{t+1} reaches its full size of N. This procedure ensures that, among solutions of equal non-domination rank, those that contribute most to the diversity of the front are prioritized.

Computational Complexity and Efficiency The overall computational complexity of a single generation of the NSGA-II main loop is governed by its most intensive operations. The fast non-dominated sorting of the $2N$ individuals in the combined population has a complexity of $\mathbf{O(M(2N)^2)}$, which simplifies to $\mathbf{O(MN^2)}$. The crowding distance assignment, which requires sorting within each front, has a worst-case complexity of $\mathbf{O(MN \log N)}$.

Since the non-dominated sorting term is quadratically dependent on the population size. At the same time, the crowding distance is log-linear; however, the overall complexity of the algorithm is dominated by the sorting step, resulting in a total complexity of $\mathbf{O(MN^2)}$ per generation. This quadratic complexity represents a monumental leap in efficiency compared to the cubic complexity of its predecessor, making NSGA-II a practical and scalable algorithm for complex optimization problems with large populations. Furthermore, a practical implementation can optimize this process by halting the non-dominated sort as soon as enough fronts have been identified to fill the N slots of the next generation, often leading to even better average-case performance.

Crucially, the diversity preservation mechanism in NSGA-II is entirely self-contained. By using the crowding-comparison operator in both the tournament selection phase and

the final population reduction step, the algorithm naturally favors solutions in less dense regions of the search space. This eliminates the need for any external, user-defined niching parameters (such as the σ_{share} parameter required in the original NSGA), thereby making the algorithm more robust, easier to implement, and less sensitive to parameter tuning.

3 | Related works

The relentless advancement of Machine Learning (ML) and Deep Learning (DL) models has exposed the inherent limitations of traditional von Neumann computing architectures. The ever-increasing model complexity and data volume have intensified the energy and latency penalties associated with data movement between separate processing and memory units—a critical challenge known as the von Neumann bottleneck. In response, a rich landscape of specialized hardware accelerators has emerged, with In-Memory Computing (IMC) standing out as a particularly promising paradigm for achieving significant gains in energy efficiency and computational throughput.

This chapter provides a structured review of the state-of-the-art research that underpins the automated design and optimization of these advanced accelerator systems. The body of work is organized into a logical hierarchy that reflects the distinct yet interconnected layers of the design process, moving from foundational hardware modeling to holistic system-level exploration. The survey is divided into three primary sections:

1. **Modeling Tools:** At the foundation of any design exploration are the tools used to abstract and evaluate hardware performance. This section begins by examining specific IMC accelerator modules that serve as the architectural basis for this thesis. We will analyze their core operational principles and the analytical modeling techniques developed to estimate their performance, power, and area without necessitating physical fabrication.

2. **Mapping Tools:** An accelerator's performance is not determined by its hardware alone, but by the efficiency with which a computational workload is scheduled onto it. This section delves into the crucial task of mapping. We will explore state-of-the-art tools designed to navigate the vast search space of possible dataflows and tiling strategies—the map-space—to identify an optimal mapping for a given workload and architecture.

3. **Exploration Tools:** Operating at the highest level of abstraction, these tools address the ultimate challenge of hardware-software co-design. They automate the simultaneous search for optimal hardware parameters and their corresponding op-

timal mappings. The methodologies employed by these exploration tools serve as a direct inspiration for the core contributions of this work, and a thorough analysis of their strengths and limitations will establish the context for the novel approaches proposed in this thesis.

By systematically examining these three pillars of accelerator design automation, this chapter will construct a comprehensive overview of the current research landscape, thereby highlighting the gaps and opportunities that motivate the thesis presented.

3.1. Modeling Tools

3.1.1. Digital In-Memory Computing (DIMC) Modules

As previously discussed, **In-Memory Computing (IMC)** is an architectural solution designed to overcome the limitations of the traditional von Neumann paradigm. By performing the Multiply-and-Accumulate (MAC) operations dominant in Deep Learning (DL) models directly within memory, IMC architectures drastically reduce the high latency and energy costs of data movement between processing and memory units. This is especially effective for the convolutional layers in neural networks, which feature significant data reuse.

What follows is an analysis of a representative digital IMC module from the literature.

The Colonnade Macro

Colonnade is a reconfigurable, SRAM-based **digital bit-serial In-Memory Computing** macro designed as a fully digital accelerator for neural network tasks [16]. It effectively combines the robustness of digital computation with the efficiency of the IMC approach.

The fundamental component of the architecture is its custom-designed **digital bitcell**, which enables the in-memory computation. Each bitcell consists of four main blocks:

- **6T SRAM Cell:** A standard memory cell that stores a single bit of a network weight.
- **XNOR Gate:** Performs the bitwise multiplication between the stored weight bit and a serialized input bit.
- **Full-Adder (FA):** Acts as the accumulation block, summing the result from the XNOR gate with the partial sum and carry from the cell above it in the column.
- **Multiplexers (MUXs):** Two 2:1 MUXs are used to configure the cell's operational

mode based on its position within a column.

Multiple bitcells are stacked vertically to create a **"Column MAC,"** which functions as a single bit-serial multiplier-accumulator. The key to Colonnade's flexibility lies in its ability to reconfigure these cells in two distinct functional modes:

1. **Type-A (Active Mode):** In this mode, the bitcell uses all its components to perform both multiplication and accumulation. The number of Type-A cells in a column directly defines the **precision of the weight** (e.g., 4 Type-A cells are used for a 4-bit weight).

2. **Type-B (Accumulate-Only Mode):** Here, only the Full-Adder is enabled. These cells are added to a column to **extend the precision of the final output**, propagating sums and carries to prevent numerical overflow during accumulation.

To better comprehend how these components integrate to form and operate the proposed macro, a practical operational example is illustrative. Let us assume an *N-bit weight* from a conventional neural network layer. This N-bit weight is stored across the SRAM cells of N **Type-A bitcells**, which are configured for both multiplication and accumulation. The input operand is then processed in a *bit-serial* fashion, commencing with its **Least Significant Bit (LSB)**.

The computation unfolds over a series of clock cycles. In the initial cycle, the input's LSB is broadcast to the entire column and is simultaneously multiplied by every bit of the stored weight. The resulting products are then summed by the vertical chain of Full-Adders, which functions as a *ripple-carry adder*, thereby producing a first **partial sum** at the column's output. This same operation proceeds for all subsequent bits of the input, ultimately generating several partial sums equal to the input's bit precision.

The macro itself is structured as a two-dimensional (2D) array of these bitcells, wherein multiple columns operate in parallel, each processing a different weight. The partial sum output from one column is passed as an input to the adjacent column, implementing a **systolic architecture** in which data flows horizontally across the array.

The sequence of partial sums generated at each clock cycle from the row of MAC columns must be combined to yield the final result. This function is performed by the **Post-Accumulator**.

The primary strength of this macro lies in its **high energy efficiency**, achieving performance levels comparable to those of analog CIMs. This is complemented by the inherent *robustness* of its fully digital approach, which mitigates issues common in analog circuits, such as noise susceptibility. A key feature is its **flexibility and reconfigurability**; in-

deed, the architecture can be adapted for varying operand precisions simply by changing the bitcells' operational mode. Furthermore, the design presents significant advantages with respect to both physical *area* and *scalability*.

Desoli et al.: 8T SRAM-Based Energy-Efficient Compute-in-Memory Processors for ML

Desoli *et al.* [7] present a scalable Neural Processing Unit (NPU) built from fully digital SRAM-based In-Memory Compute (DIMC) tiles, designed for high energy efficiency, deterministic behavior, and compatibility with advanced CMOS scaling.
Each 32 KB DIMC tile uses a pushed-rule 8T 1R1W SRAM and can operate either as a single memory array in memory-mapped mode or as independent subarrays in compute mode. In compute mode, multiple read wordlines (RWLs) are activated simultaneously, driving interleaved digital MAC slices. A tile (Figure 3.1) supports 256,4b, 512,2b, or 1024,1b MAC operations per cycle via a shared accumulation pipeline. The design also features partial-sum input/output paths, multi-RWL decoding, a 256-bit/cycle write interface, and low-voltage operation down to 0.525V using programmable assist circuits and forward body-biasing.
Up to eight tiles form a cluster, allowing activation broadcasting, tile chaining for larger kernels, and striping strategies to match the topology of each neural network layer. Weight tensors can be preloaded or streamed on demand. AXI slave and high-bandwidth activation interfaces handle memory access and computation triggers via a 1024-bit feature buffer. Partial sums are multicast-packed, and output multiplexers enable interleaving, packing, bit shifting, rounding, and sign extension. Each tile also integrates a ReLU activation unit, with final outputs sent to tensor DMA engines for storage.

Fabricated in 18 nm FD-SOI, the complete NPU integrates 64 DIMC tiles (2 MB) in 4.2 mm^2, reaching 57 TOPS at 4b and up to 310 TOPS/W in 1b mode, with a compute density of 13.6 TOPS/mm^2. A dedicated compiler manages tensor striping, tile chaining, and parallel execution, allowing even large networks, such as VGG, to fit entirely within DIMC storage, thereby minimizing activation reloads and achieving near-peak throughput.

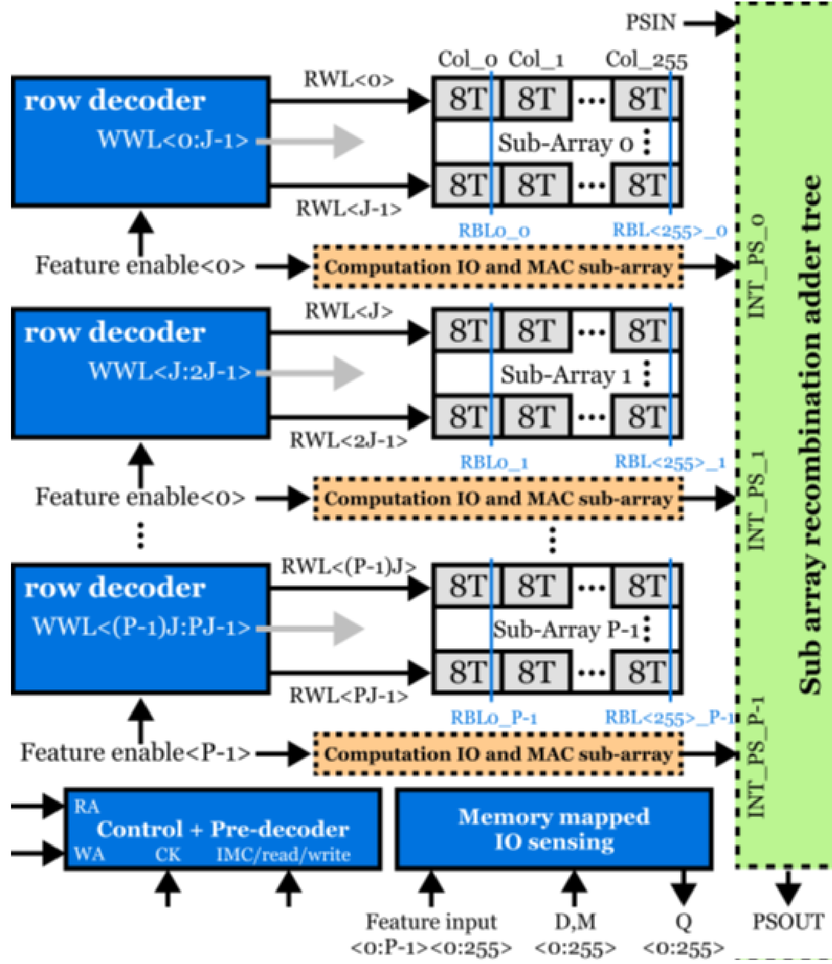


Figure 3.1: Desoli Digital IMC architecture.

3.1.2. CimLoop

CiMLoop is an open-source modeling framework engineered for the analysis, evaluation, and design space exploration of **IMC** systems intended for the acceleration of **Deep Neural Networks (DNNs)** [2].

The framework is designed to address three primary challenges in IMC modeling: *flexibility*, *accuracy*, and *speed*.

Flexible architectures IMC macros and implementations, as discussed in the previous sections, differ significantly in their circuit designs, architectural hierarchies, and dataflow behaviors. These differences directly impact data movement patterns, making it essential for a modeling framework to provide a *flexible* and *composable* way to describe architectures. Such flexibility enables users to represent a broad spectrum of components, their interconnections, and their associated data movement strategies.

The **CIMLoop** framework was designed precisely with this goal in mind, offering a flexible and hierarchical modeling abstraction that distinguishes it from other existing tools. In CIMLoop, architectures are expressed through a nested **container hierarchy**, where each *container* encapsulates a logical group of components. A *component* represents any entity that stores, moves, or processes data.

For each component, the user can define key behavioral attributes that govern its role in the dataflow:

- **Data Reuse:** Defines whether data can be temporally retained across cycles. Components with *temporal reuse* (e.g., buffers) can store data between iterations, while those with *no temporal reuse* must reload data each cycle. Components without temporal reuse may optionally support *coalescing*, i.e., merging multiple accesses to the same data into a single memory transaction.
- **Bypass and Spatial Reuse:** Describes how data can traverse the architecture. Data may bypass specific components without activating them (e.g., input data bypassing a weight buffer). Spatially, data can be either *reused* (through multicast or reduction across components) or *not reused* (unicast to each component individually).

A *container* in CIMLoop may consist of either individual components or nested subcontainers, forming a hierarchical organization known as a *container hierarchy*. This hierarchical abstraction is particularly effective for modeling IMC systems, as it enables the modular representation of complex architectures composed of heterogeneous components.

Such a structure facilitates the isolation of architectural layers, enhances compatibility with various modeling tools, and supports flexible abstractions for representing memory hierarchies. Moreover, it enables users to naturally express nesting strategies and mixed architectural compositions, capturing the interaction between computational units, buffers, and communication fabrics within a unified framework.

Accuracy Challenge To accurately guide architectural design choices, the modeling tool must provide precise energy estimations. However, the energy consumption of device- and circuit-level components in IMC architectures is inherently *data-value-dependent*—that is, it varies according to the actual data values processed by each component.

This data dependence IMC modeling can significantly influence the total system energy, meaning that neglecting it would lead to inaccurate or misleading evaluations. Therefore, achieving **high energy accuracy** in IMCmodeling requires explicitly capturing and

accounting for data-value-dependent behavior across all components.

CIMLoop addresses this challenge by explicitly modeling how data values propagate through each component and how these affect the overall energy consumption. To capture this data-dependent behavior, the framework follows a three-step procedure:

(a) Workload Operand Distributions: Instead of modeling full tensors, CIMLoop represents each workload operand (inputs, outputs, and weights) using value distributions. This abstraction enables efficient modeling while preserving the statistical behavior of data across layers.

(b) Encoding and Slicing: Each tensor is then represented according to the encoding and slicing scheme of the underlying IMChardware. Supported encoding types include offset, differential, XNOR, and magnitude-only schemes, among others. After encoding, bit-level slicing enables CIMLoop to spatially and temporally map data elements across components, ensuring compatibility with Timeloop-based mapping flows.

(c) Component Energy Modeling: Finally, with encoded and sliced data distributions, CIMLoop evaluates energy at the component level. Each component type (e.g., DACs, ADCs, buffers, memory cells) models energy consumption according to its physical behavior—such as voltage duration or switching activity—yielding an accurate and data-aware estimation of the total energy. Additionally, **CIMLoop** provides a flexible plug-in interface that allows users to integrate custom, data-value-dependent energy models for components within their own architectures. The framework includes several built-in plug-ins—such as *Accelergy* and *CACTI*, as well as dedicated models for analog elements like ADCs or neurosynaptic arrays—that enable the accurate characterization of different circuit behaviors, including array row/column drivers and conversion modules.

Fast Modeling The design space to be explored in IMC systems is typically vast, which makes the speed of the modeling tool a crucial requirement. At the same time, high accuracy is essential, as the energy consumption of each component depends on the specific data values it propagates. Capturing this data-value dependence in detail, however, can make modeling computationally expensive. To overcome this limitation, CIMLoop precomputes the average energy consumed per action for each component type. These averages can then be reused across any number of similar operations, allowing CIMLoop to evaluate even large and complex architectures in constant time. This strategy is particularly advantageous for IMC systems, where many components operate in parallel.

To model data-dependent behavior efficiently, CIMLoop employs statistical representations of DNN operands. Instead of using full tensors, the framework relies on the *dis-*

tributions of operand values—namely, probability mass functions that describe inputs, weights, and outputs. This approach enables CIMLoop to estimate energy accurately without requiring a complete DNN execution. The decoupling between workload profiling and system modeling significantly accelerates design space exploration, enabling multi-fidelity modeling and providing users with the flexibility to trade off accuracy and runtime according to their specific needs.

For computational efficiency, CIMLoop assumes independence among tensor distributions and stores them separately. This simplification reduces complexity from $O(N^T)$ to $O(NT)$ while maintaining high accuracy in most practical cases. Moreover, CIMLoop assumes that the *per-action energy* of each component is invariant to the specific mapping used. This assumption holds for regular dense mappings, where tensor elements are treated uniformly. As a result, CIMLoop can precompute and reuse average per-action energy across multiple mappings, achieving a balance between modeling precision and exploration speed.

3.2. Mapping Tools

3.2.1. Timeloop

Timeloop is a foundational infrastructure for the systematic evaluation and exploration of the DNN accelerator design space [22]. Its central premise is that a fair comparison between different hardware architectures is only possible if each architecture is evaluated using its own optimal mapping for a given workload.

To achieve this, Timeloeloop decouples the description of the **architecture** from the **mapspace**. The architecture is specified through a concise, hierarchical representation of its compute units, memory levels, and interconnects. The mapspace, which encompasses all valid ways of scheduling a workload on that architecture, is defined by tiling factors, loop ordering, and parallelism strategies.

The core of Timeloop is its **mapper**, an automated search engine that explores this vast mapspace to find the mapping that optimizes a user-defined objective (e.g., energy, latency, or EDP). It uses a fast analytical model to project the performance of each mapping without resorting to slow, cycle-accurate simulation. This systematic search ensures that an architecture's potential is fully realized before it is compared against others. By automating the mapping process, Timeloop elevates accelerator design from an ad-hoc art to a more rigorous and scientific process, providing a robust framework for both evaluating existing designs and exploring novel ones.

3.2.2. FactorFlow

FactorFlow represents a comprehensive framework, encompassing both an analytical performance model and an automated mapper, designed to identify near-optimal mappings for computational kernels onto Spatial Architectures (SAs) [28]. Initially conceived for General Matrix Multiplications (GEMMs), its scope has broadened to include convolutional workloads as well.

GEMMs, operations involving the product of two matrices potentially summed with a *bias* term, form a cornerstone of many scientific and deep learning applications. Computationally, a GEMM can be represented as three nested loops executing a Multiply-Accumulate (MAC) operation. Spatial Architectures have emerged as an effective hardware paradigm for accelerating GEMMs. These architectures typically consist of an array of Processing Elements (PEs) capable of performing MAC operations, interconnected to facilitate data sharing and reuse across a dedicated memory hierarchy.

The primary performance metrics considered within this context are energy consumption, execution latency, and their product, the Energy-Delay Product (EDP). The rationale for employing SAs for GEMM computation lies in their ability to exploit data reuse inherent in the algorithm. By minimizing accesses to slower, more energy-intensive levels of the memory hierarchy, SAs can significantly reduce the EDP, which is often dominated by memory access costs.

FactorFlow specifically addresses the challenge of efficiently finding optimal mappings for diverse GEMM-SA combinations within practical time constraints. The methodology commences with the formal definition of the target SA, abstracting it as a hierarchical composition of three fundamental component types:

- **Memory Level:** Represents storage elements (e.g., register files, buffers, DRAM) characterized by capacity, access energy, and bandwidth.
- **Spatial Fanout Level:** Models the replication of hardware resources (e.g., PEs, subsequent memory levels) in a multi-instance configuration, defining spatial parallelism capabilities.
- **Compute Level:** Represents the functional units performing MAC operations, defined by their energy cost and latency per operation.

Once the architecture is formally specified, the computationally intensive task is the mapping process. This involves determining the optimal strategy for allocating operands throughout the defined memory hierarchy and binding each MAC operation to a specific

PE within the compute level.

This process generates a vast set of potential configurations, known as the **map-space**. A mapping is considered *valid* only if it satisfies both hardware resource constraints and any user-defined limitations. The subset of configurations meeting these conditions defines the **valid map-space**. Selecting the optimal point within this space involves several key mapping decisions, which in the FactorFlow framework are:

1. **Tiling**: The GEMM operation is decomposed into progressively smaller tiles to fit within the hierarchy of the Systolic Array (SA). FactorFlow achieves this by nesting loops corresponding to each hierarchy level, where each performs a fraction of the total iterations across all dimensions.

2. **Parallelism Strategy**: Selected loop iterations are unfolded across different hierarchy levels to enable concurrent execution of distinct data portions, improving throughput and resource utilization.

3. **Loop Ordering**: The order of loops in a GEMM can be redefined without affecting correctness but with a substantial impact on performance. In this context, the **dataflow** refers to the specific loop order, determining how data moves and is reused within the architecture.

The resulting **map-space** is thus composed of all possible combinations of tiling, parallelism, and loop ordering decisions.

Mapper The FactorFlow framework includes a dedicated **mapper** responsible for navigating the map-space and selecting the configuration that minimizes the *Energy–Delay Product per Operation Unit (EDPoU)*. The mapper operates through three heuristic-based stages:

1. *Iterative Permutation*: Exploration of different loop permutations using adaptive heuristics to identify promising configurations based on performance estimates.

2. *Fanout Maximization*: Optimization of spatial data distribution to maximize data reuse and minimize communication overhead across processing elements.

3. *Greedy Descent Factor Allocation*: A final refinement step that allocates tiling and parallelization factors across hierarchy levels through a greedy descent procedure toward configurations achieving minimal EDPoU.

At the end of this process, the output corresponds to the optimal mapping for the selected workload–architecture pair. These procedures have been shown to complete within a short

time, demonstrating both the efficiency and effectiveness of the proposed tool.

3.3. Exploration Tools

3.3.1. DOSA

DOSA is a Design Space Exploration (DSE) framework based on a differentiable single-loop model, specifically targeting DNN accelerators [11].

In the field of DSE for hardware accelerators, many existing works address the mapping and design spaces separately. DOSA, instead, takes a unified approach: it aims to capture the relationships between DNN mapping factors and performance objectives within a differentiable analytical model.

The exploration of the map-space, which includes both classical matrix multiplications and convolutional workloads, follows three main steps:

- **Spatial Loop Tiling** – defines which loops are mapped to parallel spatial resources.
- **Temporal Loop Tiling** – specifies which loop iterations are grouped into blocks at each memory level.
- **Loop Ordering** – determines the order of access across dimensions.

To efficiently search this ample space, DOSA employs gradient descent optimization.

Tool Flow Given a workload represented as a set of layers, DOSA organizes its flow as a single loop managing the entire exploration. The main steps are as follows:

1. **Mapping Generation:** create mappings using CoSA, targeting *randomly selected* valid hardware architectures.
2. **Hardware Parameterization:** from these mapping–design pairs, compute the required resources and convert them into minimal hardware parameterizations.
3. **Memory Access Estimation:** using DOSA's differentiable model, estimate the number of memory accesses for each mapping.
4. **Performance Prediction:** combine arithmetic operation counts and memory accesses to derive roofline-based latency and event-based energy predictions for each layer's mappings.
5. **Optimization:** update all mappings in parallel using gradient descent.

This procedure is repeated iteratively from the second step until the gradient converges.

One key distinction of DOSA compared to other frameworks lies in its computation of hardware resources starting from a given mapping. The target accelerator for this process is the *Gemmini* architecture, which consists of a Processing Element (PE) array, scratchpad memories, and accumulator SRAMs. DOSA estimates:

- **PE capacity requirements**, assuming square PE arrays as supported by Gemmini;
- **Buffer capacity requirements**, calculated as the sum of buffer needs across all workload levels.

These computations adapt based on the optimization strategies considered, such as operand-stationary dataflows or others.

Evaluation Once mappings and architectural requirements are defined, the next step is to evaluate accelerator performance. DOSA provides three key analytical components for this:

- **Traffic Estimator:** employs differentiable non-convex functions to compute data movement at each buffer level, estimating the number of reads, writes, and updates.
- **Latency Model:** latency is estimated by dividing the total number of MAC operations by the product of all spatial factors in the mapping, expressed in cycles. Memory access latency can also be evaluated per hierarchy level.
- **Energy Model:** energy is estimated using well-established tools such as CACTI and Accelergy.

By combining these estimations, DOSA applies gradient descent to minimize the model's Energy-Delay Product (EDP).

Finally, DOSA produces optimized exploration results by employing advanced optimization techniques as part of its **search strategy**, including alternative mapping and design exploration algorithms, both iterative and gradient-based **loop ordering** strategies, and additional mechanisms such as *start point rejection, rounding, invalid mapping prevention, and pipeline fusion.*

3.3.2. Di Gamma

Di Gamma introduces a domain-aware genetic algorithm for hardware–mapping co-optimization targeting DNN accelerators [15]. The methodology presented in this thesis takes strong

inspiration from Di Gamma; therefore, understanding its workflow is essential to contextualize the proposed work.

Unlike the previously discussed frameworks, Di Gamma begins from a predefined *design budget*, expressed in terms of resource constraints, and a given DNN model. The ultimate goal of the algorithm is to jointly determine an accelerator design configuration and a corresponding optimal mapping strategy for the given workload, thus achieving a balanced hardware–software co-optimization.

Co-Optimization Design Similar to DOSA, Di Gamma performs a **co-optimization** between the design and mapping exploration processes, through what the authors define as a **Co-Opt framework**. This framework takes as input the target model, optimization objectives, design budget, and chosen optimization algorithm. It produces an optimized accelerator design point, characterized by both hardware configuration and mapping strategy. The exploration space can optionally be constrained by additional user-defined design limitations, effectively reducing the search domain.

The framework consists of two main components:

- **Optimization Block:** This module handles the problem specification, considering the user-defined constraints and employing a genetic algorithm. The number of acceptable design points is treated as a hyperparameter that controls the breadth of exploration.
- **Evaluation Block:** This module contains an encoding and decoding mechanism that defines the representation of potential solutions and enables the evaluation of their fitness. A key strength of Di Gamma lies precisely in this encoding scheme, where design points are first encoded for assessment and then decoded to recover the actual configuration parameters.

The evaluation phase itself leverages the *MAESTRO* framework, an open-source analytical tool for modeling DNN dataflow and performance.

Domain-Aware Genetic Algorithm The Di Gamma optimization process follows a classic **genetic algorithm** paradigm: encoded design points are iteratively perturbed through combinations of genetic operators to evolve toward optimal solutions. However, Di Gamma distinguishes itself from traditional genetic algorithms through its **domain-aware operators**, specifically designed for the hardware–mapping co-optimization problem.

The genetic operators responsible for mapping (such as tiling, loop ordering, parallelism,

and clustering) are adapted from the GAMMA framework [15]. In addition, a dedicated hardware-level operator, *Mutate-HW*, perturbs the configuration of the processing elements (PEs) by adjusting parameters π_{L2} and π_{L1}, which determine the total number of PEs and the aspect ratio of the PE array.

For buffer allocation, Di Gamma introduces an intelligent allocation strategy that determines optimal buffer sizes for the L_1 and L_2 levels. This process leverages the minimum buffer requirements obtained during the decoding stage (Section III-C2), ensuring that each level is allocated precisely the amount of memory required to maximize utilization efficiency.

The framework defines two main classes of genetic operators:

- **Mutate-HW:** operates on the hardware design space, perturbing the PE configuration and buffer allocations.
- **Mutate-Map:** acts on the mapping space, modifying mapping parameters while maintaining consistency with the associated hardware configuration.

Together, these specialized operators enable Di Gamma to effectively traverse both the hardware and mapping design spaces, allowing for an efficient co-evolution of architecture and dataflow.

4 | Proposed In-Memory Computing-based Methodology

4.1. Overview

This chapter introduces the proposed methodology, which extends the traditional design space exploration paradigm by jointly considering architectural and dataflow aspects inherent to In-Memory Computing (IMC) accelerators. The approach focuses on systematically analyzing and optimizing IMC-based hardware architectures to identify the most suitable configurations under given workload and design constraints.

The methodology builds upon and extends the *FactorFlow* framework, a state-of-the-art tool for mapping computational workloads onto spatial architectures. The proposed extensions enable *FactorFlow* to model IMC-based accelerators, overcoming its original limitations and enabling an exploration process that seamlessly integrates computation–memory co-design.

This chapter is organized as follows. Section 4.2 discusses the limitations of the original *FactorFlow* framework and the motivations behind its extension. Section 4.3.1 formalizes the problem definition, outlines the fundamental stages of the exploration process, and introduces the parameter space definition and sampling strategies. Finally, the chapter concludes by describing the multi-objective evolutionary algorithm used to identify Pareto-optimal design solutions.

4.2. Template Architectures

The **template architecture** in this thesis is based on **spatial architectures (SAs)**, which, in their basic configuration, consist of multidimensional arrays of Processing Elements (PEs) that execute Multiply-and-Accumulate (MAC) operations: each PE is con-

nected to its neighbors and to a memory hierarchy through specialized interconnects, enabling efficient data movement both across PEs and between memory levels.

As discussed in the previous chapter, **FactorFlow** is one tool that facilitates this management. In its standard configuration, *FactorFlow* models an accelerator using three hierarchical levels: **Memory**, **Fanout**, and **Compute**.
As shown in Figure 4.1, a typical SA modeling connects two memory levels, representing DRAM and an SRAM buffer, while the two fanout levels are used to distribute the PEs into an $n \times m$ array. Within FactorFlow, a PE is modeled as a compute level, paired with a register that is itself treated as a memory level. Each PE performs one MAC operation per cycle, leveraging the hierarchical memory structure to minimize data movement and maximize throughput.

The first step in implementing the proposed methodology was to integrate **In-Memory Computing (IMC)** capabilities into the *FactorFlow* framework, resulting in an IMC-enabled spatial architecture, which serves as the **template architecture** of this work.
While *FactorFlow* was initially designed for architectures based on classical spatial arrays of processing elements (PEs) and hierarchical memory hierarchies, this thesis introduces additional modeling components that enable the accurate representation of IMC-enabled accelerators. These modifications extend both the structural modeling and the energy–latency estimation mechanisms of the framework.

To incorporate IMC architectures, two new levels were introduced, as shown in Figure4.2: `CIM Memory` and `CIM Compute`. The first models the *memory subcomponent* of the IMC cell—responsible for operand storage—while the second models its *computational subcomponent*, which performs digital multiply–accumulate (MAC) operations directly within the memory array. Although a physical IMC cell intrinsically combines storage and computation within the same device, in this work, the two functionalities are represented as distinct entities within the *FactorFlow* hierarchy. This abstraction is not meant to alter the physical semantics of the IMC operation, but rather to preserve modeling fidelity while complying with internal framework constraints—specifically, the requirement that the lowest hierarchy level must correspond to a compute stage. The separation ensures that both the data-storage behavior (modeled by `CIM Memory Level`) and the in-situ compute behavior (modeled by `CIM Compute Level`) can be independently parameterized and analyzed. Consequently, each `CIM Memory Level` is directly paired with a corresponding `CIM Compute Level`, effectively reproducing the natural memory–compute coupling characteristic of IMC architectures while maintaining consistency with *FactorFlow*'s hierarchical abstraction model.

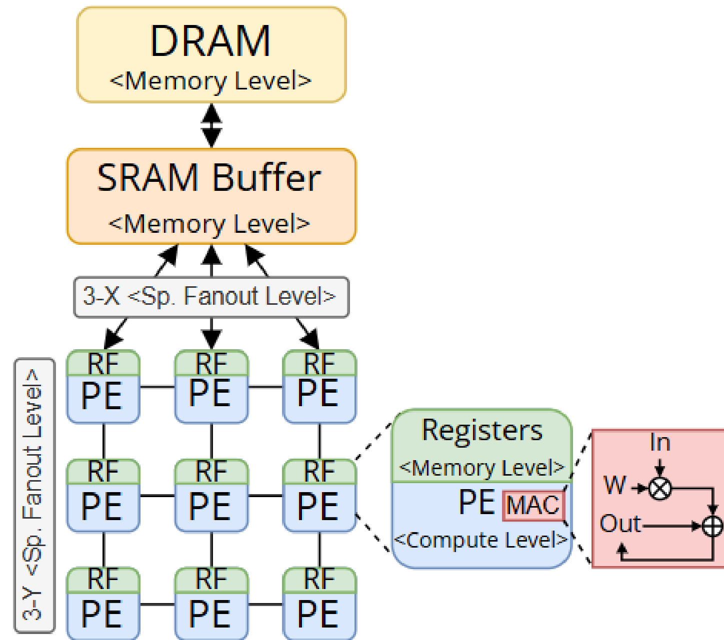


Figure 4.1: Template Spatial Architecture without IMC integration.

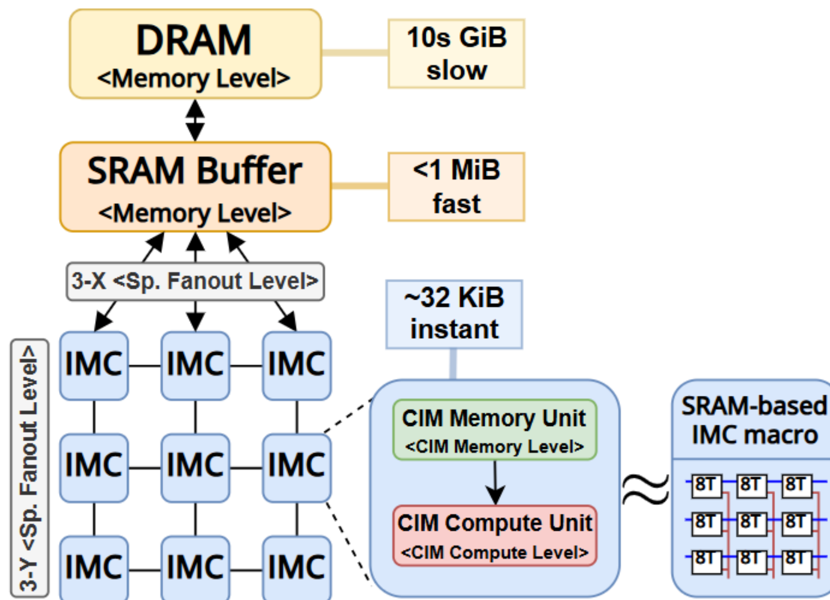


Figure 4.2: Template architecture including IMC integration through the newly introduced `CIM Memory` and `CIM Compute` levels.

4.2.1. IMC Modeling and Characterization.

Once the hierarchical integration was established, the next step involved extending the energy and latency estimation mechanisms to account for IMC-specific operations. The

IMC cell arrays used in this thesis are based on the structures simulated and characterized by the University of Ferrara. The collaboration provided detailed measurements of both **energy consumption** and **latency** across different IMC array dimensions, supply voltages (V_{DD}), and sparsity conditions.

The IMC cell simulated for this purpose is shown in Figure 4.3, representing the fully digital SRAM 8T-based bitcell used as the fundamental computational element. These bitcells are arranged into a two-dimensional array of size $J \times K$, forming the basic structure of the DIMC macro. All circuits were designed and simulated with a 22 nm HP Predictive Technology Model (PTM).
Figure 4.4 presents the complete DIMC array modeled and simulated in this study. The architecture includes a row decoder, which generates the signals `RWL_0` through `RWL_J`, together with the associated control and pre-decoding logic responsible for asserting the read-access (`RA`) and write-access (`WA`) signals, as well as the global clock signal `CK`. The tile also includes a memory-mapped I/O interface, which is not directly simulated but is assumed to be present in the architecture, following the concept introduced in [7]. Its omission in the simulation does not affect the final results.
This interface receives the feature-map inputs `FeatureMap`$[0 : P-1][0 : K]$, along with the control signals `D` and `M` for each column k, and produces the corresponding output vector `Q`$[0 : K]$, supporting standard read and write operations in addition to IMC computation. Finally, the row decoder drives the wordline signals `WWL`$[0 : J-1]$, ensuring correct row selection during both memory-access and in-memory compute operations.

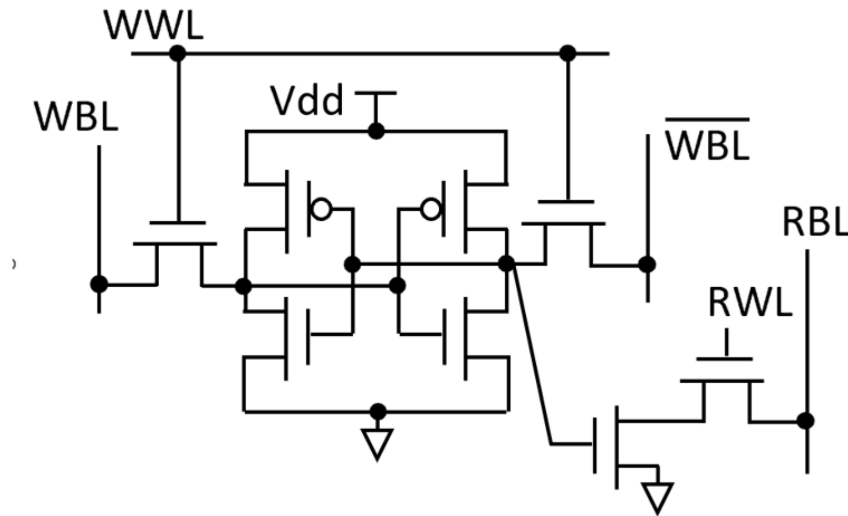


Figure 4.3: Simulated 8T SRAM cell for digital In-Memory Computing (IMC) in 22 nm technology.

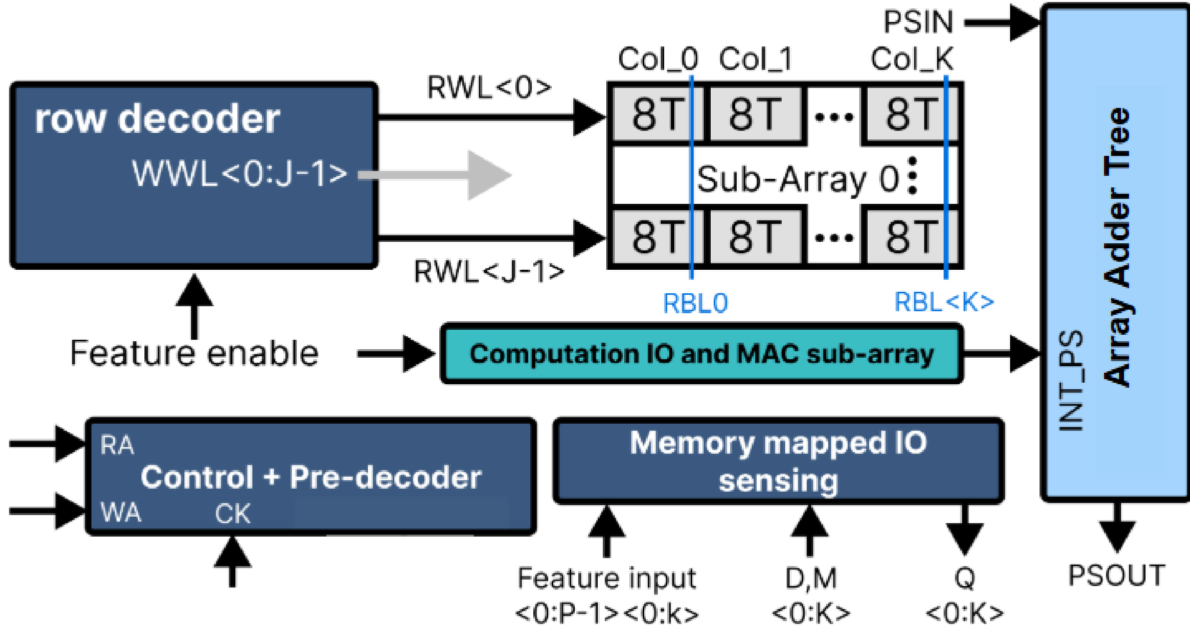


Figure 4.4: Simulated digital In-Memory Computing (IMC) array, including the integrated adder-tree for partial-sum accumulation.

Each IMC array measurement includes the energy associated with the following fundamental operations:

- **Read:** corresponding to the MAC computation phase within the cell;
- **Write:** representing operand loading into the cell array;
- **Hold:** accounting for leakage energy during idle states, holding the value stored.

These parameters, together, enable the fine-grained estimation of both the computational and memory-related contributions of the IMC architecture to total energy and latency.

The acquired raw data include:

- **Latency:** measured as the number of clock cycles required per operation (1 ns per row read or write);
- **Energy:** computed as the time integral of the current drawn by the V_{DD} generator during each operation.

Each IMC operation—*read*, *hold*, and *write*—was simulated under multiple voltage and sparsity/switching activity conditions. The experimental characterization distinguishes between sparsity (used for *read*/*hold*) and switching activity (used for *write*) to capture realistic workload variations.

In particular, voltage levels range from $0.45V$ to $0.80V$ (this latter being the nominal value of the 22 nm PTM). Each supply voltage V_{DD} was associated with a corresponding reference voltage V_{ref} for the SRAM peripheral circuitry, increasing approximately linearly from $0.34V$ to $0.6V$.

All operating points were characterized under the same switching activity distributions. Specifically, Table 4.1 presents all the simulation values for each parameter.

Table 4.1: Available design parameters for the target arrays. For the smallest 4×4 arrays, only subsets of activity levels were considered due to limited cells: sparsity 25 %, 50 %, 75 % (*read*/*hold*) and switching 25 %, 50 %, 75 %, 100 % (*write*); other combinations were omitted (NaN).

Parameter	**Values**
Array size	4×4, 16×16, 24×24, 32×32
Switching activity (%)	13, 25, 38, 50, 63, 75, 88, 100
Sparsity (%)	13, 25, 38, 50, 63, 75, 88
V_{DD} **(V)**	0.45, 0.50, 0.55, 0.60, 0.65, 0.70, 0.75, 0.80
$V_{\mathbf{REF}}$ **(V)**	0.34, 0.38, 0.41, 0.45, 0.49, 0.53, 0.56, 0.60

As shown in Figure 4.5, the simulation results clearly confirm the expected quadratic dependence of energy consumption on the supply voltage (V_{DD}). This behavior aligns with theoretical predictions based on dynamic power dissipation models, where the energy consumption scales proportionally to V_{DD}^2. The observed trend validates both the correctness of the simulation setup and the physical consistency of the modeled IMC array, reinforcing the reliability of the obtained results.
This dataset serves as the foundation for constructing polynomial interpolation models, which are used to estimate energy and latency values for previously unseen configurations during design space exploration.

During the interpolation phase, a **polynomial fitting** approach was identified as the most suitable method to capture the non-linear behavior observed in the simulated IMC data. The fitting process was performed using the *MATLAB Curve Fitting Toolbox*, which enabled accurate approximation of the energy trends across different voltage levels, array sizes, and sparsity conditions. An example of the resulting polynomial fitting, compared against the original simulated data, is shown in Figure 4.6.

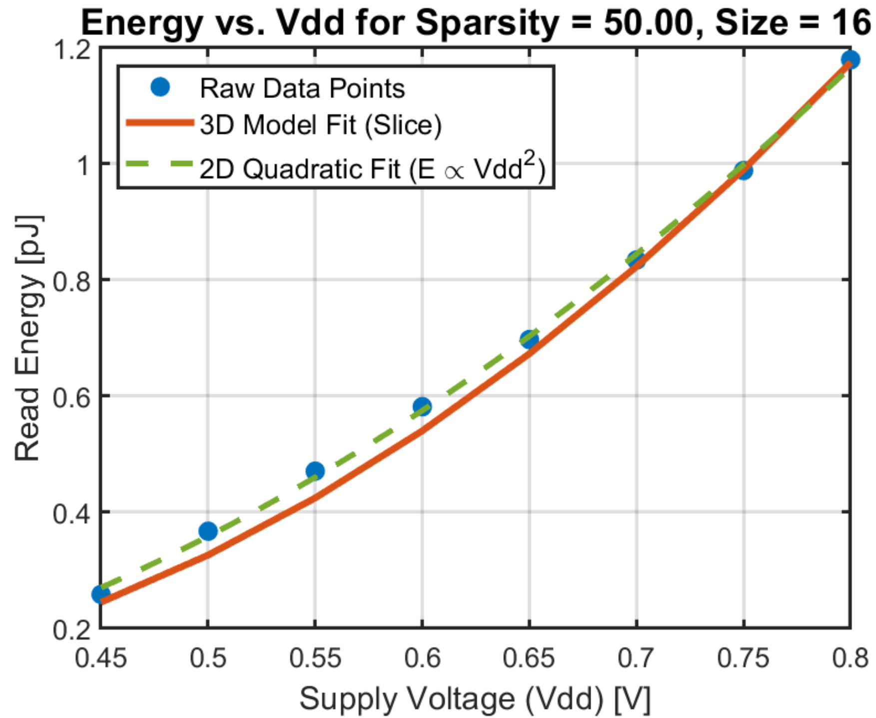


Figure 4.5: Relationship between energy consumption and V_{DD}, illustrating the characteristic quadratic dependency.

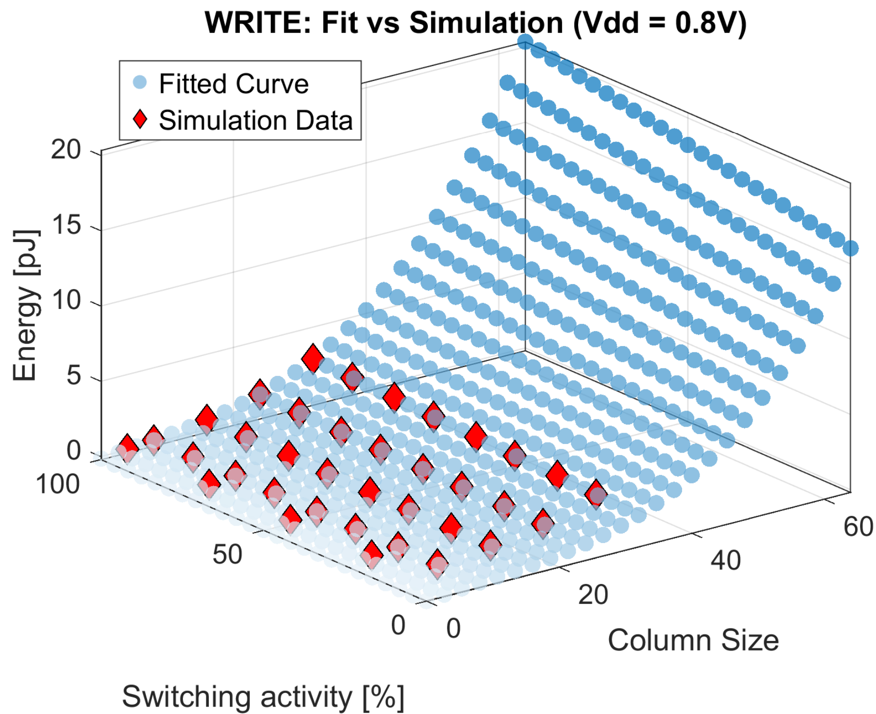


Figure 4.6: Polynomial fitting curve (set of blue points) compared with simulation data (red points) for the write operation with $Vdd = 0.8V$.

From the polynomial interpolation step, a separate model was generated for each V_{DD} value present in the raw simulation dataset. Each polynomial captures the dependence of the energy consumption on the *sparsity* (or switching activity) of the workload, as well as on the IMC array size. These fitted models were exported and are dynamically loaded during the configuration of IMC compute levels within the design space exploration workflow.

The selection of the appropriate polynomial model is based on the parameter values of the architecture under evaluation. In the proposed framework, these parameters are determined as follows:

- The **IMC array size** is explicitly defined during the architecture modeling phase.
- The **sparsity or switching activity** can be provided in two ways:
  1. **User-defined**, allowing manual specification for custom workloads.
  2. **Automatically inferred** for standard CNN workloads such as *ResNet-18* and *VGG-16*. Since pretrained models rarely contain exact zeros, sparsity is estimated by computing the proportion of weights with magnitude below 10^{-5}.

For unknown or generic workloads—where no reliable sparsity information is available—a default sparsity level of 20% is adopted. This value corresponds to the average sparsity observed across common CNN architectures, ensuring consistent model selection during the exploration process.

At runtime, the corresponding energy value for a specific configuration—defined by array size, workload sparsity, and switching activity—is computed by evaluating the appropriate polynomial model.
Table 4.2 reports examples of the resulting energy estimations for different configurations of IMC arrays operating at different V_{DD} values, array size, and sparsity. The listed energy values correspond to the *read* operation of the IMC cells.

Table 4.2: Read energy consumptions for different tile configurations at varying V_{DD}, array size and sparsity.

V_{DD} (V)	Array Size	Sparsity (%)	Energy (pJ)
0.45	4×4	13	0.0159
0.50	8×8	25	0.1091
0.55	16×16	38	0.4907
0.60	24×24	50	1.3246
0.65	32×32	63	2.8423
0.70	32×32	75	3.1548

These interpolated values are subsequently used to compute total energy consumption within the FactorFlow-based evaluation process, ensuring consistency between simulated data and surrogate-assisted estimations across the explored architectural configurations.

Array Adder Tree

Beyond IMC-specific enhancements, an additional refinement was introduced to improve modeling accuracy for spatial accelerators employing hierarchical fanout structures. In such designs, intermediate fanout levels often include *accumulator units*, responsible for aggregating partial sums generated across parallel compute paths. These accumulator components were modeled as **array adder trees** and integrated directly into the fanout level abstraction, with their associated energy consumption values derived from empirical data provided by the University of Ferrara.

The inclusion of the adder tree modeling ensures that the framework captures the complete energy footprint of architectures with hierarchical data-parallel structures, complementing the IMC extensions to produce a unified and physically consistent model of computation and data movement.

The computation of the energy consumed by an adder tree depends on the number of adders that compose its hierarchical tree structure. Given a base adder with a known arity (i.e., the number of inputs it can handle per instance) and known energy and latency characteristics, the total energy consumption is obtained as follows:

$$E_{\text{total}} = E_{\text{adder}} \times \left\lceil \frac{N_{\text{op}} - 1}{\text{arity} - 1} \right\rceil \times N_{\text{iter}}$$

where:

- E_{adder} is the energy consumption of a single adder instance;
- N_{op} is the total number of operands output by the fanout level;
- arity is the number of inputs that each adder can process per operation;
- N_{iter} represents the total number of iterations required at that level, including both local and hierarchical iterations.

The latency computation follows the same hierarchical structure as the energy calculation; however, since latency values are accumulated linearly through the tree depth, the formula is identical in form and is therefore omitted for brevity.

The energy and latency values of the base adders were provided by the **University of**

Ferrara, using the same experimental setup employed for the characterization of IMC cells described in Section 4.2. To accurately capture the performance of the adder, a Carry Lookahead Adder (CLA) was simulated—a widely used digital adder designed to accelerate binary addition by reducing the time required for carry propagation. Unlike simpler adders, such as ripple-carry adders, where each bit must wait for the previous carry to be computed, a CLA predicts the carry values in advance using the *generate* and *propagate* signals. This approach enables the parallel computation of carry bits across all positions in the adder, significantly reducing the critical path delay and improving overall throughput for high-speed arithmetic operations.

Figure 4.7- 4.8 illustrates the operation of a single-bit CLA, showing both the calculation of the carry (C_0) and sum (S_0) bits. The *generate* signal indicates whether a particular bit position will produce a carry regardless of the incoming carry, while the *propagate* signal determines whether an incoming carry will be transmitted to the next higher bit. Two precision configurations were modeled—**4-bit** and **8-bit** CLA adders—by replicating the single-bit CLA structure. Both implementations are based on 22nm technology, which provides a realistic reference for modern digital design and ensures accurate estimation of energy consumption and latency. The two configurations offer flexibility depending on the bit-width requirements of the target workload or architecture: smaller 4-bit adders can be used in low-precision or energy-constrained scenarios, while 8-bit adders enable higher precision arithmetic when required. These CLA models serve as fundamental building blocks for evaluating the energy and performance trade-offs in larger digital and IMC-enabled computing systems.

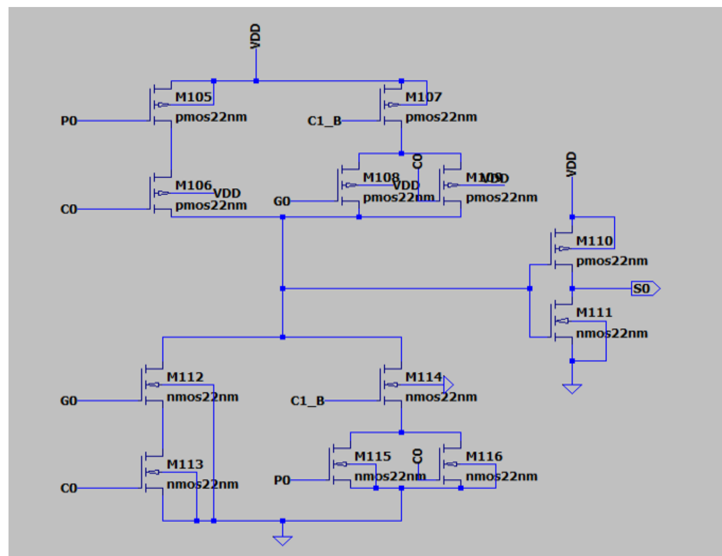


Figure 4.7: Description of CLA 1-bit adder sum circuit block implemented using CMOS current mirror logic.

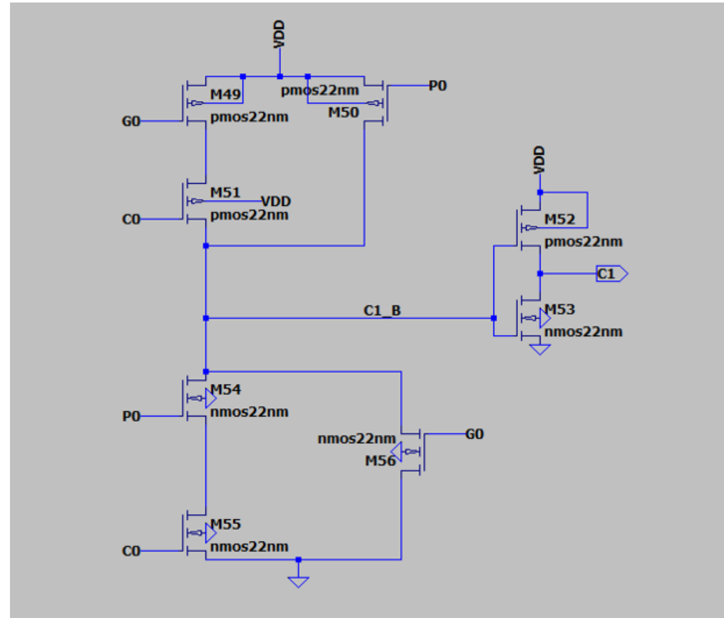


Figure 4.8: Description of CLA 1-bit adder carry circuit block implemented using CMOS current mirror logic.

Overall, these modifications allow *FactorFlow* to seamlessly handle architectures that combine conventional spatial compute hierarchies with IMC-based computational units. This extended modeling capability provides the foundation for a holistic design space exploration process that jointly optimizes both memory and compute dimensions within next-generation accelerator architectures.

4.3. Design Space Exploration Process

4.3.1. Problem Formulation and Initial Setup

Design Space Exploration (DSE) is a systematic process aimed at identifying optimal hardware configurations within a multidimensional space defined by architectural parameters and design constraints. The objective of DSE is to determine a set of configurations—typically corresponding to the **Pareto front**—that represents the best trade-offs among competing design objectives such as latency, energy efficiency, and area.

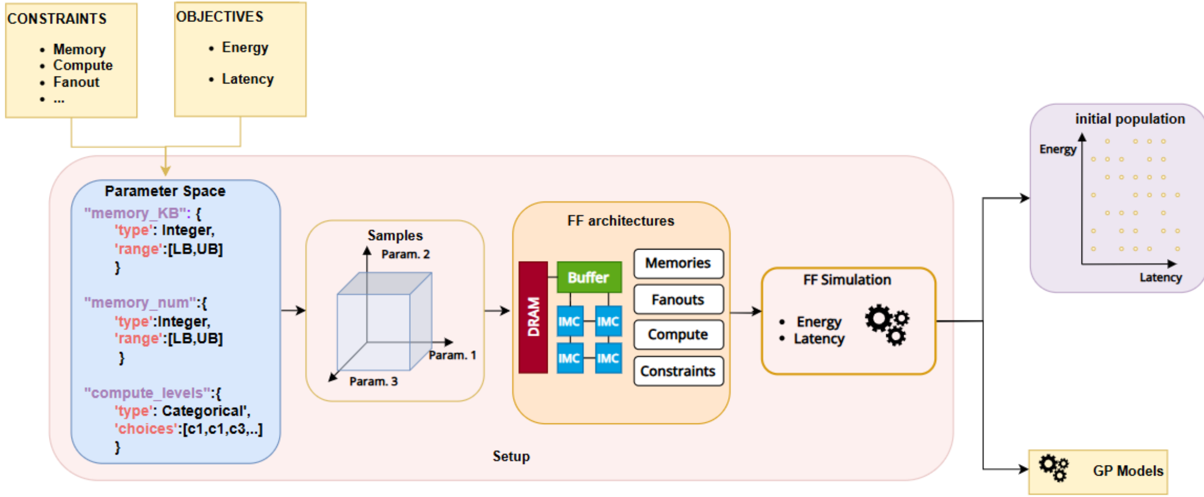


Figure 4.9: Overview of the setup phase, illustrating the workflow from parameter space sampling to FactorFlow architecture instantiation and Gaussian Process model training.

The DSE process can be divided into four main stages:

1. **Parameter Space Definition:** This stage defines the feasible subspace of the overall design space by applying user-defined constraints, creating the so-called *parameter space*. These constraints specify acceptable limits on architectural parameters, thereby restricting exploration to realistic design points that comply with hardware, performance, or resource limitations. The choice of constraints is a crucial design decision, as it directly affects both the size and the complexity of the search space. Looser constraints enable a broader exploration—potentially leading to more optimal configurations. In contrast, tighter or fixed constraints restrict the search space, reducing the likelihood of discovering unconventional yet efficient solutions.

2. **Sampling:** Once the constrained space is established, a *sampling strategy* is employed to select an initial set of design points within the feasible region of the design space. Ideally, the sampling process would explore *all* possible configurations, evaluate each one, and identify the optimal solutions according to Pareto dominance. Such a complete enumeration guarantees the discovery of the globally optimal Pareto set. However, even under constraints, the dimensionality of the design space is typically very high, making exhaustive search computationally prohibitive in terms of both time and resource requirements. Therefore, alternative and more efficient sampling strategies must be adopted. An effective sampling method should generate a diverse and well-distributed set of samples, ensuring balanced coverage of the design space while facilitating rapid convergence toward optimal solutions.

3. **Exploration:** The exploration phase represents the core of the process. It is typically based on the initial set of solutions selected during the sampling phase, which defines the starting set for the exploration. Here, the algorithm iteratively navigates the design space, selecting new candidate configurations based on previous evaluations, progressively refining the search to identify high-performing or Pareto-optimal configurations.

4. **Termination and Pareto Set Identification:** The final stage defines the stopping criteria of the exploration process and extracts the Pareto-optimal set, representing the best trade-offs among the competing objectives of interest.

The following sections describe the implementation of these stages in the proposed framework.

4.3.2. Parameter Space Definition

The exploration framework targets **In-Memory Computing (IMC)-based hardware accelerators**. The design parameters are carefully selected to capture the architectural characteristics that most significantly influence performance, energy consumption, and parallelism.

The *parameter space* is defined as the collection of all valid parameter combinations, where each parameter is characterized by its data type, domain, and range of admissible values. This formal definition serves as the foundation for both the sampling and exploration phases.

In the proposed framework, the parameter space includes:

- **Memory Capacity (KB):** An integer variable defining the total on-chip memory capacity available for computation. The off-chip DRAM is assumed to have infinite capacity, consistent with the assumption used in *FactorFlow* [28].
- **Compute Units:** Categorical variable describing the dimensions of the IMC compute array. The available categories correspond to configurations reported in Section 4.2.
- **Fanout Levels:** An integer variable representing the number of hierarchical fanout stages used to enable data parallelism across the accelerator.
- **Register Count:** An integer variable defining the number of registers available per processing unit, directly affecting data reuse and throughput.

- **Parallelism Dimension:** Fixed parameter indicating the specific axis or loop dimension along which parallel execution is applied in the architecture.

Each parameter is associated with a *range/choices field*. For integer parameters, the *range* specifies the interval of admissible values explored during sampling, whereas for categorical parameters, the *choices* field defines the set of discrete configurations available.

Some parameters are dynamically defined during the exploration process. For example, the proportion of total memory distributed across available on-chip buffers can either be specified by the user (fixed) or automatically generated at runtime as a *continuous* variable within the range $[0, 1]$, depending on the number of memory units sampled.

A similar approach is applied to the distribution of compute units and registers. In these cases, the allocation is modeled as an *integer* variable ranging from 1 to the total number of compute units or registers, effectively encoding a permutation that defines how these resources are distributed among architectural levels.

Together, these parameters form the **design vector** for each configuration in the exploration process. This vector constitutes the basis for evaluating and optimizing IMC-based accelerators in terms of performance, energy efficiency, and resource utilization.

It is also important to note that *fixed parameters*—those not included in the exploration—serve only to constrain the design space and are not part of the sampling process. These parameters are introduced a posteriori, once the initial set of sampled configurations is generated, and are used to define constant architectural characteristics that remain invariant throughout the exploration.

As summarized in Table 4.3, the design space used in this thesis is constrained between 128 and 2048 KB of total on-chip memory, distributed across 2 to 6 memory units, each with a proportional allocation of the total capacity (either explored or fixed). The compute array can assume 16, 64, 256, or 576 IMC units, distributed over 2 to 4 fanout levels. Each fanout stage is associated with a specific portion of the total compute resources, while 2 to 4 registers are optionally distributed across the compute levels following a defined permutation.

Parameter	Type	Domain / Choices
`memory_kb`	Integer	$[128, 2048]$
`memory_num`	Integer	[2, 6] .
`memory_distribution_i`	Real	[0, 1] .
`compute_levels`	Categorical	{16, 64, 256, 576}
`fanout`	Integer	[2, 4]
`compute_levels_distribution_i`	Integer	[1, 576]
`registers_num`	Integer	[2, 4]
`reg_permutation_i`	Integer	[1, 576]

Table 4.3: Summary of the architectural parameters explored during the DSE. Each entry specifies its data type and admissible domain or set of choices.

Optimization Objectives. A critical aspect of any design space exploration process is the definition of the *objectives*, which answer the fundamental question: *"What should be optimized?"* Objectives represent the measurable quantities that characterize the performance of each configuration, guiding the exploration toward optimal trade-offs. Based on these objectives, an **objective function** is defined, which determines the decision-making process of the exploration algorithm. When the goal is minimization, configurations with lower objective values dominate in the selection process.

In the context of hardware design exploration, the goal is typically to minimize computation latency and energy consumption, which together capture both performance and efficiency. Accordingly, this thesis defines two primary objectives:

- **Latency (ns):** Total execution time of the workload mapped onto the accelerator.
- **Energy (mJ):** Total energy consumption estimated across all active components during computation.

The trade-off between these two objectives defines the Pareto-optimal frontier of the explored design space. It is important to note that objectives are not included within the parameter space itself, as they are not variables to be explored but outcomes computed by the framework for each configuration. These metrics are used exclusively during dominance evaluation to determine the relative quality of candidate solutions.

4.3.3. Sampling Strategy: Latin Hypercube

As previously discussed, selecting the initial set of samples is a critical step in the overall design space exploration process. The quality and diversity of these initial samples directly influence the efficiency of the subsequent optimization phase and the likelihood of identifying Pareto-optimal configurations.

For the proposed work, the **Latin Hypercube Sampling (LHS)** technique was adopted as the primary sampling strategy. Starting even from a high-dimensional and extended design space, this method ensures a *stratified sampling* of the parameter space, dividing each of the d dimensions of a subspace D_k into N non-overlapping intervals $\{D_{k,j}\}_{j \in \{1,\dots,N\}}$, each with an equal probability of selection $1/N$. This approach reduces the variability of estimates and ensures that the entire sampling domain is covered more uniformly. Formally, each dimension of the design space satisfies [19, 30]:

$$D_k = \bigcup_{j=1}^{N} D_{k,j}, \quad \text{with } D_{k,j_1} \cap D_{k,j_2} = \emptyset \text{ for } j_1 \neq j_2.$$

Latin Hypercube Sampling can also be generalized to allow stratification on sets of orthogonal subspaces within the sample space. A "Latinized" stratified sampling approach combines the advantages of both stratified and Latin sampling schemes. Previous studies across various engineering domains have demonstrated that LHS achieves high accuracy with significantly reduced computational effort compared to crude Monte Carlo methods. In the context of hardware accelerator design, this property is particularly valuable, as it allows an efficient exploration of large, high-dimensional parameter spaces while maintaining statistical robustness of the sampled configurations.

In the context of this work, the dimensionality d corresponds to the number of parameters defined in the parameter space that are subject to exploration. Given a desired number of samples N (set as a user configuration), the sampler generates an (N, d) data structure. Each sample n is represented by a vector of values within the normalized range $[0, 1]$, where each element corresponds to a parameter of the design space. These normalized values are then mapped to their actual domains according to the parameter type:

- For **integer parameters**, the normalized values are linearly scaled and rounded to fall within the defined integer range.
- For **categorical parameters**, the normalized values are used as indices to select one of the discrete categorical options.
- For **continuous parameters** (e.g., memory distribution ratios, if explored), the

values are directly interpreted within $[0, 1]$ without discretization.

Special care is required for dependent variables, such as those representing *distributions* or *permutations*. In particular, the sampled memory distribution values are adjusted after sampling to ensure that they sum to one, thus representing valid proportions of total memory capacity. Similarly, permutation variables are normalized to ensure that their product or cumulative assignment corresponds to the total number of compute units or registers specified for that configuration. This adjustment process prevents invalid configurations (e.g., zero allocation) and guarantees consistent use of available hardware resources across all sampled architectures.

4.3.4. Architecture Configurations and Initial Population

At the end of the sampling phase, N candidate architectures are obtained. Each sampled configuration defines a potential accelerator instance characterized by specific architectural features such as total on-chip memory, number of compute levels, fanout depth, and register organization.

FactorFlow Architectures. Before these architectures can be used as initial samples for exploration, they must be evaluated and represented in a format compatible with the *FactorFlow* framework. In the proposed implementation, the sampled architectures are not initially described using the hierarchical level structure native to *FactorFlow*. Therefore, a translation step is required to convert each sampled configuration into a valid *FactorFlow*-compliant representation.

In *FactorFlow*, an accelerator architecture is described as a sequence of hierarchical levels—namely **Memory**, **Fanout**, and **Compute**—each characterized by specific parameters. This structure is maintained and extended to incorporate IMC-specific features:

- **Memory Level:** The first level to be defined, consisting of as many memory units as specified by the sampled value of `memory_num`. Each memory unit is assigned a capacity according to the sampled (or fixed) distribution ratios.
- **Fanout Level:** The number of fanout stages is determined by the `fanout` parameter. Each fanout level defines its mesh size according to the corresponding distribution of compute units.
- **Compute Level:** In *FactorFlow*, the compute level represents the processing array. For this work, each compute level corresponds to an IMC array, inherently combining memory and computation capabilities as described in Section 4.2. Although only a

single compute level is explicitly defined, its effective recurrence is governed by the mesh specifications inherited from the fanout levels.

Additional architectural parameters, such as bandwidth or access energy per level, can either be specified by the user as *additional features* or left as default values predefined in the framework. This flexible representation enables users to incorporate technology-dependent attributes as needed, while maintaining generality for exploratory analysis.

Initial Sample Evaluation. Once the architectures are correctly instantiated, each one is evaluated within the *FactorFlow* framework. The objective values computed—namely energy consumption and latency—are then associated with the corresponding sampled configuration, producing a fully annotated dataset of initial samples.

To ensure a comprehensive evaluation, each architecture is assessed under multiple representative workloads. In this work, approximately twenty workloads were selected, corresponding to all convolutional layers of two canonical deep neural networks: **ResNet-18** and **VGG-16**. These models are widely adopted as benchmarks in both academia and industry, as their convolutional layers capture the dominant computational characteristics of modern deep learning workloads. Evaluating the candidate accelerators on these workloads provides a realistic and general assessment of architectural performance across a diverse range of memory and compute patterns.

While users may optionally define custom workloads, relying on a single network or layer for exploration is discouraged. Optimizing exclusively for one workload can result in architectures that are overly specialized and lack generalization—a condition undesirable in practical accelerator design.

The resulting evaluations collectively form the **initial sample set** of the design exploration algorithm. Each sample occupies a specific point in the design space, characterized by its computed energy–latency trade-off. These samples serve as the foundation for subsequent evolutionary exploration, guiding the search toward regions of the space that exhibit promising performance–efficiency characteristics.

4.3.5. GPR Model Training

To achieve sufficient coverage of the design space, the initial sample set must include a large number of candidate configurations—typically in the order of hundreds—each evaluated under approximately twenty representative workloads. Although the *FactorFlow* framework performs these evaluations efficiently, the cumulative cost of simulating hundreds of configurations across multiple workloads remains significant, especially when

computational resources are limited.

To efficiently guide the exploration process and avoid exhaustive evaluations of all candidate configurations, this thesis employs a surrogate modeling strategy based on **Gaussian Process Regression (GPR)**.

GPR was selected due to its ability to provide both accurate mean predictions and uncertainty quantification for each estimate, which is essential in guiding the balance between exploration and exploitation during multi-objective optimization. Moreover, GPR models are highly sample-efficient, achieving reliable performance even with a limited number of training samples—typically on the order of a few hundred—making them particularly suitable for design-space exploration tasks where each sample evaluation incurs a non-negligible computational cost [33]. The purpose of the surrogate model is to learn an analytical approximation of the relationship between the architectural parameters (input features) and the corresponding performance metrics (objectives), namely *energy consumption* and *latency*. This model provides both the predicted mean and uncertainty of each configuration.

Kernel Selection and Model Training. Gaussian Process models are fitted independently for each objective (energy and latency). To identify the most suitable model, the framework automatically tests multiple covariance kernels. It selects the one yielding the highest **Log Marginal Likelihood (LML)**, a standard metric in Bayesian regression assessing model fit and generalization capability. The tested kernels include:

- **Radial Basis Function (RBF)** kernel — suited for smooth, stationary functions.
- **Matern** kernels with $\nu = 1.5$ and $\nu = 2.5$ — capable of modeling less smooth or non-stationary behaviors.

Each kernel is combined with a **Constant Kernel** for global scaling and a **White Noise Kernel** to account for observation noise. The model is optimized using multiple random restarts of the internal hyperparameter optimizer (`n_restarts_optimizer = 10`) to avoid local minima.

The best-performing model is retained for each objective, and the kernel parameters (e.g., characteristic length scale, noise variance) are stored for subsequent prediction and analysis.

Optional Incremental Model Updating: As the exploration process progresses and new architectures are validated, their results can optionally be integrated into the training dataset. When enabled, newly validated samples are added to the training dataset, and

both GPR models are retrained from scratch on the expanded dataset. Although this retraining ensures global model consistency and allows the surrogate to capture emerging trends in the design space, it is inherently computationally expensive. Each retraining involves re-optimizing the kernel hyperparameters, which scales cubically with the number of samples ($O(N^3)$). Therefore, incremental updates are performed only at specific intervals or when a significant amount of new data is available, ensuring a balanced trade-off between model accuracy and runtime efficiency.

Prediction and Uncertainty Estimation. Once trained, the GPR models can predict the expected energy and latency of new, unseen architectures using the `gaussian_predict()` function. Given a new batch of design samples, the function:

1. Applies the same preprocessing transformations (scaling and one-hot encoding) used during training.
2. Computes the posterior mean and standard deviation for both objectives in the log-normalized space.
3. Inverts the scaling and logarithmic transformations to recover predictions in the original energy (mJ) and latency (ns) domains.

The model thus provides:

- μ_E, μ_L — predicted mean energy and latency;
- σ_E, σ_L — predictive uncertainty for each estimate.

These uncertainty estimates are crucial for multi-objective optimization, as they enable the implementation of acquisition functions such as the *Lower Confidence Bound (LCB)* or *Expected Improvement (EI)* to balance exploration and exploitation.

Integration within the Exploration Workflow. The Gaussian Process predictor acts as a statistical surrogate between the sampling phase and the evolutionary optimization algorithm. Rather than re-simulating every configuration through the *FactorFlow* evaluation pipeline, which would be computationally expensive, the trained GPRs provide fast and differentiable estimates of performance metrics. This allows the exploration algorithm to evaluate thousands of design points per iteration while maintaining accuracy and model reliability.

The dual-model setup (one for energy and one for latency) is particularly suitable for the multi-objective nature of this work, where the ultimate goal is to identify Pareto-optimal architectures offering the best trade-off between energy efficiency and execution time.

4.4. Multi-Objective Evolutionary Exploration Algorithm

Once the parameter space has been defined and the Gaussian Process (GP) surrogate models have been trained, the next step of the proposed methodology focuses on the actual exploration process. This stage introduces the **Multi-Objective Evolutionary Algorithm (MOEA)**, which constitutes the central decision-making component of the framework. Its purpose is to explore the design space efficiently, guided by the surrogate models, to identify architectural configurations that offer the best trade-offs between conflicting design objectives—specifically, minimizing *energy consumption* and *latency*.

In the context of hardware accelerator design, the optimization problem is inherently *multi-objective*: an architecture that minimizes energy consumption may not necessarily achieve the lowest latency, and vice versa. Moreover, the design space is typically characterized by non-linear and discontinuous relationships among variables, which makes gradient-based or deterministic approaches ineffective. For this reason, the exploration methodology relies on *evolutionary algorithms*, a class of population-based metaheuristics that are particularly well-suited for high-dimensional, non-convex optimization. Evolutionary algorithms are inspired by the mechanisms of natural selection, where populations of candidate solutions evolve over successive generations through operations analogous to reproduction, mutation, and survival of the fittest. Their intrinsic stochasticity, coupled with their ability to simultaneously handle multiple objectives, makes them especially effective for design-space exploration problems such as those considered in this work.

Among evolutionary paradigms, **Genetic Algorithms (GAs)** are a natural fit. They operate by iteratively evolving a population of candidate solutions through a cycle of selection, crossover, and mutation, guided by objective functions that measure the performance of each individual. Through this iterative process, the population progressively converges toward regions of the search space associated with high-performing designs.

For the proposed methodology, the selected algorithm is the **Non-dominated Sorting Genetic Algorithm II (NSGA-II)** [5], one of the most widely adopted algorithms for multi-objective optimization. NSGA-II is characterized by three fundamental mechanisms—*fast non-dominated sorting*, *crowding-distance computation*, and *elitist selection*—that allow it to approximate the Pareto-optimal front efficiently. These features ensure both convergence toward high-quality solutions and the preservation of diversity among individuals, preventing premature convergence and promoting exploration across the entire design landscape. This balance between exploration and exploitation makes

NSGA-II particularly suitable for the present framework, where both the accuracy and diversity of the architectural candidates must be preserved to investigate the trade-off surface between performance and energy efficiency thoroughly.

NSGA-II pairs well with surrogate models because its population-based evaluations can be efficiently estimated by the GPR predictor, allowing for numerous candidate assessments per generation while preserving the algorithm's diversity-preserving mechanisms.

Before delving into the implementation details, it is useful to introduce the terminology and conceptual foundations that will be consistently used throughout this section. In genetic algorithms, the optimization process revolves around the concepts of *generation* and *population*, where each generation represents a new evolutionary stage of the exploration. The terminology employed in this thesis is summarized below.

- **Population (P_t):** The population represents the set of all candidate design configurations considered at generation t. Each element of the population encodes a unique configuration of architectural parameters—such as the number of memory units, compute levels, fanout stages, and register allocations—and is associated with objective values (energy and latency) either obtained through full simulation via the *FactorFlow* framework or estimated through the trained GPR surrogate models. The initial population, P_0, corresponds to the set of samples defined in Section 4.3.1, and serves as the foundation upon which subsequent generations build.
- **Survivors or Elites:** A key feature of NSGA-II is its *elitist* nature, which allows the algorithm to preserve the best-performing solutions across generations. These solutions—often referred to as *elites* or *survivors*—are those that exhibit the most favorable trade-offs among objectives according to the NSGA-II dominance and diversity criteria. Retaining these individuals ensures that the evolutionary process does not lose valuable information about the best regions of the design space, while still allowing for exploration through the creation of new candidate configurations. Throughout this thesis, the terms *elites* and *survivors* will be used interchangeably.
- **Parents (p):** From the current population P_t, a subset of individuals is selected to serve as *parents* for the next generation. The selection process favors individuals with superior fitness—i.e., those belonging to higher Pareto fronts and exhibiting greater crowding distances—thus promoting both quality and diversity. Each parent contributes to the creation of one or more offspring through crossover and mutation operations, which combine and slightly perturb the parent parameters to explore new regions of the design space.

- **Offspring (Q_t):** The newly generated individuals, derived from parent solutions, form the *offspring population* Q_t. Each offspring represents a potential new architectural configuration, created through random variations of the parent parameters. Their objective values are estimated using the Gaussian Process models trained earlier, thus enabling rapid evaluation without the need for costly full simulations. Optionally, a small subset of offspring per generation can be evaluated with full *FactorFlow* simulations to validate surrogate predictions and provide data points for periodic retraining. This surrogate-assisted evaluation strategy ensures that the exploration remains computationally tractable even for large populations.
- **Fitness:** The term *fitness* refers to the measure of performance used to compare and rank individuals. In the case of NSGA-II, the fitness of a solution is determined by its dominance rank (Pareto front level) and its crowding distance. While the term originates from classical GA terminology, in this context it directly corresponds to the position of a design in the multi-objective ranking.

The following subsections describe the mechanisms governing the operation of the proposed evolutionary exploration algorithm, including the elitism and survivor-selection strategies, the mutation and reproduction processes, and the iterative convergence behavior. Finally, the section concludes by outlining the termination conditions and explaining how the final Pareto-optimal front is extracted from the evolved populations.

4.4.1. Evolutionary Process

NSGA-II belongs to the family of *elitist* evolutionary algorithms. The term *elitist* refers to the algorithm's capability to retain the best solutions found so far, preventing them from being lost due to stochastic variations in subsequent generations. This property is particularly valuable in the context of hardware design exploration, where each evaluated configuration represents a non-trivial computational effort. By preserving the top-performing solutions, the algorithm ensures steady progress toward convergence, while simultaneously using them as reference points to guide the creation of new candidate architectures.

To identify these elite solutions, NSGA-II relies on two complementary ranking mechanisms: *non-dominated sorting* and the *crowding-distance metric*. Together, these metrics enable the algorithm to order individuals in a population not only based on their objective performance, but also according to their contribution to the overall diversity of the population. A graphical overview is presented in Figure 4.10.

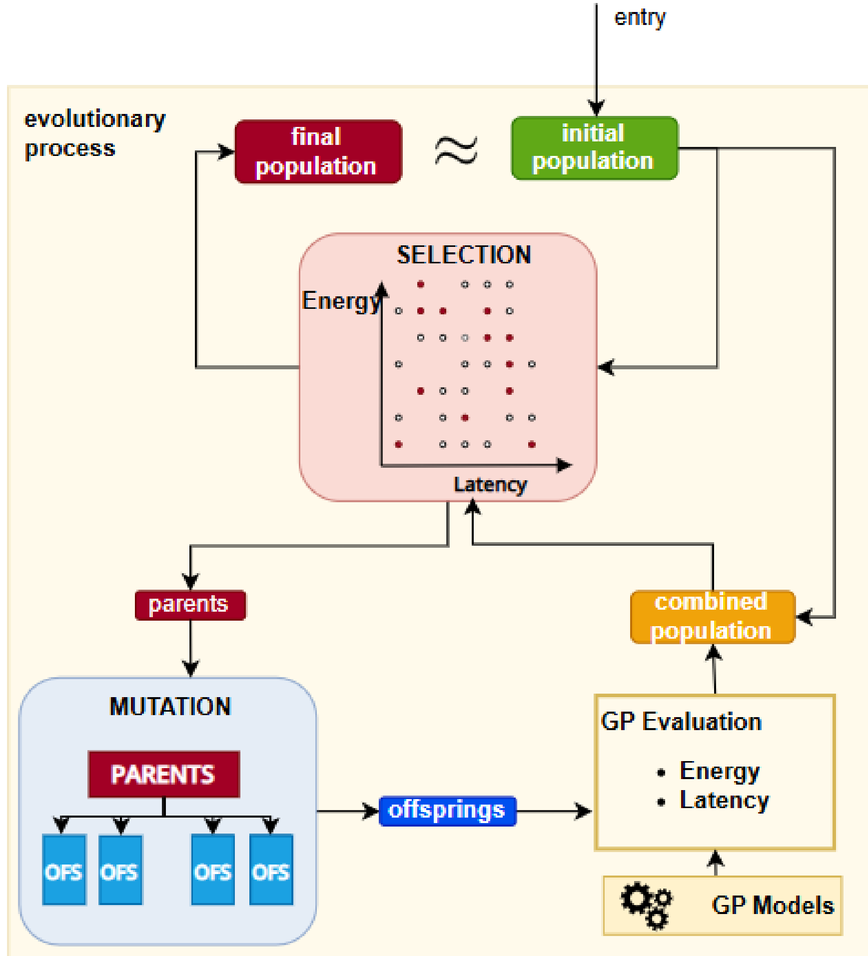


Figure 4.10: Evolutionary exploration workflow: population initialization via LHS, surrogate-assisted offspring evaluation (GPR), selection and mutation (NSGA-II).

Selection

Survivors In a multi-objective context, the notion of "best" solution is inherently ambiguous, since improvements in one objective (e.g., latency) may worsen another (e.g., energy). To address this, NSGA-II organizes solutions into multiple dominance-based layers called *Pareto fronts*. Formally, let $F = \{f_1, f_2, \dots, f_M\}$ be the set of objective functions to minimize, and $\mathbf{x}_a$ and $\mathbf{x}_b$ two candidate solutions. A solution $\mathbf{x}_a$ is said to *dominate* $\mathbf{x}_b$ if:

$$\forall i \in [1, M] : f_i(\mathbf{x}_a) \leq f_i(\mathbf{x}_b) \quad \text{and} \quad \exists j \in [1, M] : f_j(\mathbf{x}_a) < f_j(\mathbf{x}_b)$$

That is, $\mathbf{x}_a$ performs no worse in all objectives and strictly better in at least one. The first Pareto front F_1, therefore, contains all non-dominated solutions, while the second F_2 contains solutions dominated only by those in F_1, and so on. This non-dominated sorting is implemented efficiently in NSGA-II through a nested-loop algorithm that tracks for each solution its *domination count* and the set of other solutions it dominates.

While non-dominated sorting determines the hierarchy of fronts, it does not distinguish between solutions within the same front. For this purpose, NSGA-II introduces the *crowding-distance metric*, which quantifies the degree of isolation of each individual in the objective space. Given a front, solutions are first sorted with respect to each objective. Boundary solutions (those with minimum and maximum objective values) are assigned an infinite distance to ensure their preservation, as they represent the extreme trade-offs of the Pareto front. For every intermediate solution i, the normalized crowding distance is computed as:

$$d_i^m = \frac{f_m(i+1) - f_m(i-1)}{f_m^{\max} - f_m^{\min}}$$

and the total distance is obtained as:

$$\text{CrowdingDistance}(i) = \sum_{m=1}^{M} d_i^m$$

Solutions with higher crowding distances are preferred, since they lie in sparsely populated regions of the objective space, contributing to diversity and exploration.

By combining non-dominated sorting and crowding distance, the algorithm can rank all individuals in a population. A fixed number S of top-ranked solutions is retained as the elite subset for the next generation.

Parents Once the survivors are determined, the algorithm proceeds to generate the next population through the standard genetic operations: selection, crossover, and mutation. Parent selection is carried out using a *tournament-based* strategy. In this approach, random pairs of individuals are drawn from the elite pool, and each is assigned a random score. The individual with the lowest score is then selected as a parent. This procedure is repeated until the required number of parents is obtained, ensuring a balance between preserving high-performing individuals and maintaining population diversity.

It is worth noting that this step is not depicted in the schematic representation, as parent selection occurs only at the beginning of each iteration—when new parents are required—and not at the end of the evolutionary cycle.

Mutation and Offspring Generation

Each parent p contributes to the creation of one or more offspring through a stochastic variation process known as *mutation*. Mutation plays a central role in maintaining diversity and avoiding premature convergence toward suboptimal regions of the design space. It acts as a controlled source of randomness, introducing small perturbations into the architectural parameters of a parent configuration to explore nearby regions of the search space. In the context of design-space exploration for hardware accelerators, this mechanism allows the algorithm to test new combinations of architectural features—such as different memory distributions or compute-level arrangements—while leveraging information gathered from previous generations.

For continuous parameters, mutation is performed by adding a normally distributed perturbation to the parent value:

$$x' = x + \mathcal{N}(0, \eta^2)$$

where η defines the *mutation power*, i.e., the standard deviation of the Gaussian distribution. This parameter directly controls the magnitude of the variation applied to each design variable and is defined as:

$$\eta = (x_{\text{max}} - x_{\text{min}}) \cdot r_{\text{mut}}$$

where r_{mut} (mutation ratio) determines the relative amplitude of the mutation with respect to the parameter's admissible range, typically $r_{mut} \in [0.05, 0.2]$.

The role of the mutation power. The mutation power controls the intensity of the perturbation applied to the decision variables during mutation, and therefore determines how far an offspring can deviate from its parent. In the proposed framework, this hyperparameter plays a key role in shaping the **diversity** and **refinement quality** of the candidate solutions. A high mutation power (η large) introduces stronger perturbations, producing offspring that differ more substantially from their parents. This increases the variability of the population and may help uncover new local configurations, but it often results in noisier refinement and less stable improvements from one generation to the next. Conversely, a low mutation power (η small) restricts the variation around each parent, favoring more progressive and controlled adjustments to the design parameters. While this improves local refinement, it also reduces the variety of candidate solutions, making the algorithm more sensitive to small local minima. Thus, selecting an appropriate mutation power is essential to maintain a sufficient degree of variability without sacrificing the stability of improvements across generations. Empirically, moderate mutation ratios

r_{mut} offered a reliable balance between maintaining diversity within the population and ensuring consistent refinement of the Pareto front.

For discrete or categorical parameters, mutation is implemented through random reselection within the valid domain of values. This ensures that the modified parameter remains valid and coherent with architectural constraints. When structural parameters change (e.g., the number of memory units), dependent parameters (memory distributions and permutations) are regenerated by a deterministic normalization procedure to preserve consistency (e.g., re-normalizing distribution ratios so that they sum to one).

The number of offspring O generated per generation is fixed at initialization. Each offspring represents a new architectural configuration and is subsequently evaluated using the Gaussian Process predictors, which estimate its corresponding energy and latency metrics. This surrogate-based evaluation significantly reduces computational overhead, allowing a greater number of design iterations within the same time budget.

The role of the offspring amount The number of offspring generated at each generation is a key design choice for the overall quality of the methodology. It governs the balance between *exploitation* and *exploration*—a core concept in all evolutionary algorithms[4]—by determining how effectively the algorithm investigates new regions of the solution space instead of remaining close to the initial samples. In particular, a larger number of offspring increases the algorithm's ability to explore, providing a broader view of the design space and allowing it to move sufficiently far from the initial candidate solutions. Conversely, when too few offspring are generated, the algorithm tends to exploit the existing solutions without sufficiently expanding the search, thus limiting its ability to discover new and potentially better solutions. This concept differs from the previously discussed mutation power, which concerns the local refinement and uniformity of newly generated solutions. However, the two aspects are related: during a highly explorative phase, a very low mutation power may slow down the exploration process. Still, the primary driver of exploration remains the number of generated solutions, while their quality is a consequential effect.

4.4.2. Iteration and Termination

Once the offspring population Q_t has been generated and evaluated, it is merged with the parent population P_t to form a combined set $R_t = P_t \cup Q_t$. Non-dominated sorting and crowding-distance ranking are then reapplied to R_t, and the top N solutions are retained to form the next generation:

$$P_{t+1} = \text{Survivor_Selection}(R_t, N)$$

This iterative process continues for a fixed number of generations G, defined as a hyper-parameter of the exploration.

Over successive generations, the population evolves toward convergence, gradually approximating the Pareto-optimal front that represents the best possible trade-offs between energy and latency. At the end of the exploration, the final Pareto front is extracted by applying non-dominated sorting to the last population and selecting only the first front F_1, which contains the optimal configurations according to the chosen objectives.

The resulting Pareto front provides a clear visual and analytical summary of the design-space exploration outcome. Each point on the front corresponds to a valid architectural configuration, representing a different compromise between energy efficiency and computational speed. This set of solutions enables the designer to make informed decisions based on the target application's requirements and constraints, thereby closing the loop of the proposed co-design methodology.

4.4.3. Optional Optimizations

Despite the inherent efficiency and robustness of NSGA-II, the optimization process can further benefit from additional control mechanisms designed to enhance convergence stability, adaptability, and exploration efficiency. In this work, the evolutionary workflow has therefore been extended with a set of **optional optimizations**, conceived to assess whether complementary heuristic strategies could improve the algorithm's overall performance in the specific context of Design Space Exploration (DSE) for In-Memory Computing (IMC) architectures.

These optional mechanisms are centered on controlling the *mutation power*, a key parameter that governs the trade-off between exploration and exploitation. As previously discussed in Section 4.4.1, the mutation power determines the scale of variations applied during the creation of new candidate solutions. Controlling its evolution over time allows the algorithm to dynamically adjust its search behavior in response to the exploration's progress, preventing stagnation and improving diversity preservation. Three adaptive mechanisms were implemented and analyzed: **Mutation Adaptation**, **Mutation Decay**, and **Simulated Annealing-based Adaptation**.

Mutation Adaptation. As detailed earlier, the mutation power directly influences how far new offspring can deviate from their parents in the design space. In standard Genetic Algorithms, this parameter is typically fixed at the beginning of the optimization and remains constant throughout the process. However, a static definition fails to reflect

the evolving nature of the search, where the population progressively shifts from global exploration in the early stages toward local refinement near convergence. To overcome this limitation, the **Mutation Adaptation** mechanism introduces a dynamic adjustment of the mutation power based on the observed progress of the evolutionary process.

The concept of *progress* is measured using the **Hypervolume** indicator, a well-established metric in multi-objective optimization that quantifies the portion of the objective space dominated by the current Pareto front with respect to a reference point r.

The relative improvement of the exploration between two generations is computed as:

$$\Delta HV_t = HV(F_t, r) - HV(F_{t-1}, r)$$

where $HV(F_t, r)$ refers to the hypervolume computed on the Pareto front of generation t with r as refer point.

If the algorithm exhibits an improvement ($\Delta HV_t \geq 0.1$), the mutation power η_t is slightly increased to promote further exploration. Conversely, when no progress is observed, η_t is reduced to encourage local refinement. Formally:

$$\eta_{t+1} = \begin{cases} \eta_t \cdot (1 + \alpha), & \text{if } \Delta HV_t > 0 \\ \eta_t \cdot (1 - \alpha), & \text{otherwise} \end{cases}$$

where α is a small adaptation coefficient (typically in the range $0.05 \leq \alpha \leq 0.2$). This adaptive mechanism enables the exploration to self-regulate its exploratory aggressiveness based on quantitative feedback, achieving a balance between discovering new regions of the design space and refining promising areas.

Mutation Decay. At the beginning of the evolutionary process, the Latin Hypercube Sampling (LHS) ensures that the initial population is broadly and uniformly distributed across the design space. In this early stage, a high mutation power is desirable, as it encourages the generation of candidate architectures that are significantly different from the initial solutions, thus maximizing coverage and promoting exploration. As the search progresses, however, the algorithm naturally approaches high-quality regions of the Pareto front, where significant mutations become counterproductive, potentially disrupting well-performing solutions.

To handle this transition in a simple yet effective way, the proposed framework incorporates the concept of **Mutation Decay**. This mechanism enforces a deterministic, monotonically decreasing schedule for the mutation power, independent of any dynamic

feedback or progress metric. Formally, the mutation power at generation t is defined as:

$$\eta_{t+1} = \eta_t \cdot \delta$$

where η_0 is the initial mutation power and δ is the fixed decay rate, typically constant (e.g., 0.95-0.99). This exponential decay ensures that the algorithm progressively shifts from exploration to exploitation over time: early generations perform broad exploratory jumps, while later generations focus on local optimization around the most promising designs.

Simulated Annealing-based Adaptation. While Mutation Adaptation and Decay provide deterministic control mechanisms, a third optional optimization introduces a stochastic adjustment method inspired by the **Simulated Annealing (SA)** paradigm. Although SA is traditionally employed as a standalone heuristic optimization method (see Section 2.3.4), its probabilistic acceptance rule can be integrated within NSGA-II to regulate the balance between exploration and exploitation in a meta-heuristic fashion.

In this adaptation scheme, the mutation power evolves according to a probabilistic rule governed by an *annealing temperature* T_t, which decreases progressively over generations according to:

$$T_{t+1} = T_t \cdot (1 - \beta)$$

where $\beta \in (0, 1)$ represents the cooling rate. At each iteration, a random number $r \in [0, 1]$ is generated, and the mutation power is updated as:

$$\eta_{t+1} = \begin{cases} \eta_t \cdot (1 - \gamma), & \text{if } r < e^{-\frac{1}{T_t}} \\ \eta_t \cdot (1 + \gamma), & \text{otherwise} \end{cases}$$

where γ defines the adjustment amplitude. This stochastic rule ensures that, during the early stages of the search (when T_t is high), large mutations are more probable, promoting exploration; as the temperature decreases, the algorithm becomes more conservative, favoring local exploitation around the most promising regions.

Retraining. A known limitation of Gaussian Process (GP) surrogate models arises when they are required to predict the performance of configurations that deviate significantly from those included in the initial training set. In such cases, the GPR predictions may not only become highly uncertain, but also yield values that are physically unrealistic or infeasible for practical architectural configurations. While this issue could theoretically be mitigated by constructing a very large initial training dataset, doing so would entail

evaluating a correspondingly large number of samples through full simulations in the *FactorFlow* framework—an approach that is computationally expensive and time-consuming.

To alleviate this problem, the proposed framework includes an optional optimization step referred to as **Retraining**. Retraining introduces a dynamic refinement of the GPR models during the evolutionary process, ensuring that the surrogate predictors remain consistent with the progressively explored regions of the design space. Specifically, at fixed intervals—every k generations (e.g., $k = 5$), with k defined as a static hyperparameter—the algorithm selects a small subset of representative solutions from the current population. These are typically composed of a few of the best-performing and worst-performing individuals (e.g., four in total), which are then re-evaluated through full *FactorFlow* simulations to obtain accurate ground-truth values of energy and latency. The newly obtained evaluations are subsequently added to the GP's training set, and the model is retrained over this expanded dataset. Choose k to balance fidelity and cost: smaller k improves surrogate accuracy but increases runtime.

This periodic retraining procedure enables the surrogate model to remain *up to date* with the evolving search trajectory, improving its predictive accuracy and reliability as the exploration progresses. Consequently, the GPR predictions for new offspring become more consistent with the actual underlying performance trends of the design space, reducing the likelihood of misleading or implausible estimations.

However, it is important to note that this optimization also introduces a significant computational overhead. Each retraining step requires temporarily pausing the evolutionary loop to perform several costly evaluations through the FactorFlow framework and to refit the GPR model, both of which may increase total runtime. Therefore, retraining should be enabled only when high predictive fidelity is critical for convergence quality, or when the exploration involves highly dynamic design regions where the GPR model's generalization capability tends to degrade rapidly.

4.5. Implementation

This section presents the technical implementation of the proposed methodology, outlining its software structure, configuration management, design choices, and implementation constraints.Due to the strong integration with the *FactorFlow* framework, several design and implementation decisions are directly derived from the framework's internal architecture and computational model.

4.5.1. Programming Environment and Framework Integration

The proposed methodology has been developed in Python 3.13t[25], an experimental release that introduces the free-threading feature, removing the Global Interpreter Lock (GIL) and enabling genuine parallel execution across threads.

This capability significantly enhances performance for concurrent workloads—especially simulation-heavy and data-intensive tasks such as those in the FactorFlow evaluation pipeline. The choice of this environment stems from the need to ensure seamless compatibility with *FactorFlow*, which achieves high throughput through a multi-threaded execution model.

Nevertheless, this decision introduces several practical constraints. Several third-party machine learning libraries, including `scikit-learn`[23], are not yet fully compatible with Python 3.13t or GIL-free execution. Since these packages are fundamental to the surrogate modeling component, particularly for the Gaussian Process (GP) training and prediction phases, an alternative solution was required.

To address this limitation, the implementation adopts a **multi-process isolation strategy**. A secondary virtual environment based on Python 3.12 is created and used for all machine learning–related tasks, ensuring stability and library compatibility. The rest of the framework, including the configuration logic and parallel execution of FactorFlow-based evaluations, continues to exploit the performance advantages of the free-threaded architecture. This hybrid approach provides a practical trade-off between computational efficiency and library compatibility, with only minimal inter-process communication overhead.

When the evolutionary engine—implemented within the NSGA-II module—is executed, it runs entirely inside the Python 3.12 environment. This guarantees full support for all dependencies while maintaining a responsive and scalable control structure in the primary Python 3.13t process.

4.5.2. Code Architecture and Workflow

The overall software design combines both **object-oriented** and **procedural programming** paradigms, prioritizing modularity, reusability, and extensibility. Each phase of the methodology—sampling, surrogate modeling, and evolutionary optimization—is implemented as a distinct module with clearly defined interfaces, allowing independent testing and future extensions.

Entry Point and Command-Line Interface. The exploration process begins from the `exploration_cli.py` module, which acts as the main entry point of the system. It parses user-defined arguments, initializes all configuration parameters, and launches the exploration workflow. The typical syntax for execution is:

```
-c constraints_list -o objectives_list [-p problem -algorithm
algorithm_type -features features_list]
```

The mandatory arguments define the **constraints** and **objectives** to be considered during exploration, while the remaining ones are optional and allow customization of the algorithmic behavior. Specifically:

- **Constraints:**
  1. `--memory_kb lower_bound upper_bound` — defines the total memory capacity range (in kB);
  2. `--memory_num lower_bound upper_bound` — specifies the allowable number of memory units;
  3. `--compute_levels [choices]` — defines the number of IMC computational hierarchy levels;
  4. `--memory_types [choices]` — specifies the memory technology or type;
  5. `--fanout lower_bound upper_bound` — sets the fanout or connectivity parameter range.
- **Objectives:**
  1. `--energy value` (in mJ);
  2. `--latency value` (in ns).

Optional arguments include:

- **Problem:** defines whether the exploration targets a single workload or multiple workloads;
- **Algorithm:** specifies the optimization strategy to be used (e.g., standard NSGA-II, full-space search, or sampling-only mode);
- **Features:** enables optional behaviors such as retraining, adaptive mutation, or debugging.

All parsed arguments are stored in a configuration object and passed to the main controller for initialization. This design enables easy reproducibility of experiments and supports batch execution by modifying only input parameters.

4.5.3. Configuration Management: The `Settings` Class

The `Settings` class serves as a centralized configuration unit for all algorithmic parameters and exploration-related variables. It defines default values and user-defined overrides for the following groups of parameters:

- **Algorithmic Hyperparameters:** population size, number of generations, mutation rate, mutation power ratio, and offspring count;
- **Exploration Parameters:** constraints, objectives, and workload definitions;
- **Optimization Flags:** enabling or disabling optional optimizations such as mutation adaptation, mutation decay, or retraining;
- **Logging and Output:** paths for experiment data, checkpoint frequency, and verbosity levels.

The class ensures consistency between all modules, exposing shared parameters through a unified interface. This modular configuration mechanism simplifies debugging and enhances the transparency of the exploration process.

Despite its efficiency, this class has a drawback: it must be performed via user-defined command-line inputs. When this happens, the update has to be done for each sub-process spawned. For this purposes, *Settings* class provides an `update`, called at the begging of each sub process

4.5.4. Design Choices and Multiprocessing Strategy

The `NSGA_II_Algorithm.py` module implements the core optimization logic, encompassing population management, selection, crossover, mutation, and Pareto sorting. As previously discussed, due to compatibility constraints, this module is executed entirely within the Python 3.12 environment.

Whenever model retraining is required, the system employs the `subprocess` library to spawn independent evaluation processes. Specifically, through the `subprocess.run()` command, a new subprocess is launched that executes the `worker_eval.py` script. This worker operates independently of the main process and performs a comprehensive evaluation of each architecture using the *FactorFlow* framework.

Communication between the main process and the subprocess is handled via `.json` files. The first file contains the serialized input data, representing the architectures to be evaluated, which is read by the subprocess. After the evaluation, the worker writes the results — i.e., the fully evaluated architectures — into an output file. The main process then reads this file, integrates the results, and resumes execution.

Comprehensive error-handling routines complement this mechanism to ensure reliability and ensure optimal performance. Although this design introduces a minor latency overhead, it significantly enhances process isolation, robustness, and stability during surrogate model operations.

4.5.5. Conclusions and Limitations

The final implementation achieves a balance between performance, modularity, and flexibility. The hybrid Python 3.13t/3.12 execution model allows full compatibility with machine-learning libraries while exploiting parallel execution during architecture evaluation. Although the environment-switching mechanism introduces limited overhead, this trade-off ensures that the system remains both scalable and robust.

However, the approach also presents some limitations. The experimental nature of Python 3.13t's free-threading model required extensive debugging and validation to ensure reproducibility across runs. Additionally, the inter-process communication between the main controller and the surrogate evaluation subprocess can become a performance bottleneck in large-scale explorations involving thousands of configurations. An example is that the class *Settings* must be continuously updated for each subprocess spawned. It contains all the data that defines the configuration to run the algorithm, and this data is usually changed at runtime, differing from the default one, so a continuous update is necessary.

Despite these constraints, the final system demonstrates strong scalability and adaptability, providing a reliable foundation for surrogate-assisted design space exploration tightly integrated with the *FactorFlow* framework.

5 | Experimental Results

To evaluate the efficiency, robustness, and correctness of the proposed methodology, this chapter presents a comprehensive set of experiments and comparative analyses. The results include: (i) comparisons of exploration runs performed with and without optional optimizations; (ii) evaluations using different evolutionary algorithms (EAs) integrated with multi-objective adaptations; and (iii) additional experiments exploring the effect of alternative optimization strategies on the convergence and quality of the Pareto front.

5.1. Experimental Setup

5.1.1. Comparison Algorithms

To assess the effectiveness and robustness of the proposed methodology, several additional algorithms were implemented for comparative analysis. These include a **Monte Carlo**-based exploration, a **Simulated Annealing (SA)** method with multi-objective adaptation, and a **Full Search** baseline serving as a reference for optimality. Each approach provides a distinct perspective on exploration efficiency, sampling uniformity, and convergence behavior, offering valuable insights into the performance of the main evolutionary framework.

Simulated Annealing

As introduced in Section 2.3.4, Simulated Annealing (SA) is a stochastic optimization technique inspired by the annealing process in metallurgy. Although SA is inherently designed for single-objective optimization, this thesis developed a **multi-objective adaptation** to enable comparison with the proposed evolutionary approach.

The sampling of initial configurations follows the same procedure used for NSGA-II, employing *Latin Hypercube Sampling* to ensure balanced initial coverage of the design space. Unlike NSGA-II, however, SA does not rely on the concept of generations or populations. Instead, the optimization proceeds iteratively, guided by a *weighted sum of*

objectives that combines energy and latency into a single scalar cost function:

$$C_{\text{current}} = w_{\text{energy}} \cdot f_{\text{energy}} + w_{\text{latency}} \cdot f_{\text{latency}}$$

where w_{energy} and w_{latency} represent the relative importance of the two objectives.

At each iteration, a candidate solution is randomly selected from the initial population and undergoes mutation following the same procedure used in NSGA-II. The new candidate is evaluated using the GP predictors, and its corresponding cost C_{new} is compared with the current one. Following the classical SA acceptance rule, the new solution is accepted with a probability:

$$P = \exp\left(-\frac{C_{\text{new}} - C_{\text{current}}}{T}\right)$$

where T represents the annealing temperature, which decreases gradually according to a predefined cooling schedule. This probabilistic acceptance allows occasional uphill moves (i.e., worse solutions), helping to escape local optima and maintain exploration diversity. After all iterations are completed, the Pareto front is extracted from the final set of accepted solutions.

Full Search

While heuristic and stochastic approaches provide efficient exploration of large design spaces, none can guarantee global optimality. To establish an absolute performance reference, a **Full Search** (brute-force) exploration was implemented over a significantly constrained subspace of parameters to maintain reasonable computational cost.

In this mode, the entire constrained parameter space is exhaustively enumerated, and every possible architectural configuration is evaluated using the *FactorFlow* framework. This process yields the exact Pareto-optimal solutions for the reduced search space. Although computationally expensive, it serves as a valuable ground-truth benchmark to validate the behavior and accuracy of surrogate-assisted and evolutionary exploration methods.

For comparison purposes, an adapted version of NSGA-II was also tested under the same limited configuration set. In this case, the Latin Hypercube Sampling strategy was used for initialization; however, GP predictors were intentionally disabled due to the small number of total samples, which would not be sufficient for proper model training. While this approach increases computational time, the reduced sample size keeps it manageable, providing a fair and exhaustive point of reference for performance evaluation.

5.1.2. Exploration Metrics

The selection of appropriate **evaluation metrics** represents a fundamental step in validating the effectiveness of a Design Space Exploration (DSE) methodology. In multi-objective optimization problems, such as the one addressed in this work, the ultimate goal is to approximate the actual Pareto front as closely as possible while preserving a diverse set of trade-off solutions. Consequently, the quality of the obtained Pareto front must be evaluated using well-established indicators that quantify both convergence and diversity properties.

The following sections present the primary metrics used in this work.

Pareto Hypervolume

The **Pareto Hypervolume** measures the volume of the portion of the objective space that is *dominated* by a Pareto-optimal set F_{opt} and bounded by a reference point v_{ref}[1]. It represents the Lebesgue measure of the dominated region and can be formally defined as:

$$\mathrm{HV}_{v_{\mathrm{ref}}}(F_{opt}) = \int_{\mathbb{R}^m} [y \preceq v_{\mathrm{ref}}] \left(1 - \prod_{y^* \in F_{opt}} [y^* \preceq y]\right) dy, \tag{5.1}$$

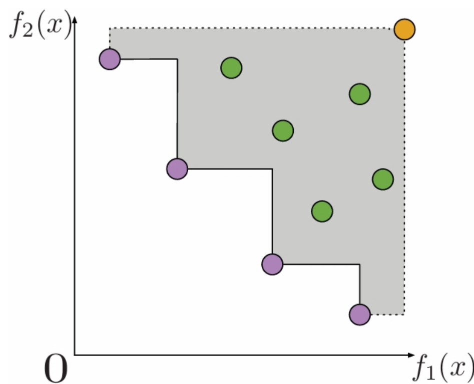
where $[\cdot]$ denotes the indicator function, returning 1 if its argument is true and 0 otherwise, and m is the number of objectives (in this case, $m = 2$, corresponding to *energy* and *latency*). Intuitively, this integral sums the volume of all regions in the objective space that are dominated by at least one Pareto-optimal solution and bounded by the reference point.
In this work, the hypervolume computation is carried out using the corresponding **hv** module of the `Pymoo` library [3]. The resulting hypervolume value is treated as a standalone scalar metric, representing the area/volume dominated by the provided set of solutions with respect to a reference point, consistent with the definition used by the library itself.

A graphical interpretation of the hypervolume concept is shown in Figure 5.1a. In the two-dimensional case, where both objectives f_1 and f_2 are minimized, points closer to the origin represent better trade-offs. The shaded gray region corresponds to the hypervolume dominated by the current Pareto front. When a new non-dominated solution is discovered, the hypervolume increases, indicating an improvement in the quality of the front. In Figure 5.1b, a more general representation of hypervolume metrics for a 3D objective space [18] is shown.

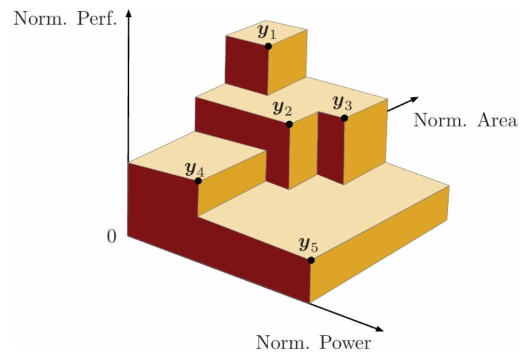
In the context of this work, the objectives correspond to two key performance metrics of IMC-based accelerators: **energy consumption** and **latency**. A better architecture achieves lower energy usage and shorter execution time. Hence, the reference point v_{ref} is defined as a pessimistic bound representing the worst possible values for both metrics, ensuring that all feasible solutions dominate it. It is obtained by evaluating a fundamental architecture consisting of only a single buffer and compute units using the FactorFlow framework. This will lead to poor mapping, providing us with the perfect worst-case scenario reference point.

During the experimental evaluation, the hypervolume is computed at each generation of the evolutionary algorithm. This allows monitoring of the convergence behavior of the NSGA-II process, as an increasing hypervolume value over time reflects continuous improvement in the quality and diversity of the explored Pareto front. Similarly, this metric is also used as a key signal for the *mutation adaptation* mechanism discussed in Section 4.4, where changes in hypervolume determine whether exploration or exploitation should be favored in subsequent iterations.

The primary objective of this analysis is to demonstrate that increasing the number of generations does not necessarily lead to a proportional improvement in exploration quality. In fact, beyond a certain point, the hypervolume tends to stabilize, indicating that the algorithm has already reached a near-optimal set of solutions and further evolution yields diminishing returns.



(a) 2D representation of the Pareto hypervolume.



(b) 3D generalization of the hypervolume concept.

Figure 5.1: Pareto hypervolume: (a) 2D representation and (b) 3D generalization.

Average Distance from Reference Set (ADRS)

Another key metric for evaluating Design Space Exploration (DSE) algorithms and comparing Pareto fronts is the **Average Distance from Reference Set (ADRS)**. This metric provides a quantitative measure of how closely the Pareto front obtained by an optimization algorithm approximates a known reference front, such as the global Pareto-optimal solution.

The ultimate goal of any DSE algorithm is to efficiently approximate the global Pareto front—typically obtained through a brute-force search—while significantly reducing computational cost and runtime. Whereas the brute-force approach guarantees the exact Pareto-optimal set for a given set of objectives and constraints, heuristic or evolutionary methods, such as the one proposed in this work, aim to produce a close approximation of this ideal front. The ADRS metric quantifies the deviation between these approximate solutions and the actual Pareto front.

Formally, let the reference Pareto front be denoted as $\Pi = \Psi(\Phi)$ and the approximate Pareto front as $\Lambda = \Psi(\Omega)$. The ADRS is then defined as follows [21]:

$$ADRS(\Pi, \Lambda) = \frac{1}{|\Pi|} \sum_{x_R \in \Pi} \left(\min_{x_A \in \Lambda} \{\delta(x_R, x_A)\} \right)$$

where $\delta(x_R, x_A)$ represents the normalized distance in the objective function space between two configurations x_R and x_A. A smaller ADRS value indicates that the approximate Pareto front is closer to the reference front, thus reflecting a higher accuracy of the exploration algorithm.

In this work, two ADRS evaluations were performed:

- The first computed the ADRS with respect to a *combined reference front*, obtained by merging all the explored points from the various algorithms and computing the Pareto front over this unified set.
- The second computed the ADRS with respect to the *full-search* Pareto front, representing the true global optimum.

The first analysis offers a comparative insight into the relative performance and dominance relationships among the different exploration strategies. In contrast, the second provides a measure of absolute accuracy with respect to the global optimum.

Spacing

In the context of Design Space Exploration (DSE), the quality of a Pareto front is not determined solely by its proximity to the optimal frontier but also by the *distribution* and *uniformity* of the solutions it contains.

The following metric was implemented to evaluate the distributional quality of the Pareto front [34].

The *spacing metric* assesses the uniformity of the non-dominated solutions along the Pareto front by measuring the variation in distance between neighboring solutions. Formally, for a set of non-dominated solutions S with cardinality $|S|$ and H objectives, the minimum Manhattan distance d_i between each solution s_i and its nearest neighbor is computed, as showed in Figure5.2, as:

$$d_i = \min_{s_k \in S,\, s_k \neq s_i} \sum_{h=1}^{H} \left| f_h(s_i) - f_h(s_k) \right|$$

where $f_h(s_i)$ denotes the value of the h-th objective for solution s_i. The standard deviation of these distances then gives the overall spacing metric SP:

$$SP = \sqrt{\frac{1}{|S|-1} \sum_{i=1}^{|S|} (d_i - \bar{d})^2}$$

where $\bar{d}$ is the mean of all d_i values. A smaller spacing value indicates a more uniformly distributed set of solutions along the Pareto front, which is generally desirable as it implies that the trade-off region is well-sampled.

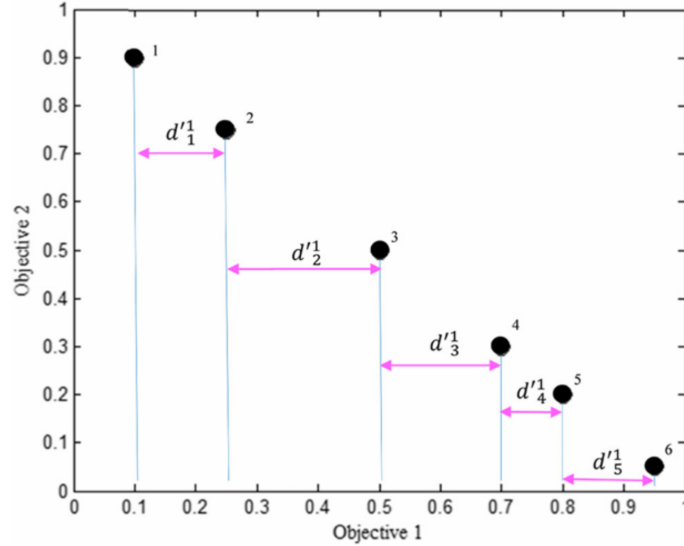


Figure 5.2: Computation of the minimum Manhattan distances d_i.

The goal of this analysis is to demonstrate how the spacing metric reflects the uniformity of the distribution of Pareto-optimal solutions, independently of the overall length or shape of the front. In fact, spacing depends solely on the relative distances between the individual solutions, something that cannot be easily appreciated from a simple visual inspection of the plots.

5.1.3. Parameter Configurations

As introduced in Section 4.5, the proposed methodology's implementation has been designed to be fully parametric and reusable. To ensure flexibility and reproducibility, all key parameters governing the algorithm's behavior are encapsulated within configurable variables. These can be modified either statically through the `Settings` class or dynamically via user-defined command-line inputs.

This section provides a concise overview of the configuration parameters employed in the experimental phase. Such parameters are essential not only for understanding the baseline operation of the algorithm but also for assessing the impact and effectiveness of the optional optimization strategies.

For clarity, the parameters are categorized into two main groups:

- **Core setup parameters** — fundamental to the base NSGA-II algorithm, including the population size, number of offspring per generation, and total number of generations;

- **Optimization parameters** — controlling the activation or deactivation of optional optimization strategies, together with their associated configuration values when enabled.

A comprehensive summary of these parameters is reported in Table 5.1 and Table 5.2.

Parameter	Type	Description
INITIAL_POPULATION	Integer	Number of samples evaluated for the initial population.
OFFSPRINGS_SIZE	Integer	Number of offspring solutions generated at each iteration from the selected parents.
GENERATIONS_SIZE	Integer	Total number of generations over which the algorithm evolves the population.
MUTATION_POWER_RATIO	Float [0,1]	Ratio based on which the mutation power is computed.

Table 5.1: Fundamental configuration parameters of the NSGA-II algorithm.

Parameter	Type	Description
ENABLE_RETRAINING	Boolean	Activation of retraining optimization.
RETRAIN_INTERVAL	Integer	Interval of generations.
RETRAIN_SAMPLE_SIZE	Integer	Number of samples to retrain with.
ENABLE_ADAPTIVE_MUTATION	Boolean	Activation of adaptive mutation optimization.
ANNEALING	Boolean	Activation of annealing optimization.
ANNEALING_TEMPERATURE	Float [0,∞]	Temperature of annealing.
COOLING_RATE	Float [0,∞]	Cooling rate for annealing.

Table 5.2: Optimizations configuration parameters.

Note that some variables never change during the following experiments.

5.2. Results

This section presents the experimental results obtained from the proposed methodology. Several Pareto fronts corresponding to different algorithmic configurations are illustrated and compared with those produced by alternative pseudo-random exploration strategies. Finally, the behavior of the algorithm is analyzed with respect to the evaluation metrics previously introduced in Section 5.1.2.

To ensure a coherent and fair comparison among experiments, all results were generated starting from the same initial sample set of 250 configurations. Each configuration was defined under the following design constraints:

- **Memory capacity (`memory_kb`)**: between 131,072 KB and 2,097,152 KB (128 MB – 2048 MB);
- **Number of memories**: from 2 to 6;
- **CIM compute levels**: all available values (4×4, 8×8, 16×16, 24×24, 32×32);
- **Fanout levels**: between 2 and 4;
- **Number of registers**: between 2 and 4.

All experiments were conducted under the same conditions, with only the optimization strategies and algorithmic parameters varying to evaluate their impact on the exploration process and the quality of the resulting Pareto fronts.

5.2.1. Basic Homogeneous Exploration

This section presents the first experimental configuration, which serves as the baseline for evaluating the overall behavior of the exploration framework. The configuration is characterized by a **homogeneous division** between offspring and survivor elements, adopting an equal 1:1 ratio. This balanced setup ensures that, at each generation, an equivalent number of individuals are replaced and preserved, maintaining equilibrium between the exploration of new candidate solutions and the exploitation of previously identified high-quality configurations.

With this setup fixed, the following analysis explores the resulting behavior of several key evaluation metrics while keeping the **mutation adaptation** mechanism active.

Figure 5.3 reports the comparison between the Pareto fronts obtained from the different

variants of the **NSGA-II** algorithm implemented in this work, together with the one derived from the **Simulated Annealing (SA)** algorithm. In this specific experiment, the mutation adaptation feature was enabled, retraining was disabled, the offspring population size was set to 125, and the total number of generations was fixed to 40. As illustrated in the figure, there is a clear and consistent domination gap in favor of the **NSGA-II** configuration in which both the *Decay* and *Simulated Annealing* optimizations are activated. The plot also highlights the strong uniformity and good overall distribution of the explored points across the design space, indicating that the algorithm effectively balances the discovery of new solutions and the refinement of existing ones. Interestingly, the standalone *Simulated Annealing* approach, despite its inherently stochastic nature, still manages to produce competitive results, as a few of its points dominate portions of the other fronts. This confirms that SA, although not deterministic, can still identify valuable regions of the search space and make meaningful contributions to the overall exploration process.

To complement the Pareto analysis, Figure 5.4 illustrates the corresponding **hypervolume trend**. A clear pattern emerges: the hypervolume increases rapidly during the first generations, when the algorithm discovers a large number of new, non-dominated solutions, and then gradually flattens as the population stabilizes and the evolutionary process approaches convergence. The growth rates among the three NSGA-II variants are closely comparable; however, the configuration that combines both optimizations consistently exhibits the steepest and most stable improvement. This behavior aligns perfectly with the Pareto front observations, reinforcing the conclusion that the combined-optimization NSGA-II variant offers the best trade-off between convergence speed and exploration depth. The coherence between the Pareto and hypervolume analyses validates the reliability of the obtained results.

The **ADRS** analysis, presented in Figure 5.5, further confirms the trends observed in the previous plots. In this metric, the NSGA-II configuration with both optimizations active is identified as the *reference Pareto front*, exhibiting a null ADRS value, indicating that it dominates all other configurations by a significant margin. As previously noted, the second-best performance is achieved by the *Simulated Annealing* approach, indicating that—while less efficient in convergence—it still provides a reasonably competitive approximation of the optimal Pareto front. This finding reinforces the observation that stochastic methods, despite their less guided nature, can still generate valuable insights and good-quality solutions within a limited number of generations.

Finally, Figure 5.6 focuses on the **spacing metric**, revealing aspects of the solution distribution that are not evident from the Pareto plots alone. Interestingly, while the optimized NSGA-II variant dominates both the basic NSGA-II and the Simulated Annealing strategies, they tend to exhibit slightly *lower spacing values*, suggesting a denser clustering of solutions along the front. This implies a somewhat richer local diversity in the final population. Although this observation is not decisive in determining the overall superiority of one method over another—since dominance and hypervolume remain the most critical indicators—it nonetheless provides a valuable perspective on the internal dynamics of the exploration process.

Overall, this initial homogeneous configuration demonstrates the validity and robustness of the proposed evolutionary framework. It confirms the consistency between different performance metrics and highlights the benefits of combining the Decay and SA-inspired optimization mechanisms in improving both convergence quality and exploration uniformity.

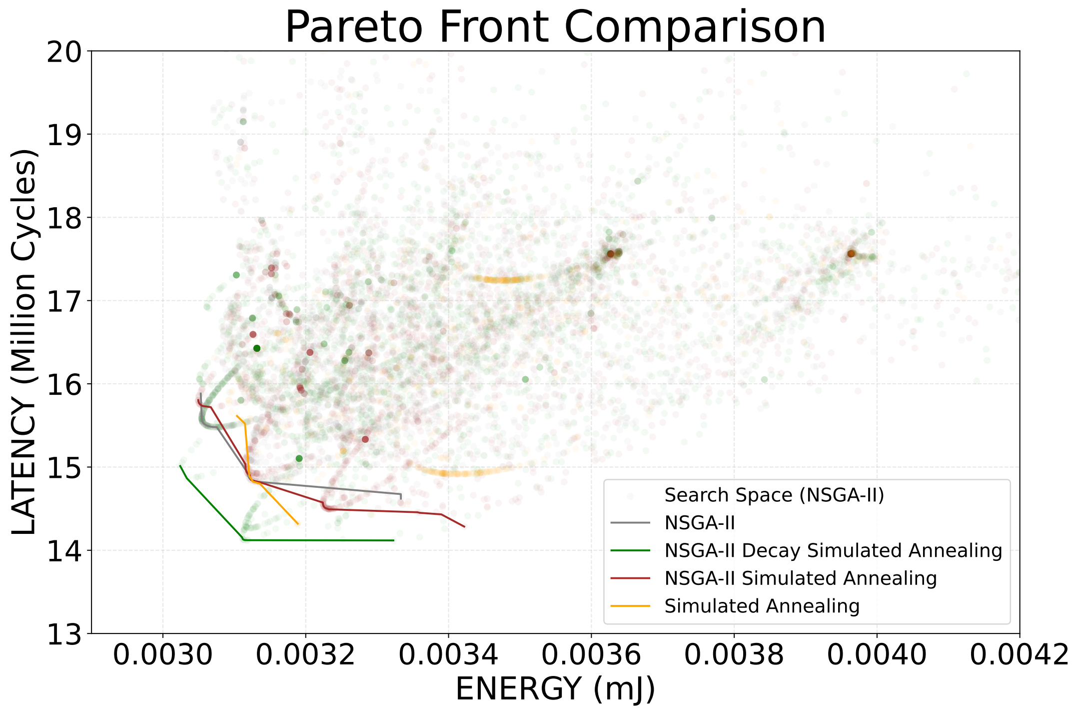


Figure 5.3: Pareto analysis.

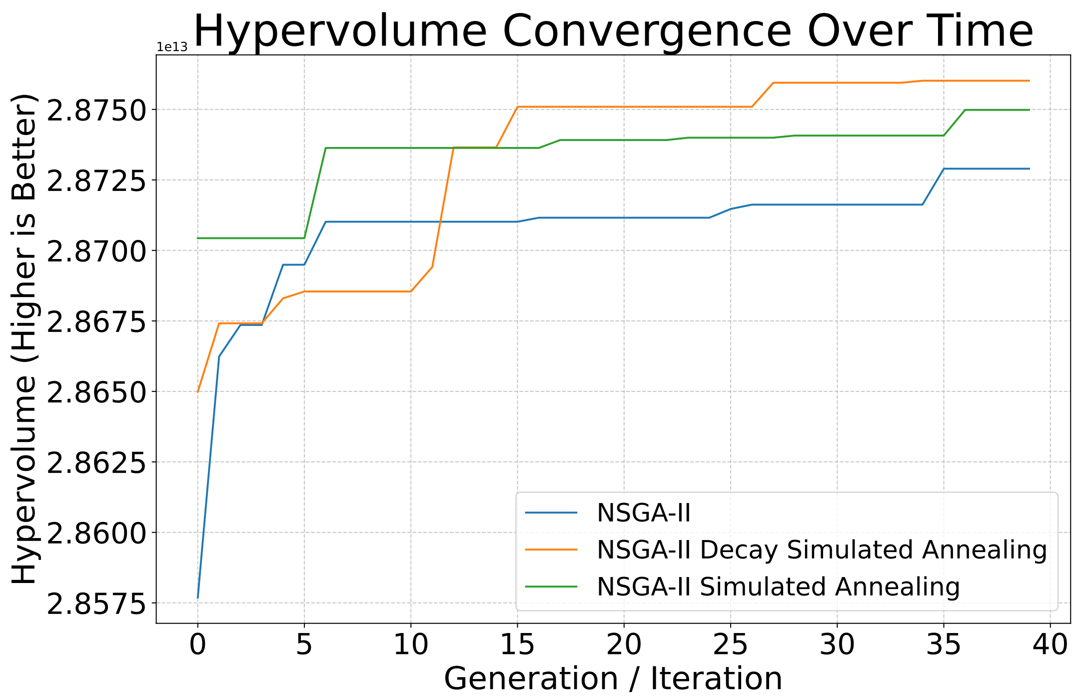


Figure 5.4: Hypervolume behavior.

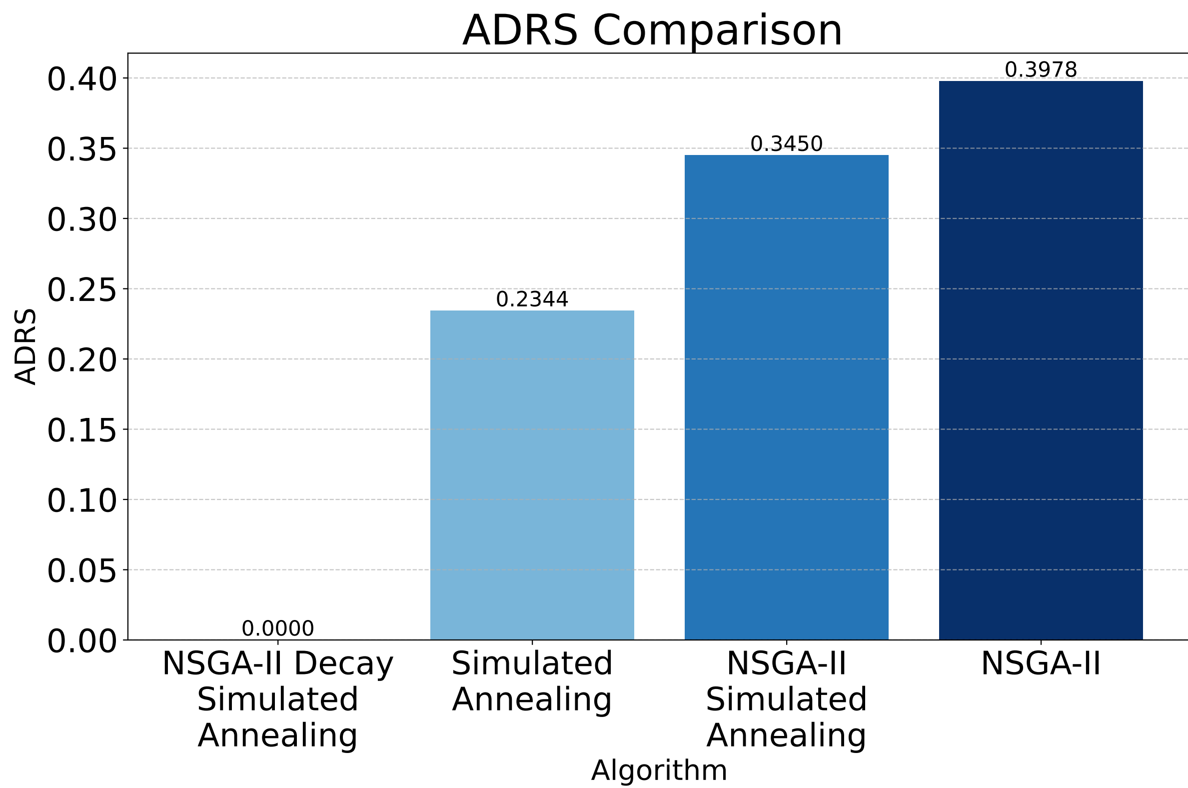


Figure 5.5: ADRS comparison.

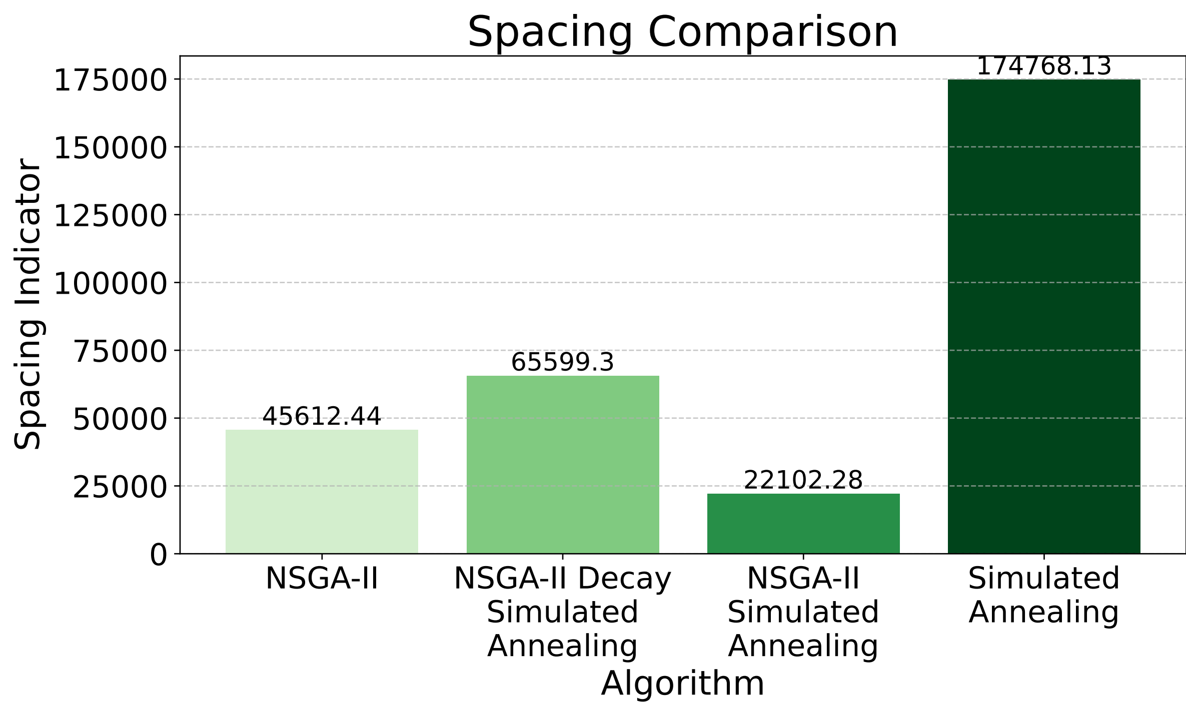


Figure 5.6: Spacing comparison.

Basic Homogeneous without Adaptation

This part of the experimental campaign focuses on analyzing the same configuration described in Section 5.2.1, but with *mutation adaptation* optimization disabled. The primary objective of this experiment is to evaluate the effect of removing this mechanism on the algorithm's ability to maintain a high level of solution diversity and on the overall quality of the resulting Pareto fronts. The mutation adaptation mechanism plays a crucial role in dynamically regulating the variability of the generated solutions across generations. By progressively adjusting the mutation rate based on evolutionary progress, the algorithm maintains a broader coverage of the design space in the early stages while encouraging a finer concentration of solutions as convergence is approached.

In Figure 5.7, the distribution of the obtained solutions appears significantly less uniform compared to the experiment where adaptation was enabled. This lack of variability causes the population to converge prematurely, producing clusters of similar solutions rather than a diverse and well-distributed set. The effect is evident for the baseline NSGA-II configuration: without any adaptive mechanism, the search tends to remain confined to limited regions of the design space, reducing the overall variety of the resulting solutions. Conversely, optimization strategies that incorporate stochastic behavior—such as the Decay and Simulated Annealing variants—partially mitigate this issue by preserving a broader solution spread, even in the absence of explicit adaptation.

Figure 5.8 further illustrates this behavior by analyzing the hypervolume trend. For the baseline NSGA-II configuration, which lacks any mechanism for dynamically modulating the mutation strength, convergence occurs much more slowly. In fact, the final hypervolume achieved by this configuration does not reach the level of the optimized variants, clearly indicating a reduced capacity to generate diverse and practical solutions. The other configurations, benefiting from their inherent stochastic components, display faster and steadier improvements in the metric.

The **ADRS** analysis in Figure 5.9 supports these observations. When the mutation adaptation mechanism is removed, the diversity among the explored solutions decreases noticeably, leading to greater overlap among the Pareto fronts produced by different algorithmic variants. In this case, no configuration perfectly represents the reference front. The closest is the NSGA-II version, which combines both Decay and Simulated Annealing optimizations, although a small ADRS distance from the constructed reference set is still observed. Moreover, the results show a pronounced overlap between the basic NSGA-II, the NSGA-II with both optimizations, and the NSGA-II with only the SA optimization.

This overlap was less evident in the previous configuration with adaptation enabled. Figure 5.10 further highlights the consequences of reduced diversity in the final population. In particular, the baseline NSGA-II algorithm tends to rely heavily on its initial individuals and their close neighbors, leading to a limited sampling of the design space. The slightly improved spacing values observed in this configuration are, in fact, misleading: they primarily result from the initial Latin Hypercube Sampling, which ensures a well-distributed starting population, rather than from the algorithm's own ability to maintain diversity during evolution.

Overall, the experimental evidence confirms that disabling the mutation adaptation mechanism results in a less effective search process, characterized by limited variety in the generated solutions, reduced coverage of the design space, and less uniform Pareto fronts. Configurations without mutation adaptation are excluded from further analysis, as their absence consistently degrades convergence and solution diversity.

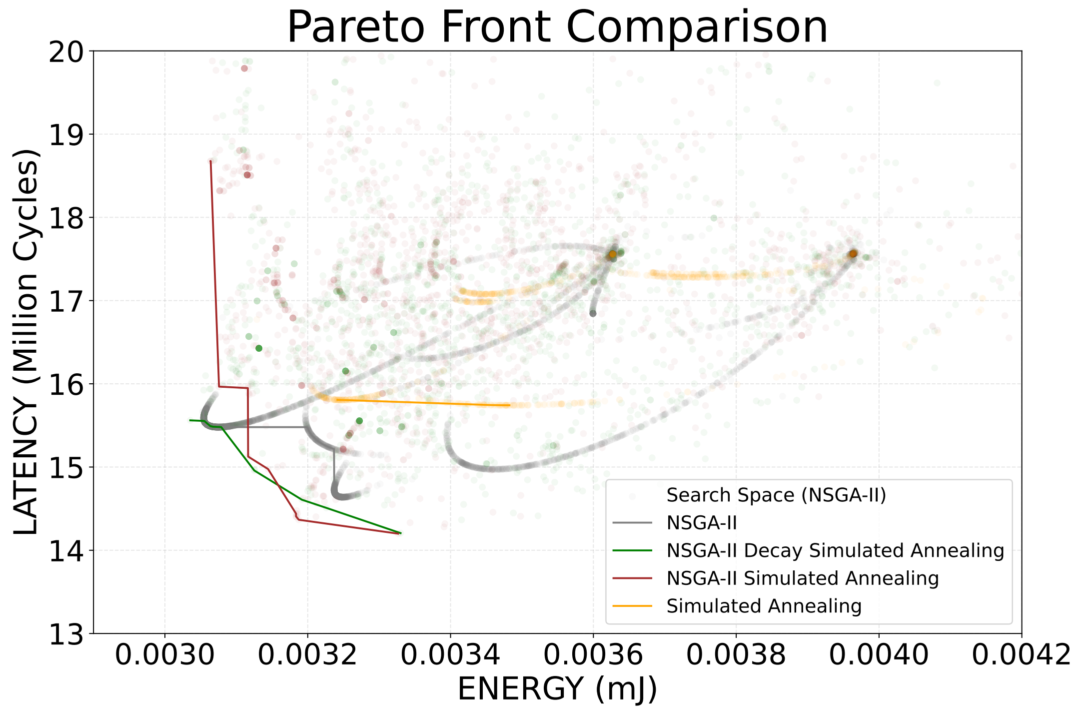


Figure 5.7: Pareto analysis.

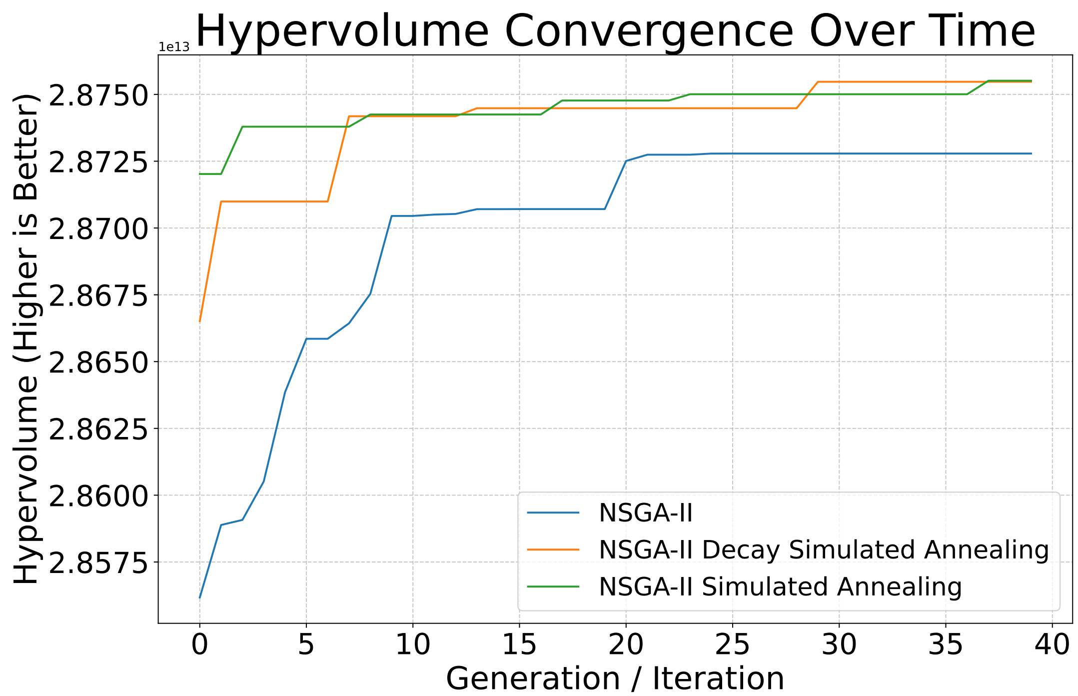


Figure 5.8: Hypervolume behavior.

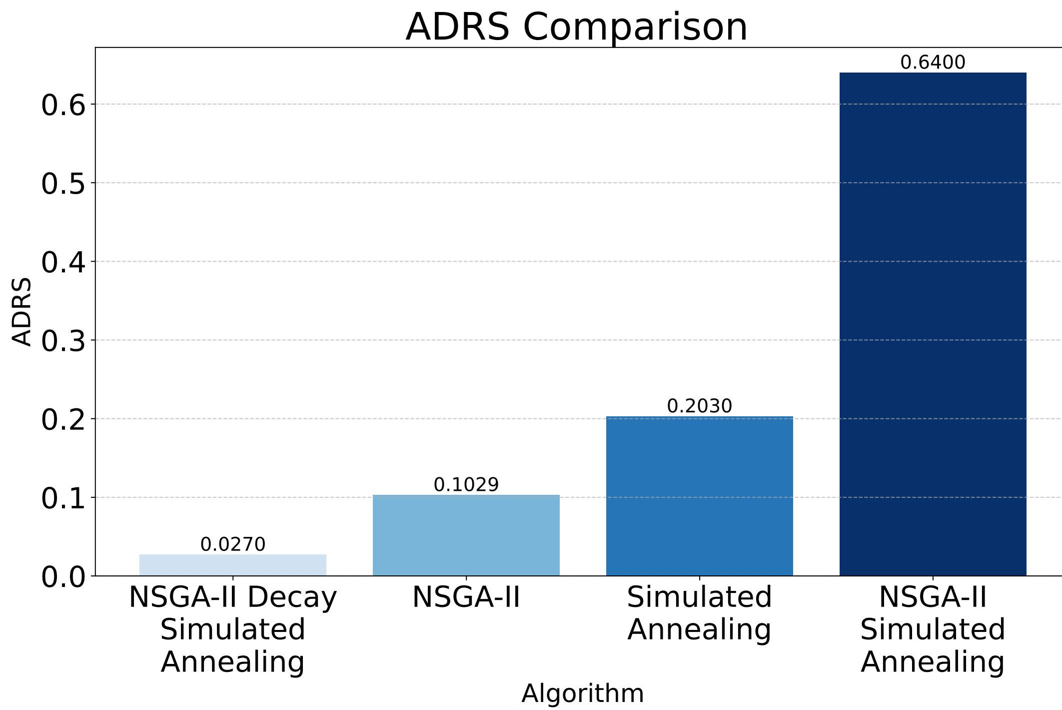


Figure 5.9: ADRS comparison.

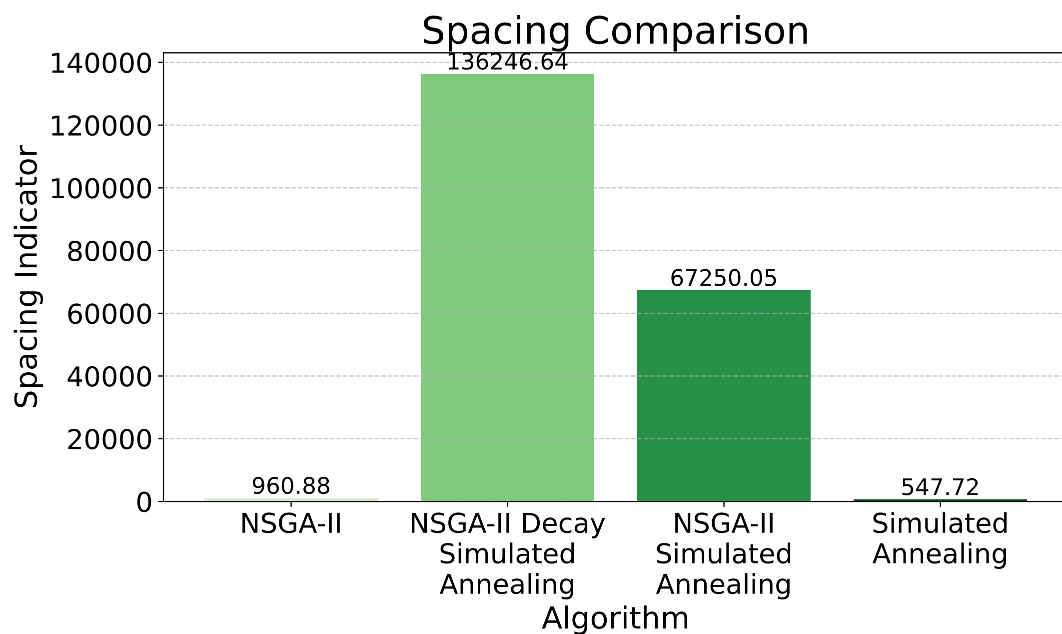


Figure 5.10: Spacing comparison.

5.2.2. Exploitation vs Exploration

This section investigates how varying the ratio between offspring and survivor elements influences the algorithm's tendency to favor *exploitation* over *exploration*. Specifically, the following experiments modify the number of offspring generated at each generation, intentionally exaggerating both extremes—producing either too few or too many offspring—to create a non-homogeneous balance with respect to the number of survivors. The objective is to analyze how this imbalance affects the number of explored solutions and the overall behavior of the main performance metrics, in comparison with the homogeneous configuration discussed in Section 5.2.1.

Figure 5.11 shows the results for a configuration with 50 offspring, 40 generations, mutation adaptation enabled, and no retraining. This experiment highlights the impact of offspring population size on the balance between exploration and exploitation. As shown, reducing the number of offspring results in a sparser coverage of the design space, confirming that offspring size plays a crucial role in ensuring proper exploration and diversity of solutions. Conversely, Figure 5.12 illustrates the opposite situation: increasing the number of offspring to over 200 yields a much denser distribution of evaluated solutions across the search space. Interestingly, in this configuration, the Simulated Annealing strategy becomes strongly dominated by the other algorithms, with the NSGA-II equipped with both optimizations maintaining its dominance, although with a smaller margin than in the homogeneous setup. This suggests that generating more offspring facilitates deeper exploration, potentially leading to improved outcomes.

The hypervolume behavior, reported in Figure 5.13, confirms these trends. With only 50 offspring per generation, the algorithm struggles to explore new regions of the design space because most of the existing population is preserved from one generation to the next. As a result, the hypervolume stagnates very early, reflecting the limited introduction of new candidate solutions. Conversely, Figure 5.14 shows that with 200 offspring per generation, over the same total number of generations, the algorithm explores more aggressively, generating many new candidates and partially "overwriting" the existing population. While this increases the chance of discovering new regions, it also introduces the risk of neglecting some of the best solutions from previous generations, leading to a different form of early stagnation of the hypervolume.

Figure 5.15a presents the ADRS results for the 50-offspring configuration. Here, no single algorithm emerges as the absolute reference set, highlighting the overlap of optimal solutions among the different strategies. In contrast, Figure 5.15b—corresponding to the 200-offspring case—shows that the NSGA-II algorithm with both optimizations activated

becomes the near-reference front, with only minor overlapping regions shared with other configurations. Although this overlap is not clearly visible in the Pareto dominance plots, it reinforces the conclusion that NSGA-II with combined Decay and Simulated Annealing remains the most effective configuration overall.

Finally, Figures 5.16a and 5.16b analyze the effect of offspring population size on the spacing metric. Comparing the results with 50 and 200 offspring, it is evident that larger offspring populations—particularly when coupled with mutation adaptation—produce consistently lower spacing values, confirming that broader exploration promotes a more uniform and well-distributed Pareto front. Simulated Annealing, on the other hand, remains largely unaffected by this parameter, as its implementation does not depend on the concept of offspring population.

The experiments in this section highlight the need to strike a balance between exploration and exploitation in evolutionary optimization. Increasing the number of offspring expands design-space coverage and maintains diversity, but excessive exploration can result in the loss of high-quality solutions that are already present in the population. Too few offspring, instead, make the search overly exploitative, refining existing solutions while missing unexplored regions and producing a narrower, less optimal Pareto front.

The homogeneous configuration offers the best compromise, preserving early diversity while steadily improving solution quality. Overall, the offspring-to-survivor ratio proves to be a key parameter that shapes the algorithm's behavior. This effect is amplified by the *elitist* nature of NSGA-II, which preserves high-quality individuals across generations, allowing even some early samples to persist and eventually become optimal.

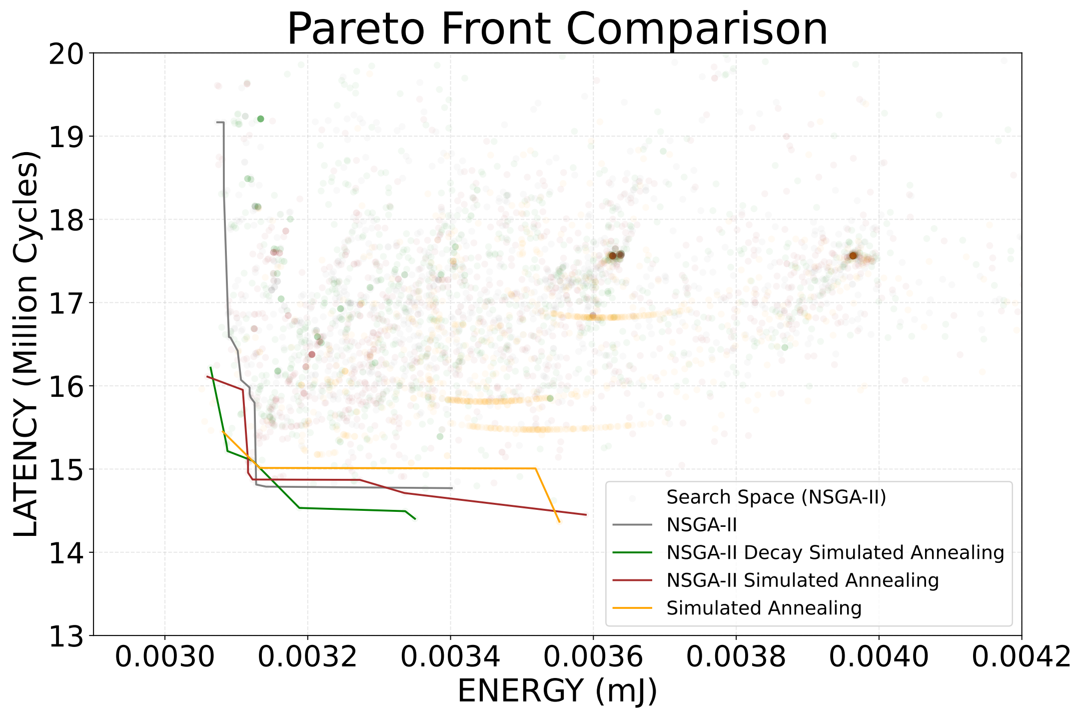


Figure 5.11: Pareto analysis 50 Offsprings.

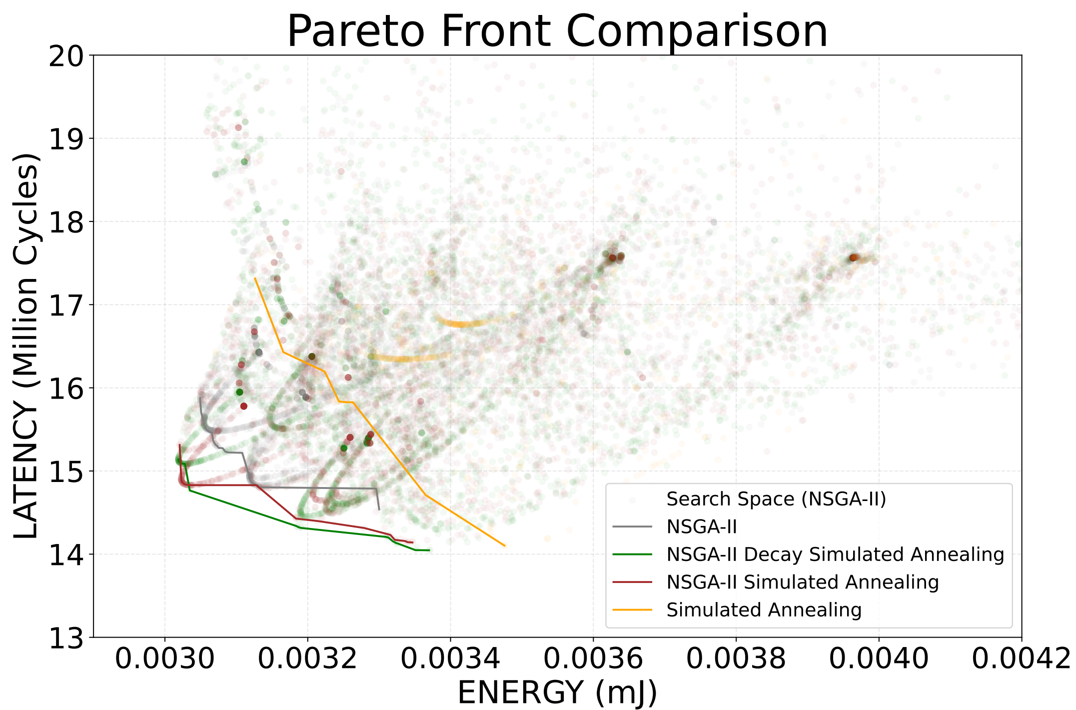


Figure 5.12: Pareto analysis 200 offsprings.

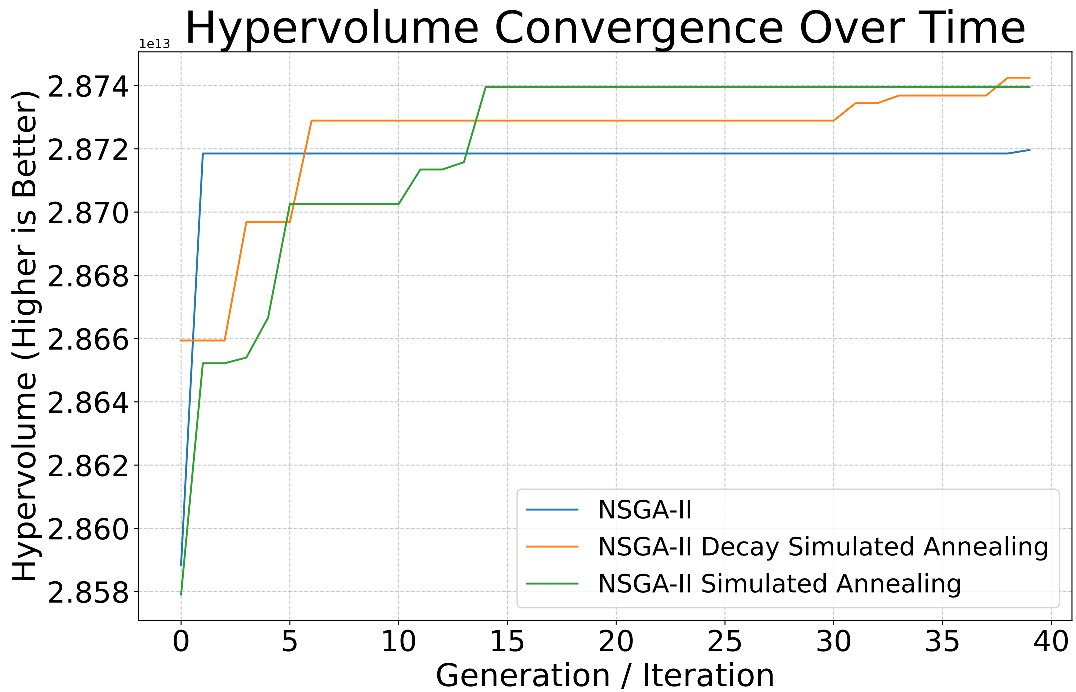


Figure 5.13: Hypervolume behavior 50 offsprings.

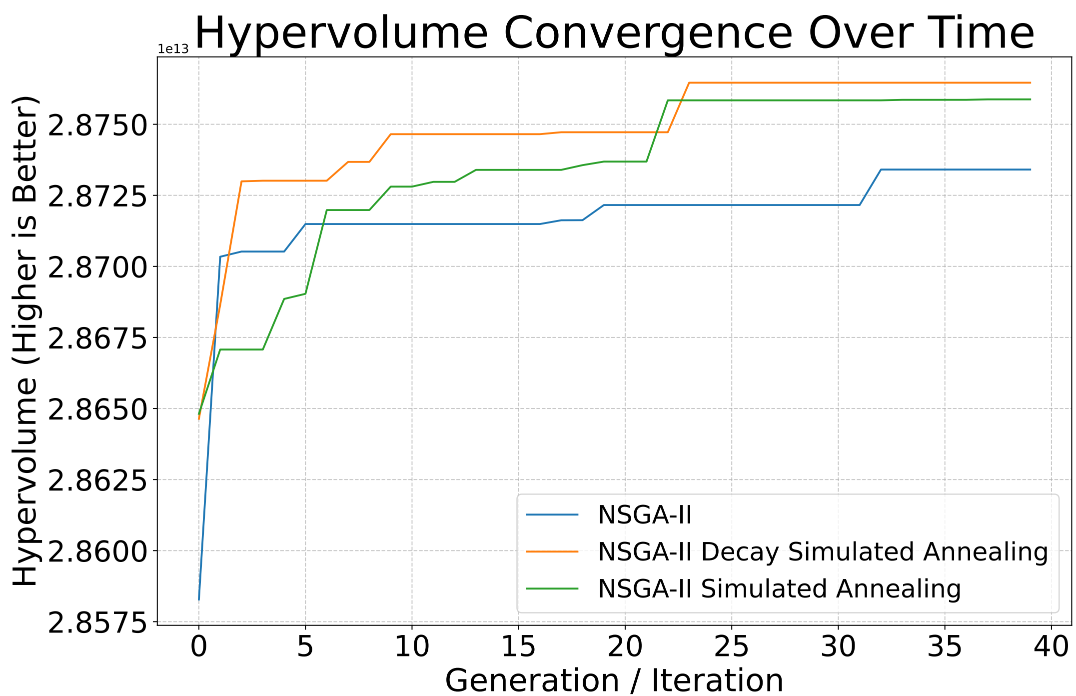


Figure 5.14: Hypervolume behavior 200 offsprings.

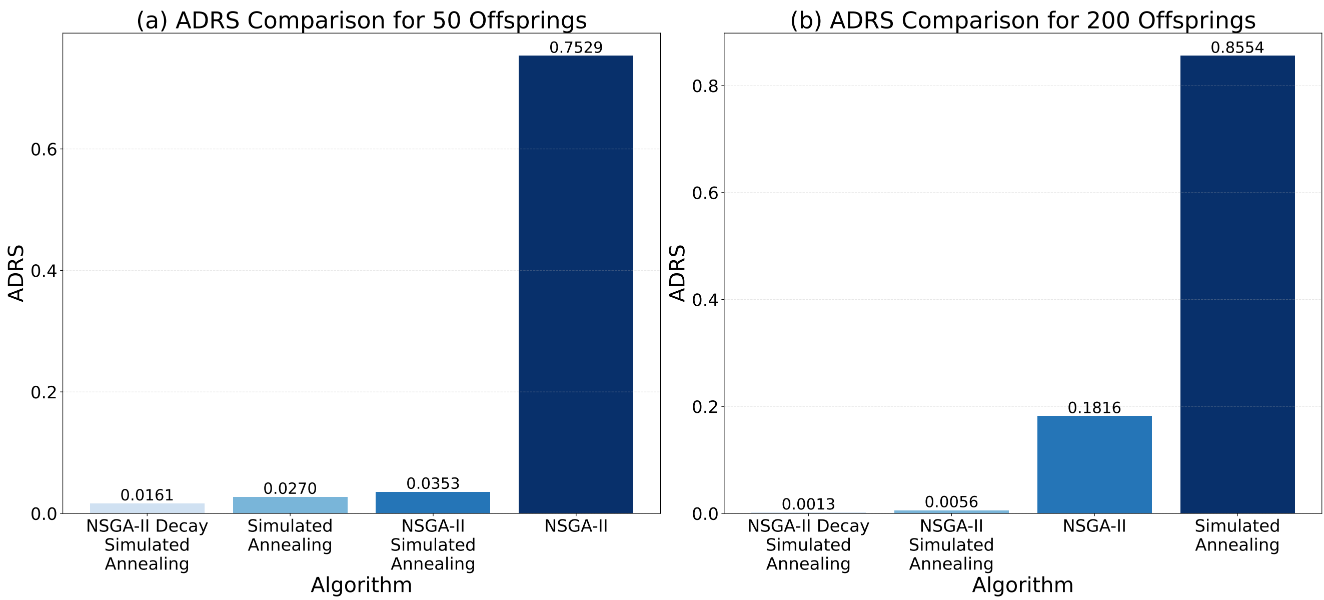


Figure 5.15: ADRS Comparison.

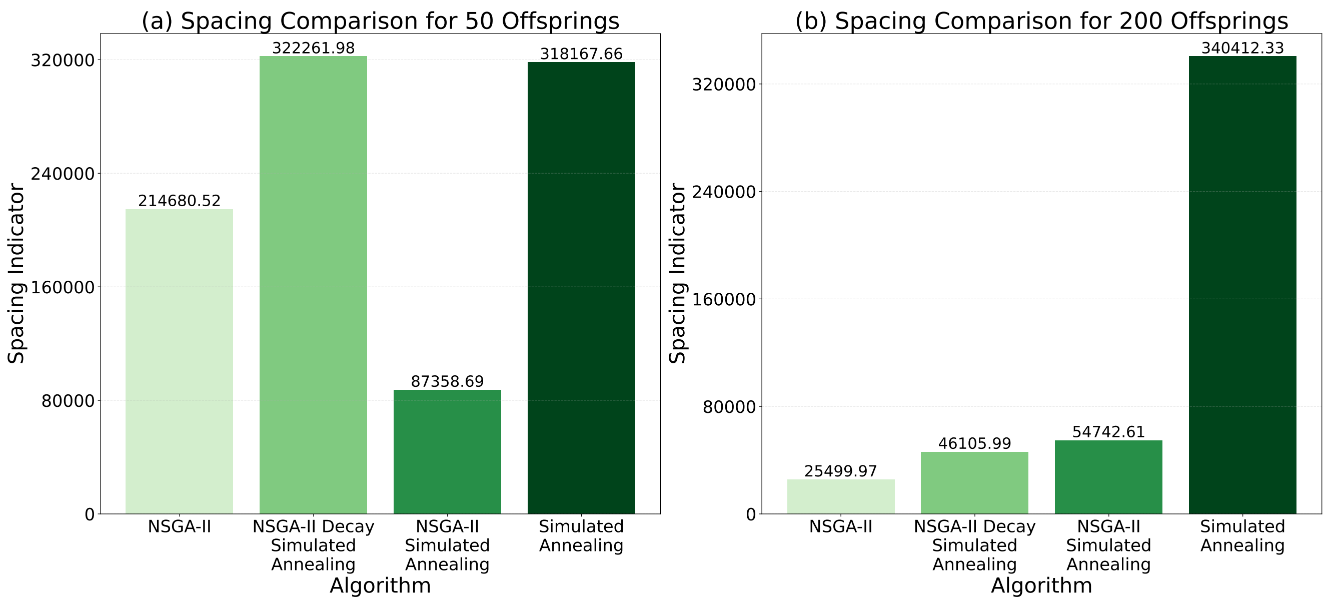


Figure 5.16: Spacing comparison.

Full Search Comparison

To conclude this part of the analysis, a comprehensive set of experiments was performed to quantitatively assess the effectiveness of the proposed algorithms by comparing their

results against a *full search* of the design space, which serves as a reference baseline representing the actual Pareto-optimal front. This exhaustive exploration provides the ground truth for evaluating the quality, accuracy, and convergence properties of the approximated Pareto fronts obtained through the implemented evolutionary methodologies.

As illustrated in Figure 5.17a, and starting from the baseline configuration characterized by a homogeneous offspring distribution and mutation adaptation enabled, the NSGA-II algorithm, with both *decay* and *simulated annealing* optimizations, consistently achieved the smallest average distance from the global Pareto front. This result confirms the algorithm's robustness in maintaining a delicate balance between exploring new regions of the design space and exploiting previously identified high-quality solutions. The improved convergence behavior and reduced ADRS distance demonstrate that the combination of decay-based mutation control and stochastic refinement through simulated annealing effectively enhances the search efficiency while preserving diversity among solutions.

The subsequent configurations, presented in Figures 5.17b–5.17b, where the offspring distribution and mutation adaptation mechanisms were intentionally varied, further corroborate this trend. Even when the number of offspring or the adaptation dynamics were altered, the homogeneous configuration continued to provide the closest approximation to the global Pareto front. This consistency highlights the stability and generalization capability of the proposed optimization approach across multiple parameter settings.

In particular, for relatively compact and well-constrained design spaces, such as the one explored in this thesis, maintaining a balanced offspring-to-survivor ratio proved fundamental for achieving reliable convergence. A homogeneous distribution ensures that each generation contributes meaningfully to the refinement of the population, enabling the algorithm to progressively enhance the quality of the elite set while avoiding premature stagnation. As a result, the obtained Pareto fronts exhibit a well-structured and densely populated distribution of solutions, reflecting an efficient and controlled convergence process that remains close to the global optimum identified by the full search.

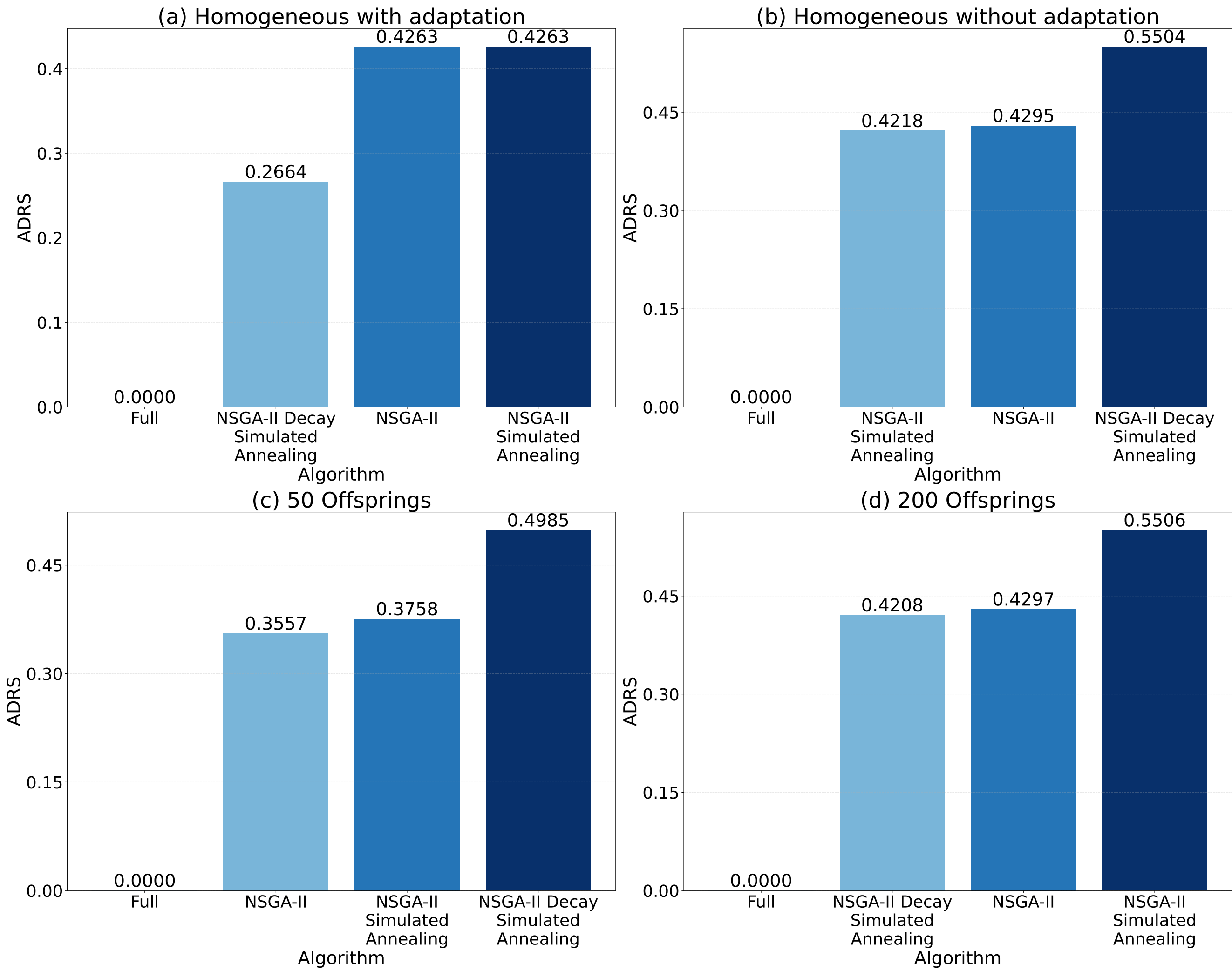


Figure 5.17: ADRS distance comparisons across various configurations, using the full search Pareto front as reference (32 samples, no retraining).

5.2.3. Re-Training Optimization

A dedicated analysis is presented in this section for the **retraining optimization**, one of the most critical mechanisms introduced in this thesis to enhance the predictive reliability of surrogate models during the evolutionary process. This optimization was conceived to address one of the fundamental challenges arising from the use of Gaussian Process (GP) models as surrogate predictors—namely, the degradation of prediction accuracy when evaluating candidate solutions that deviate significantly from the original training distribution.

The core idea behind the use of surrogate models is to reduce the computational burden of design-space exploration by replacing expensive complete evaluations with fast

approximations. However, the quality of these approximations is highly dependent on the representativeness of the training set used to build the surrogate. Even when a statistically robust sampling strategy such as Latin Hypercube Sampling is employed, the initial dataset can never fully cover the entire design space—especially in high-dimensional, nonlinear, and multi-objective optimization problems. As the evolutionary process proceeds, newly generated offspring may occupy regions of the design space that were underrepresented or completely absent in the initial training phase. When this occurs, the GP model's predictions become less reliable, producing higher uncertainty estimates.

Figure 5.18 illustrates the maximum σ_E values observed for *energy* predictions under the two considered configurations—with and without the retraining optimization. A similar behavior was also observed for the *latency* predictions, confirming that the activation of the retraining mechanism consistently reduces the uncertainty of the surrogate model across both performance metrics.

Each configuration involves 125 offspring and 40 generations, providing a representative view of the model behavior over a standard evolutionary run. When retraining is disabled, the GP surrogate exhibits considerably higher uncertainty levels across both metrics. This behavior confirms that, as the population explores less familiar regions of the design space, the GP model is forced to extrapolate beyond the boundaries of its original training data, leading to less reliable estimates. Such elevated uncertainty values can significantly affect the quality of the exploration process, increasing the likelihood of generating suboptimal or physically inconsistent solutions.

Conversely, when the retraining optimization is activated, a clear improvement in model confidence is observed. In this setup, retraining is performed once during the 40-generation run—specifically every 20 generations—by adding four new fully evaluated samples to the training dataset. These new data points, selected from both high-performing and poorly performing individuals, enrich the GP's knowledge of the design space and enable a more accurate regression of the underlying energy–latency trade-off surface. As shown in the figures, this mechanism results in a substantial reduction in the maximum σ_E values for energy, proving that even a limited number of retraining steps can significantly enhance the stability and predictive accuracy of the surrogate model. A similar consideration applies to the latency parameter σ_L, which is not reported here for brevity.

It is important to note, however, that this improvement comes at a non-negligible computational cost. As reported in Figure 5.19, configurations with retraining enabled exhibit substantially longer execution times compared to their non-retrained counterparts. This increase is due to the overhead of performing complete evaluations for the selected samples

and subsequently updating the GP model parameters, both of which are time-consuming operations relative to standard surrogate evaluations. Although this trade-off may appear disadvantageous from a performance standpoint, it is crucial to consider the quality gains achieved in predictive fidelity. The enhanced accuracy not only prevents the propagation of incorrect estimations but also guides the exploration more effectively toward promising regions of the design space, ultimately improving convergence to the actual Pareto front.

From a methodological perspective, the retraining optimization thus represents a mechanism for maintaining *model adaptivity* throughout the evolutionary process. While the surrogate initially provides a reliable approximation of the design landscape, its relevance may degrade as the exploration diverges from the initial sample set. Retraining restores the model's descriptive power by continuously realigning it with the evolving distribution of candidate solutions. This dynamic interaction between exploration and model refinement embodies a form of *co-evolution*, where both the population of designs and the predictive model evolve together toward optimality.

Ultimately, the decision to activate this optimization depends on the intended balance between computational efficiency and predictive robustness. For exploratory studies where speed is paramount, the surrogate-only approach without retraining may suffice, accepting a degree of uncertainty. Conversely, for high-precision optimization runs aimed at generating physically consistent and verifiable designs, the retraining mechanism provides a substantial advantage by ensuring a more accurate and trustworthy evaluation process. In this sense, the retraining optimization can be seen as a strategic compromise—sacrificing some runtime in exchange for a significant improvement in quality and reliability.

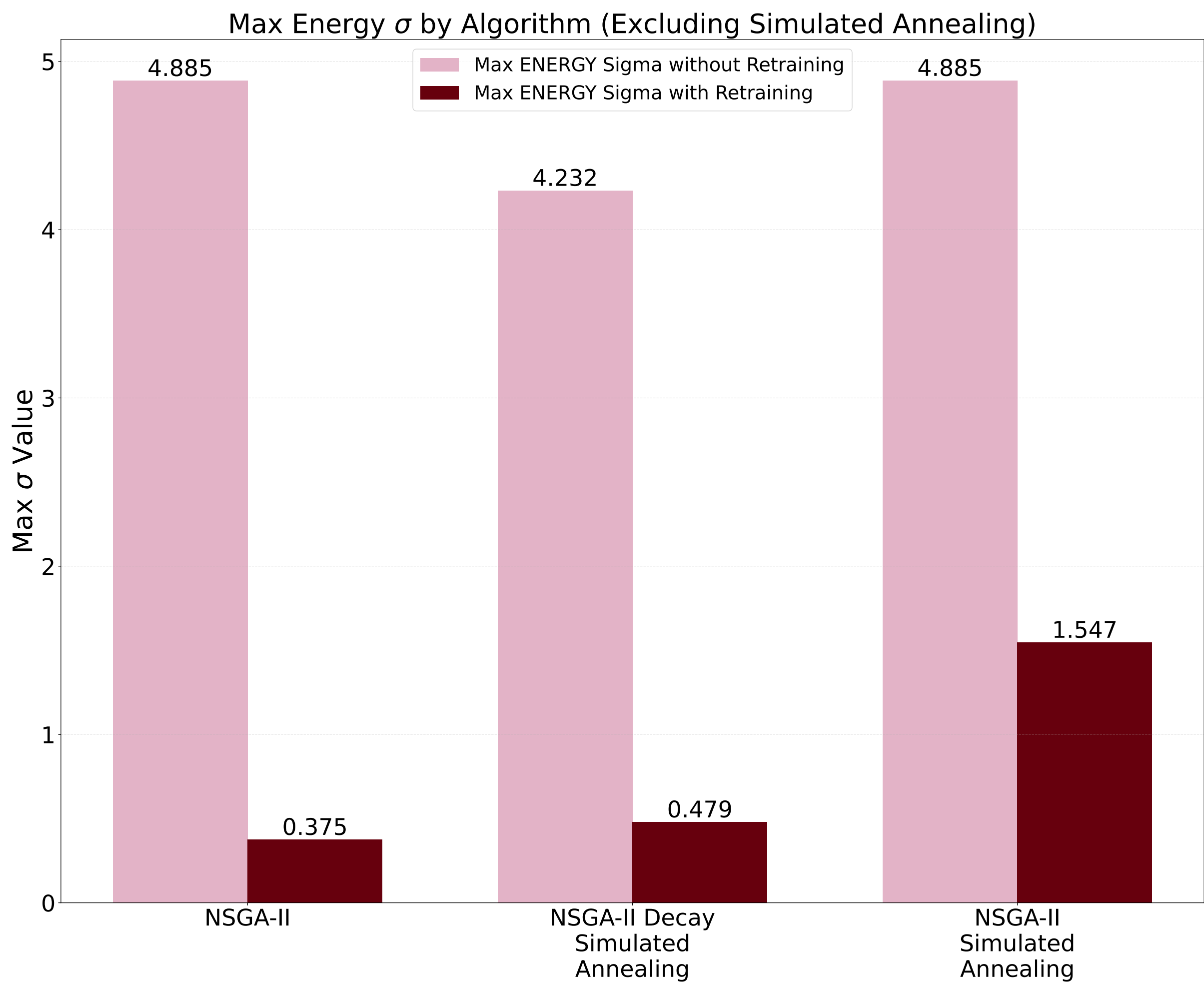


Figure 5.18: Comparison of the maximum energy σ values for different algorithms with retraining optimization enabled and disabled.

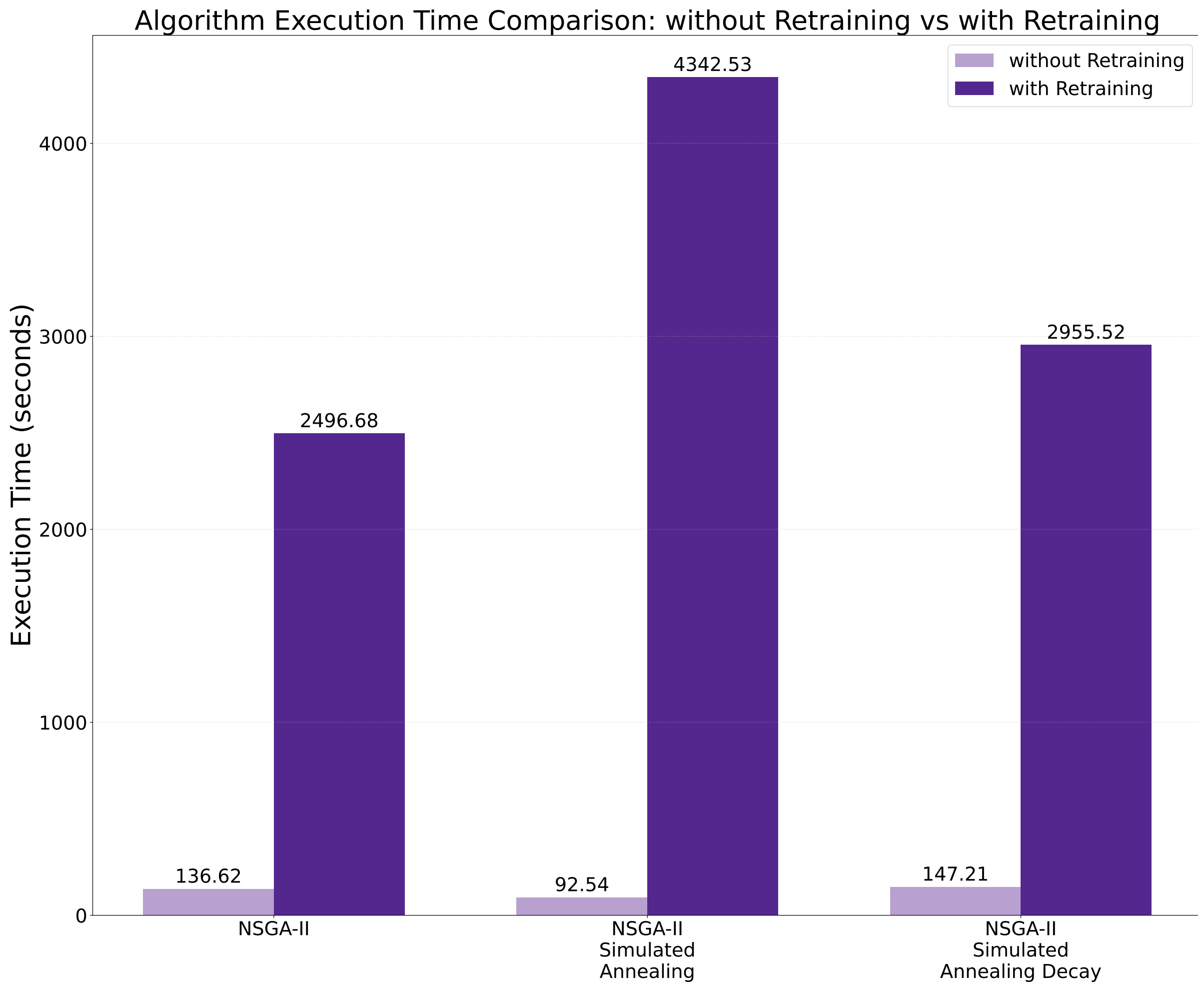


Figure 5.19: Execution time comparison for different algorithms with retraining optimization enabled and disabled.

Following, we present the same set of experiments discussed in the previous sections, now executed under identical configurations but with the **retraining optimization** enabled. The retraining mechanism is applied every 20 generations, and at each retraining event, four additional samples are selected from the current population—typically chosen among both the best and the worst individuals—to enrich the training dataset of the surrogate models. The objective of this analysis is to verify whether the main behaviors observed in the previous experiments remain consistent when this additional optimization stage is active, and to understand the degree to which retraining improves predictive accuracy and search stability.

Overall, the key dynamics identified earlier—such as the impact of mutation adaptation, the dependence on the offspring/survivor ratio, and the general progression of the population across generations—remain clearly visible. One noteworthy effect introduced by

retraining is the improvement in the predictive precision of the surrogate models, particularly for the standard NSGA-II configuration without additional optimizations. This is reflected by the consistently lower uncertainty values (i.e., reduced maximum σ) and the improved smoothness of the hypervolume curves, demonstrating a more reliable evaluation of candidate solutions throughout the evolutionary process.

Homogeneous configuration with adaptation. We begin with the homogeneous configuration generating 125 offspring per generation. As shown in Figure 5.21, the hypervolume curve exhibits a marked upward jump shortly after the retraining step. This abrupt improvement confirms that the updated surrogate model significantly enhances the quality of the search, allowing the algorithm to more accurately identify promising trade-offs and expand the Pareto front in previously uncertain regions of the design space. Although the basic NSGA-II algorithm achieves higher hypervolume values than its individually optimized variants, a detailed comparison of Pareto dominance (Figure 5.20) and ADRS (Figure 5.22) distances confirms once again that the NSGA-II configuration with both optimization stages remains the reference front. This is consistent with the observations from Section 5.2.1, reinforcing the robustness of this configuration even when additional modeling accuracy is introduced through retraining.

Effect of offspring size under retraining. The influence of offspring population size also remains evident when retraining is enabled. Figures 5.24–5.29 demonstrate that the balance between *exploration* and *exploitation* continues to be strongly shaped by the number of offspring generated at each generation. With 50 offspring, the search becomes more exploitative, converging faster but covering a narrower portion of the Pareto front. In contrast, increasing the number of offspring to 200 significantly broadens the diversity of sampled solutions, enabling the algorithm to investigate a more extensive region of the design space and to generate a denser, more uniformly distributed Pareto set. The effect is particularly visible in the spacing and hypervolume plots, where larger offspring populations lead to improved coverage and more homogeneous fronts.

Importantly, these trends remain fully consistent with the behaviors observed in Section 5.2.2 without retraining. This demonstrates that the retraining optimization enhances prediction accuracy without altering the fundamental search dynamics dictated by mutation adaptation and offspring distribution. Retraining therefore acts as a complementary mechanism: it improves the reliability of the surrogate model while preserving the characteristic exploration–exploitation balance induced by the evolutionary algorithm and its parameters.

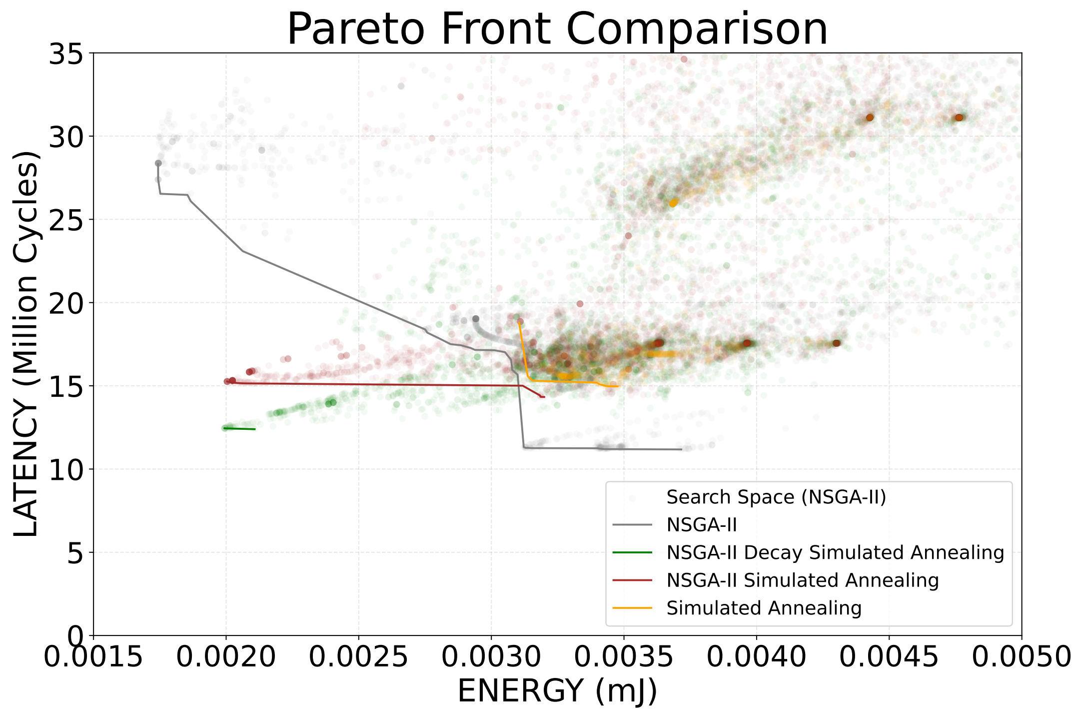


Figure 5.20: Pareto analysis.

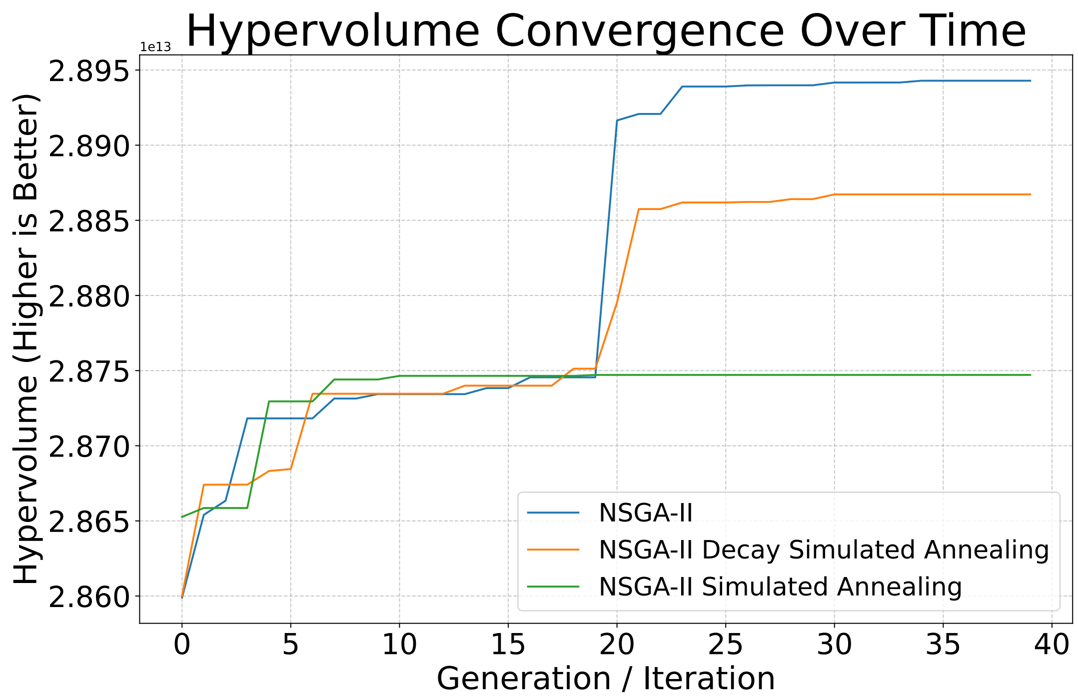


Figure 5.21: Hypervolume behavior.

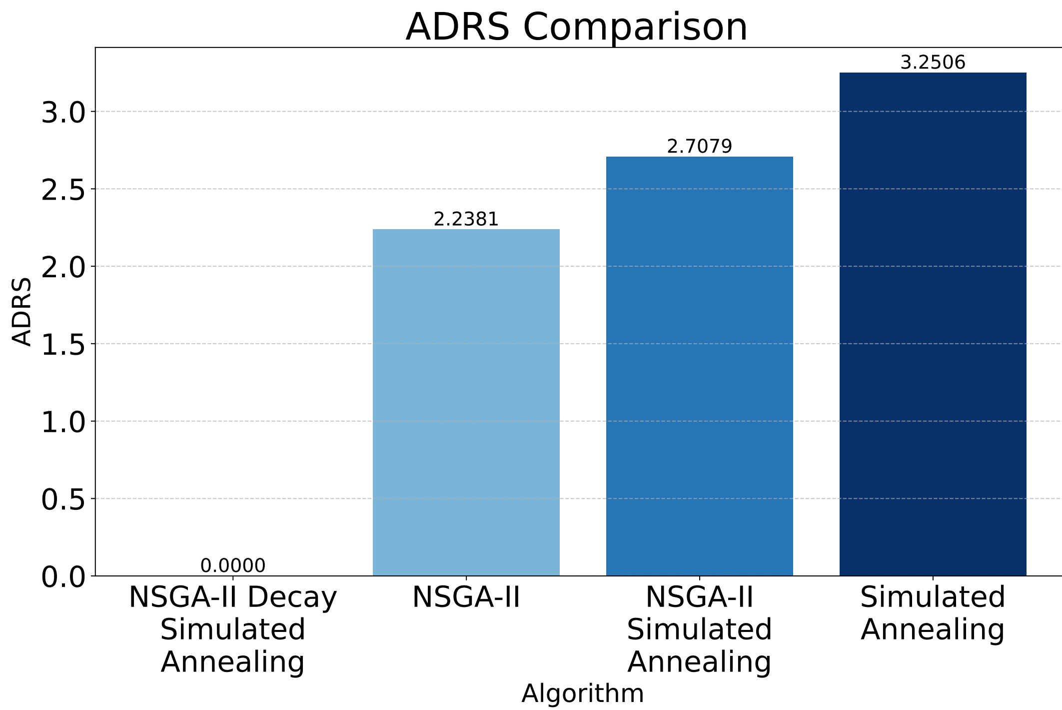


Figure 5.22: ADRS comparison.

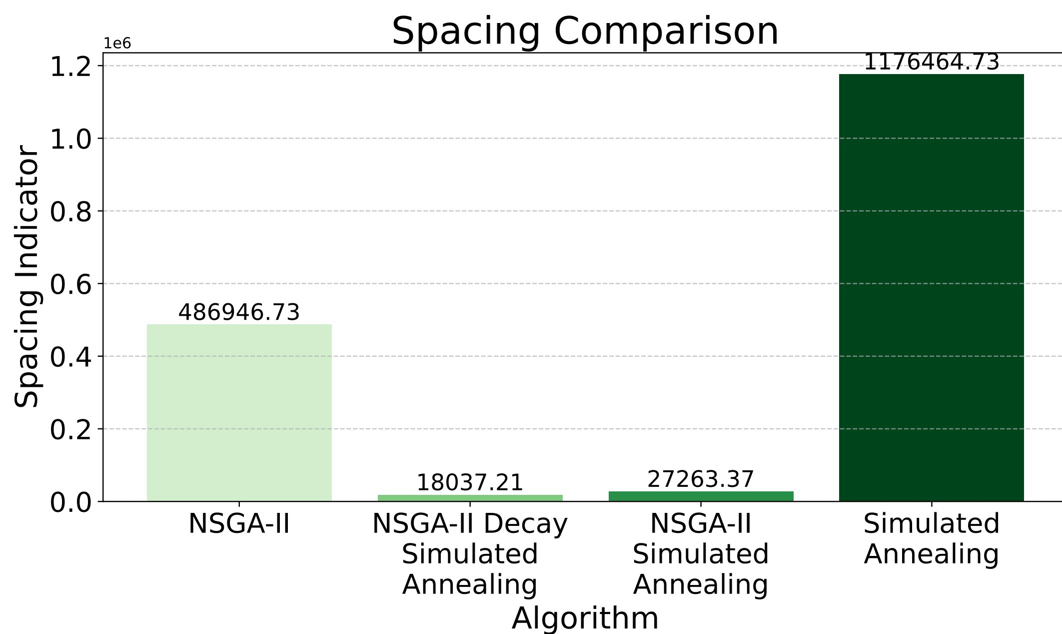


Figure 5.23: Spacing comparison.

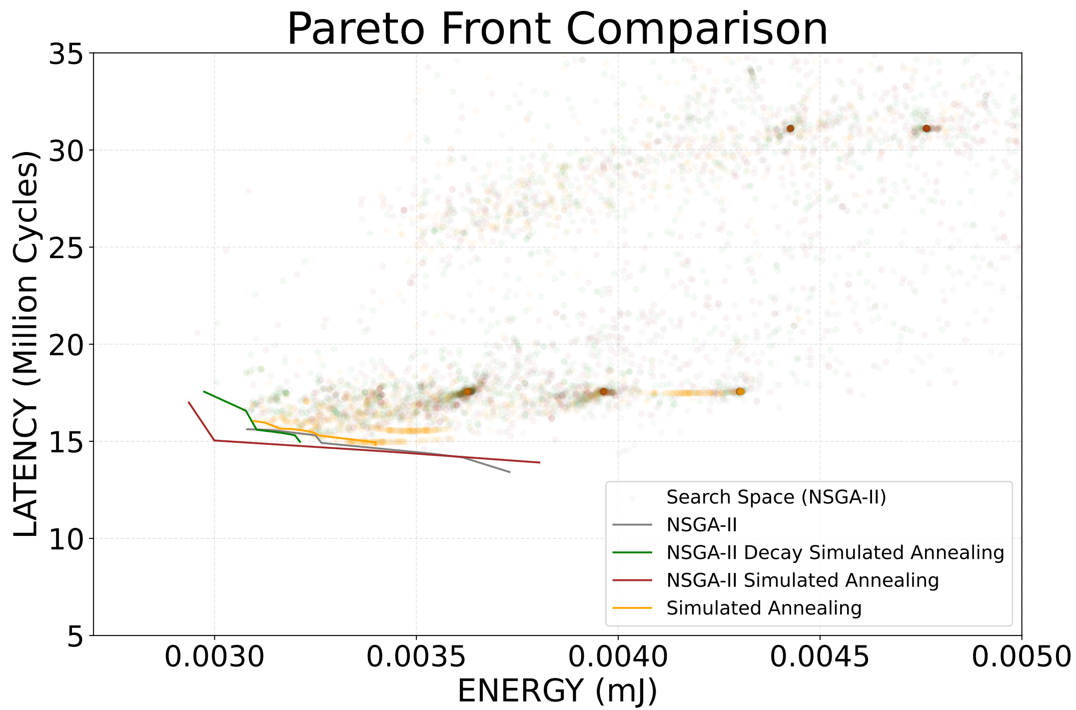


Figure 5.24: Pareto analysis 50 Offsprings.

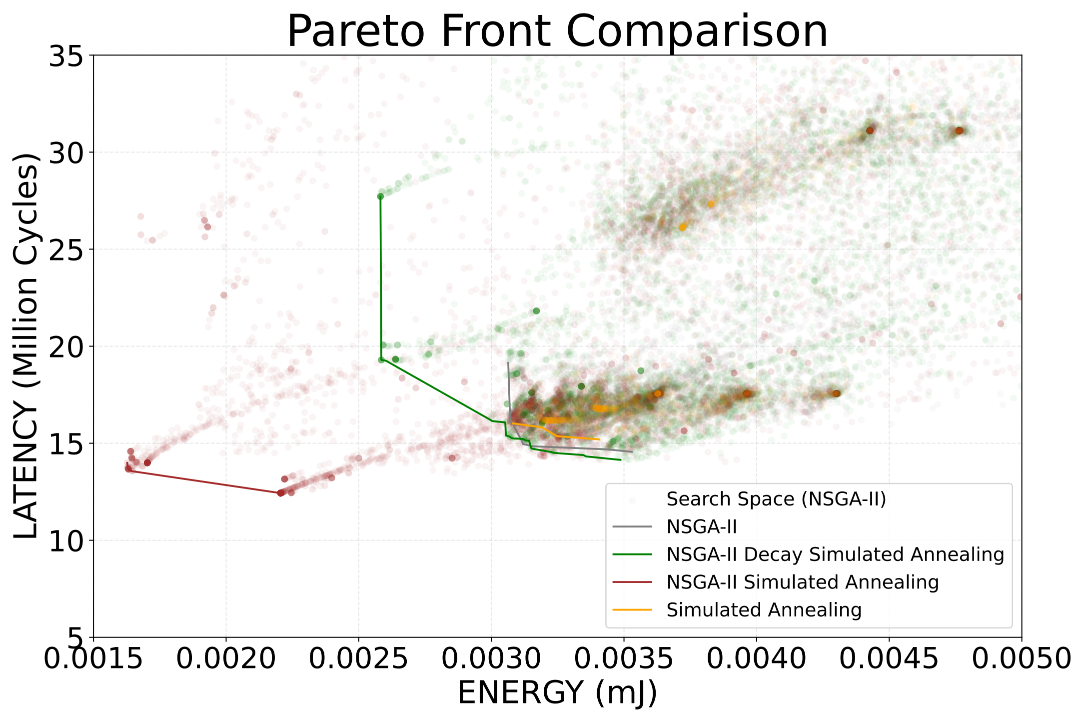


Figure 5.25: Pareto analysis 200 offsprings.

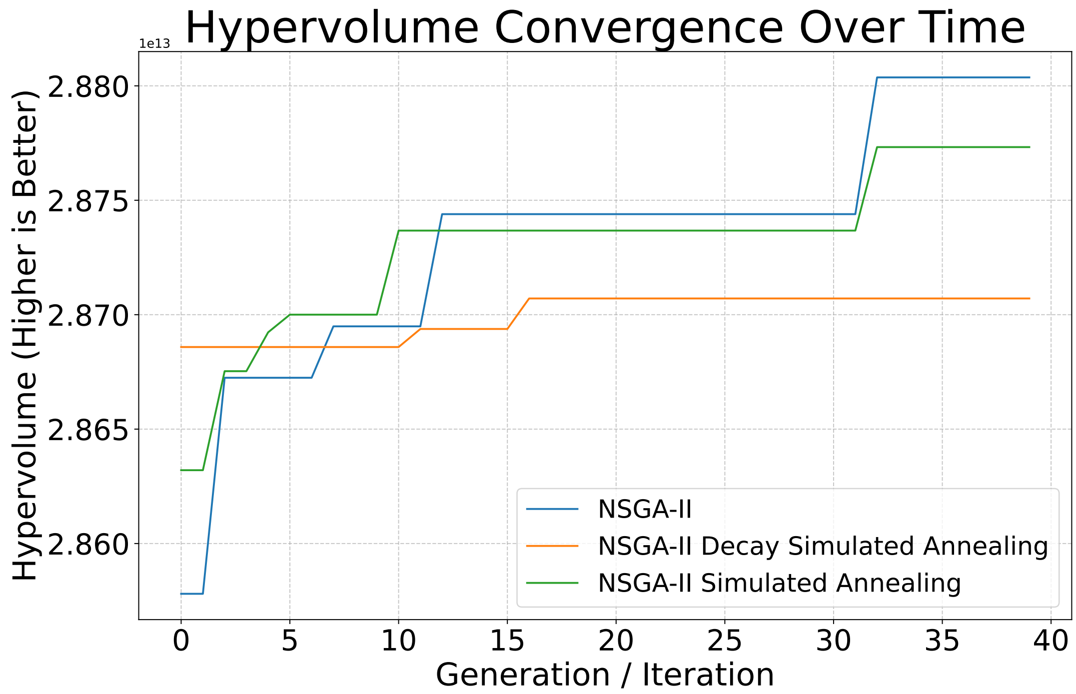


Figure 5.26: Hypervolume behavior 50 offsprings.

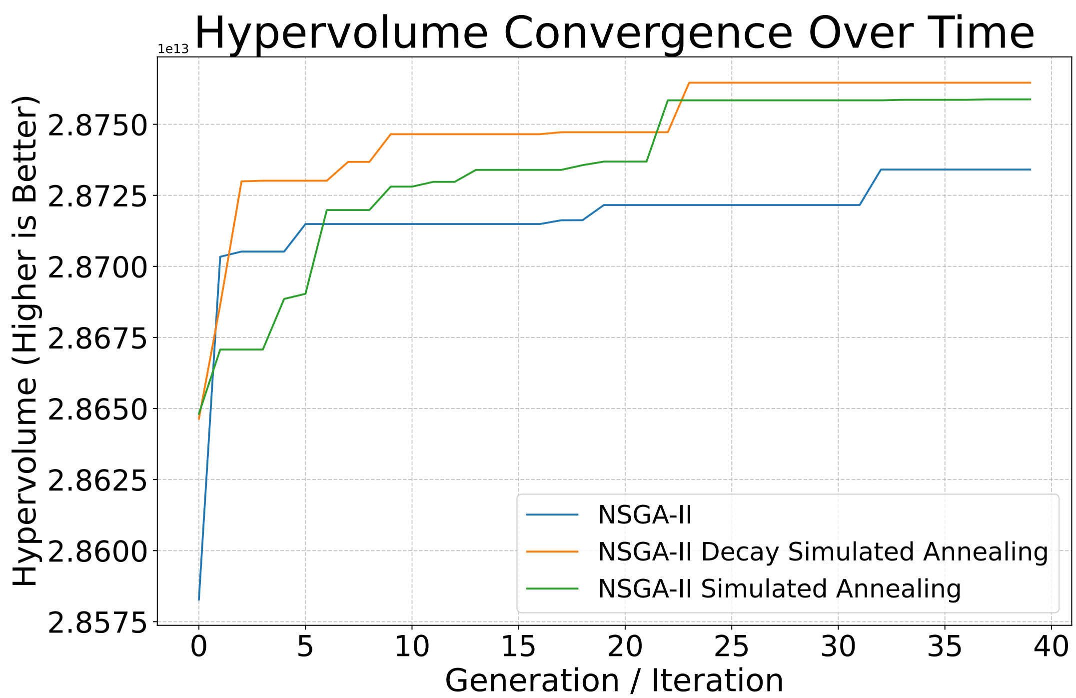


Figure 5.27: Hypervolume behavior 200 offsprings.

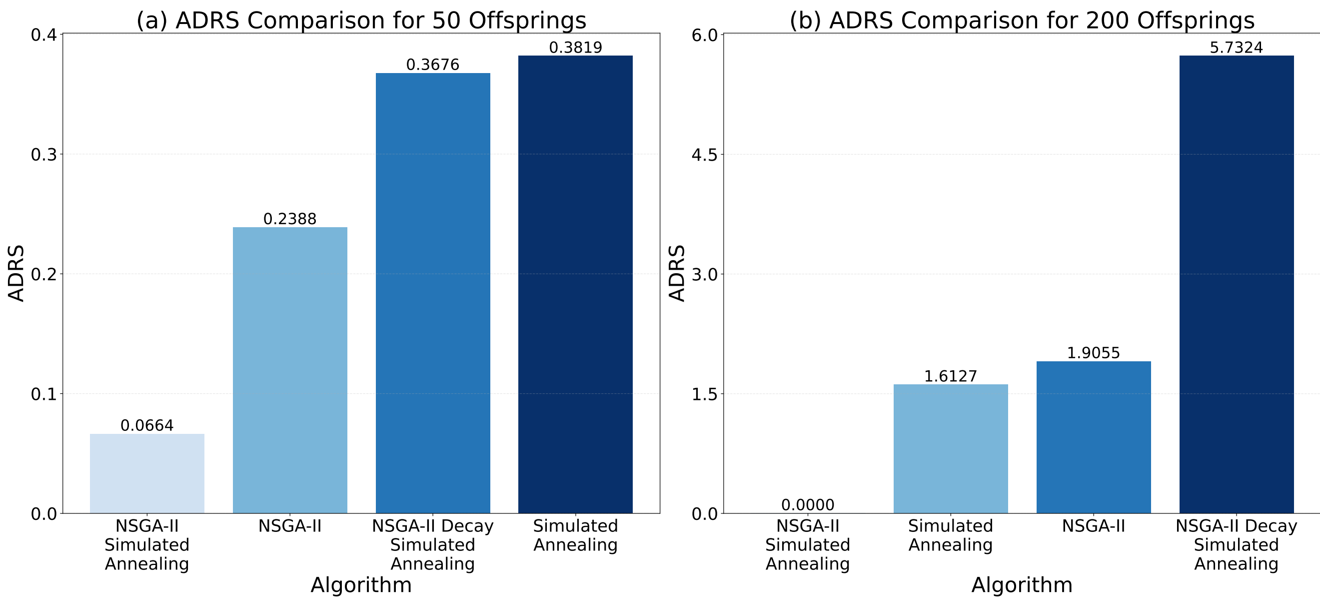


Figure 5.28: ADRS Comparison.

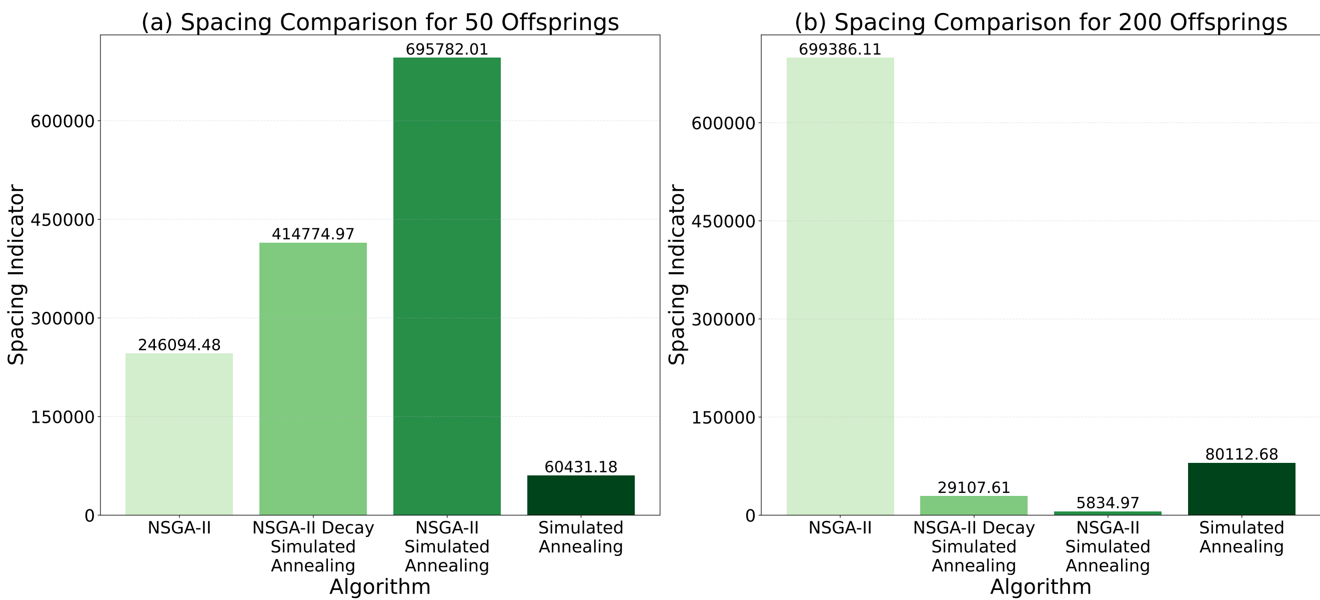


Figure 5.29: Spacing Comparison.

5.3. Quantitative Evaluation of Improvements Against the Baseline

To conclude the experimental analysis, this section presents a quantitative comparison between the Pareto-optimal solutions obtained using the proposed methodology and a predefined baseline architecture. The objective is to highlight the performance improvements achieved in terms of both **energy efficiency** and **latency reduction**, demonstrating the effectiveness of the developed exploration framework.

For this purpose, using the *FactorFlow* framework, a reference architecture inspired by the *Colonnade* design (described in Section 3.1.1) was modeled and evaluated. The modeled system consists of a stack of FactorFlow computational levels, specifically organized into two fanout levels, each configured with a mesh of 32×32 (representing 32 bitlines and 32 wordlines), corresponding to an IMC array composed of 32×32 computational cells. In addition, registers for partial result storage were included to ensure a realistic representation of accumulation processes and internal data flow dynamics.

The quantitative comparison, therefore, focuses on assessing how the architectures obtained through the design space exploration improve upon the baseline in terms of **energy consumption** and **computational latency**, providing a concrete and measurable indication of the benefits introduced by the proposed optimization methodology.

In Figure 5.30, the Pareto front points are shown, referring to the front obtained by the NSGA-II algorithm with both optimizations enabled and mutation adaptation active, without retraining, and with a homogeneous distribution of offspring (set to 125). This configuration was found to deliver the best overall results, as discussed in Section 5.2, producing a well-distributed and dominant front positioned close to the optimal Pareto front derived from the full search.

As can be observed, the improvement over the baseline Colonnade-inspired architecture is substantial: approximately a 99% reduction in energy consumption and a 41–44% improvement in latency, measured between the two extremes of the Pareto front.

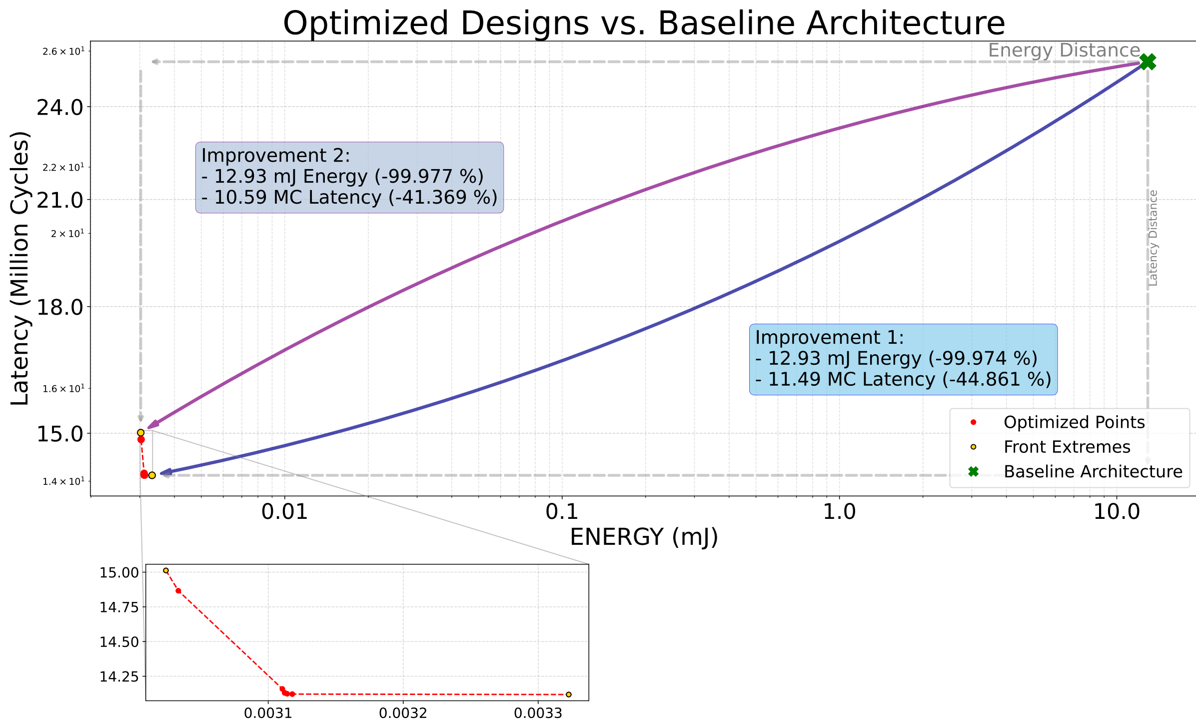


Figure 5.30: Improvement of the NSGA-II Decay and Simulated Annealing Pareto front from a Colonnade-inspired architecture.

6 | Conclusions

6.1. Summary

This thesis began with an in-depth study of *In-Memory Computing* (IMC) technology, focusing on the fundamental challenges it addresses, the limitations of traditional processor-centric architectures, and the key principles underlying its operation. The investigation also examined the primary application domains of IMC and the diverse architectural strategies employed in hardware systems that leverage this paradigm.

Subsequently, a comprehensive review of existing IMC frameworks and architectures was conducted to identify a suitable evaluation environment. Among the analyzed state-of-the-art solutions, **FactorFlow** emerged as the most appropriate framework for this work, with **CIMLoop** considered as an alternative candidate for cross-validation purposes.

The research then expanded into the fields of *Design Space Exploration* (DSE) and *Design of Experiments* (DoE), emphasizing the optimization algorithms most relevant to architectural co-design and performance tuning. After a comparative analysis of different approaches, **genetic algorithms**, and in particular the **NSGA-II** (Non-dominated Sorting Genetic Algorithm II), were selected as the core optimization strategy due to their balance between exploration and convergence efficiency.

Following this theoretical groundwork, the first significant technical milestone involved integrating IMC evaluation capabilities into the **FactorFlow** framework. This task required a detailed understanding of the framework's internal mechanics and constraints to ensure full compatibility without altering its fundamental interface. Specific extensions and validation mechanisms were introduced to handle IMC-specific architectural parameters and evaluation flows correctly.

Once the integration was complete, the next step focused on developing the DSE algorithm tailored to IMC-based design optimization. The algorithm was implemented to generate, organize, and evaluate candidate architectures efficiently, starting from a structured *Latin Hypercube Sampling* (LHS) initialization to guarantee adequate coverage of the parameter

space. Subsequent optimization phases exploited iterative improvements through NSGA-II and additional random-based exploration techniques to enhance population diversity and search robustness.

The experimental evaluation compared multiple algorithmic configurations — both base and optimized — using a set of performance metrics, including hypervolume, spacing, and ADRS distance, to assess convergence behavior and exploration quality. These results enabled the identification of a configuration that optimally balances exploration, exploitation, and computational efficiency for IMC-oriented design problems.

Finally, the obtained results were quantitatively compared against a predefined **baseline architecture**, modeled after the *Colonnade*-inspired IMC macro described in Section 3.1.1. Using the FactorFlow modeling environment, the baseline and the optimized architectures were evaluated in terms of both **energy consumption** and **latency performance**. The analysis revealed substantial improvements: approximately **99% reduction in energy consumption** and between **41–44% improvement in latency**, with numerical differences of about **12.93 mJ** in energy and between **10 ns** and **11 ns** in latency across the extremes of the Pareto front. These results confirm that the proposed exploration methodology is capable not only of identifying optimized architectural configurations but also of surpassing the performance of conventional IMC architectures modeled after state-of-the-art designs.

In summary, this thesis successfully demonstrated the feasibility and efficiency of integrating IMC evaluation within a design space exploration framework, achieving a high degree of exploration accuracy and optimization quality. Furthermore, the methodology proved effective in identifying architectures with superior **energy–latency trade-offs** compared to existing reference designs, underscoring its potential for guiding future IMC system co-design and optimization.

6.2. Future Work

Based on the methodology and results presented in this work, several future research directions can be envisioned to develop further and extend the proposed framework. These directions involve both algorithmic improvements and implementation-level optimizations, as well as potential integrations with other state-of-the-art IMC tools.

A first promising development would be the direct integration with **CIMLoop** [2]. As discussed in the previous chapters, CIMLoop relies on `.yaml` configuration files to describe its architectural models. A feasible approach would involve parsing these files to

enable both direct evaluation of CIMLoop-generated macros and the use of the framework as an evaluator within the proposed exploration methodology. This, however, would require significant coding effort, since even with the use of the `PyYAML` library, CIMLoop's configuration files include several custom and non-standard data container types. Therefore, a considerable amount of manual work would be needed to define the corresponding PyYAML classes and ensure a correct mapping between CIMLoop's parameters and the internal representation used by **FactorFlow**. Once this interface is established, the next step would consist of generating FactorFlow-based architecture models directly from the parsed elements, enabling a systematic performance comparison and validation between FactorFlow and CIMLoop evaluations.

Another valuable improvement concerns the **extension of the IMC characterization dataset**. By increasing the amount of available data for IMC cells—covering a wider range of sparsity levels, array sizes, and precision modes—it would be possible to better understand and explore a more diverse and representative set of architectural configurations. This would enrich the design space and provide more reliable evaluations of IMC behavior under different workload and mapping conditions.

From the Design Space Exploration (DSE) perspective, further optimizations could be introduced to improve the exploration process itself. Currently, some architectural parameters (such as parallelism dimensions) are treated as fixed values; allowing these parameters to be explored dynamically, possibly under specific constraints, could enable a deeper investigation of mapping strategies as part of the exploration. Increasing the number of design parameters would naturally expand the search space. Thus, the computational complexity could also lead to the discovery of more diverse and efficient IMC configurations.

During the analysis phase, one of the main challenges encountered was the presence of high **uncertainty values** (σ) in the Gaussian Process predictions. While the implemented retraining optimization mitigates this issue, a more advanced **dynamic retraining strategy** could be adopted. In this approach, retraining would be triggered adaptively whenever the model's uncertainty exceeds a predefined threshold. Since the current implementation already measures the correlation between σ values and actual prediction errors, integrating this adaptive mechanism would be relatively straightforward. Such a strategy would enable a more intelligent use of retraining, reducing unnecessary executions and minimizing the trade-off between accuracy and runtime. Alternatively, increasing the number of initial training samples could improve model accuracy at the cost of higher startup time, though uncertainty would still increase as generations progress.

Finally, several **implementation-level optimizations** could be pursued to improve computational efficiency. As discussed in Section 4.5, the current version of the framework uses a multiprocessing paradigm to ensure compatibility with the FactorFlow execution model. Although effective, this approach incurs noticeable overhead in both execution time and resource usage. With the release of Python 3.14[26] (October 2025), full support for *free threading* is expected to become stable and compatible with most widely used scientific libraries. Leveraging this new feature would allow the framework to execute fully in parallel without relying on multiprocessing, thereby significantly reducing resource contention and improving runtime performance.

Bibliography

[1] CADContest_2022_problem_c_20220822.pdf. URL https://drive.google.com/file/d/1GEIHfqPl42ncJKT2KABENC5olB7lpP2W/view?usp=sharing&usp=embed_facebook.

[2] T. Andrulis, J. S. Emer, and V. Sze. CiMLoop: A flexible, accurate, and fast compute-in-memory modeling tool. *arXiv*. doi: 10.48550/ARXIV.2405.07259. URL https://arxiv.org/abs/2405.07259. Publisher: arXiv Version Number: 2.

[3] J. Blank and K. Deb. pymoo: Multi-objective optimization in python. *IEEE Access*, 8:89497–89509, 2020.

[4] M. Črepinšek, S.-H. Liu, and M. Mernik. Exploration and exploitation in evolutionary algorithms: A survey. *ACM Computing Surveys (CSUR)*, 45(3):1–33, 2013. Modern survey covering population size as a diversity mechanism.

[5] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. 6(2):182–197. ISSN 1941-0026. doi: 10.1109/4235.996017. URL https://ieeexplore.ieee.org/document/996017/.

[6] D. Delahaye, S. Chaimatanan, and M. Mongeau. Simulated annealing: From basics to applications. In M. Gendreau and J.-Y. Potvin, editors, *Handbook of Metaheuristics*, volume 272, pages 1–35. Springer International Publishing. ISBN 978-3-319-91085-7 978-3-319-91086-4. doi: 10.1007/978-3-319-91086-4_1. URL http://link.springer.com/10.1007/978-3-319-91086-4_1. Series Title: International Series in Operations Research & Management Science.

[7] G. Desoli, N. Chawla, T. Boesch, M. Avodhyawasi, H. Rawat, H. Chawla, V. Abhijith, P. Zambotti, A. Sharma, C. Cappetta, M. Rossi, A. De Vita, and F. Girardi. 16.7 a 40-310tops/w SRAM-based all-digital up to 4b in-memory computing multi-tiled NN accelerator in FD-SOI 18nm for deep-learning edge applications. In *2023 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 260–262. doi: 10.1109/ISSCC42615.2023.10067422. URL https://ieeexplore.ieee.org/document/10067422. ISSN: 2376-8606.

[8] B. Durakovic. Design of experiments application, concepts, examples: State of the art. *Periodicals of Engineering and Natural Sciences (PEN)*, 5(3):421–439, 2017. doi: 10.21533/pen.v5i3.145.

[9] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. URL `http://arxiv.org/abs/1512.03385`.

[10] J. Ho, A. Jain, and P. Abbeel. Denoising diffusion probabilistic models. *Advances in Neural Information Processing Systems*, 2020.

[11] C. Hong, Q. Huang, G. Dinh, M. Subedar, and Y. S. Shao. DOSA: Differentiable model-based one-loop search for DNN accelerators. In *56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 209–224. ACM. doi: 10.1145/3613424.3623797. URL `https://dl.acm.org/doi/10.1145/3613424.3623797`.

[12] M. Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, 2014. doi: 10.1109/ISSCC.2014.6757323.

[13] A. Jaiswal, I. Chakraborty, A. Agrawal, and K. Roy. 8t SRAM cell as a multibit dot-product engine for beyond von neumann computing. 27(11):2556–2567. ISSN 1557-9999. doi: 10.1109/TVLSI.2019.2929245. URL `https://ieeexplore.ieee.org/document/8802267`. Conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems.

[14] N. P. Jouppi, D. H. Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma, T. Norrie, N. Patil, S. Prasad, C. Young, Z. Zhou, and D. Patterson. Ten lessons from three generations shaped google's tpuv4i. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ISCA '21, page 1–14. IEEE Press, 2021. ISBN 9781450390866. doi: 10.1109/ISCA52012.2021.00010. URL `https://doi.org/10.1109/ISCA52012.2021.00010`.

[15] S.-C. Kao, M. Pellauer, A. Parashar, and T. Krishna. DiGamma: Domain-aware genetic algorithm for HW-mapping co-optimization for DNN accelerators. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 232–237. IEEE. ISBN 978-3-9819263-6-1. doi: 10.23919/DATE54114.2022.9774568. URL `https://ieeexplore.ieee.org/document/9774568/`.

[16] H. Kim, T. Yoo, T. T.-H. Kim, and B. Kim. Colonnade: A reconfigurable SRAM-based digital bit-serial compute-in-memory macro for processing neural networks. 56(7):2221–2233. ISSN 0018-9200, 1558-173X. doi: 10.1109/JSSC.2021.3061508. URL `https://ieeexplore.ieee.org/document/9373949/`.

[17] N. Lepri, A. Glukhov, L. Cattaneo, M. Farronato, P. Mannocci, and D. Ielmini. In-memory computing for machine learning and deep learning. 11:587–601. ISSN 2168-6734. doi: 10.1109/JEDS.2023.3265875. URL `https://ieeexplore.ieee.org/document/10103697/`.

[18] S. Li, C. Bai, X. Wei, B. Shi, Y.-K. Chen, and Y. Xie. 2022 ICCAD CAD contest problem c: Microarchitecture design space exploration. In *2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–7. URL `https://ieeexplore.ieee.org/document/10069474/`. ISSN: 1558-2434.

[19] C. D. Lin and B. Tang. Latin hypercubes and space-filling designs, 2022. URL `https://arxiv.org/abs/2203.06334`.

[20] O. Mutlu, S. Ghose, J. Gómez-Luna, R. Ausavarungnirun, M. Sadrosadati, and G. F. Oliveira. A modern primer on processing in memory, 2025. URL `https://arxiv.org/abs/2012.03112`.

[21] G. Palermo, C. Silvano, and V. Zaccaria. ReSPIR: A response surface-based pareto iterative refinement for application-specific design space exploration. 28 (12):1816–1829. ISSN 1937-4151. doi: 10.1109/TCAD.2009.2028681. URL `https://ieeexplore.ieee.org/document/5324029/`.

[22] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer. Timeloop: A systematic approach to DNN accelerator evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 304–315. IEEE. ISBN 978-1-7281-0746-2. doi: 10.1109/ISPASS.2019.00042. URL `https://ieeexplore.ieee.org/document/8695666/`.

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[24] S. Perri, C. Zambelli, D. Ielmini, and C. Silvano. Digital In-Memory Computing to Accelerate Deep Learning Inference on the Edge. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 130–133. IEEE, 2024. doi: 10.1109/IPDPSW63119.2024.00037.

[25] Python Software Foundation. Python (Version 3.13.0, free-threading variant) [Computer software], 2024. URL `https://www.python.org/downloads/release/python-3130/`. Accessed: 2025-04-14. For details on the free-threading build,

see the official documentation: https://docs.python.org/3/howto/free-threading-python.html.

[26] Python Software Foundation. Python (Version 3.14.0 [Computer software], 2025. URL `https://www.python.org/downloads/release/python-3140/`. For details, see the official documentation: https://docs.python.org/3/whatsnew/3.14.html.

[27] Y. Qi, S. Zhang, and T. M. Taha. TRIM: A design space exploration model for deep neural networks inference and training accelerators. 42(5):1648–1661. ISSN 1937-4151. doi: 10.1109/TCAD.2022.3203959. URL `https://ieeexplore.ieee.org/document/9874869/`.

[28] M. Ronzani and C. Silvano. FactorFlow: Mapping GEMMs on spatial architectures through adaptive programming and greedy optimization. In *Proceedings of the 30th Asia and South Pacific Design Automation Conference*, pages 706–712. ACM. ISBN 979-8-4007-0635-6. doi: 10.1145/3658617.3697670. URL `https://dl.acm.org/doi/10.1145/3658617.3697670`.

[29] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. URL `http://arxiv.org/abs/1409.1556`.

[30] C. Song and R. Kawai. Monte carlo and variance reduction methods for structural reliability analysis: A comprehensive review. *Probabilistic Engineering Mechanics*, 73:103479, 2023. ISSN 0266-8920. doi: https://doi.org/10.1016/j.probengmech.2023.103479. URL `https://www.sciencedirect.com/science/article/pii/S0266892023000681`.

[31] J. Sun, P. Houshmand, and M. Verhelst. Analog or digital in-memory computing? benchmarking through quantitative modeling. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9. doi: 10.1109/ICCAD57390.2023.10323763. URL `http://arxiv.org/abs/2405.14978`.

[32] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, 2017.

[33] J. Wang. An intuitive tutorial to gaussian process regression. 25(4):4–11. ISSN 1521-9615, 1558-366X. doi: 10.1109/MCSE.2023.3342149. URL `http://arxiv.org/abs/2009.10862`.

[34] K. Zheng, R.-J. Yang, H. Xu, and J. Hu. A new distribution metric for comparing

pareto optimal solutions. *Structural and Multidisciplinary Optimization*, 2016. doi: 10.1007/s00158-016-1469-3.

List of Figures

2.1 Example of a ResNet building block with residual (skip) connection. 8
2.2 Evolution of memory–compute integration strategies. From left to right: (1) traditional von Neumann architectures with off-chip memory access, (2) near-memory computing using closely coupled DRAM/NVM modules, (3) on-chip memory integration with SRAM or NVM arrays, and (4) true In-Memory Computing (IMC), where computation is performed directly inside the memory array, minimizing data movement. 11
2.3 System-level perspective of Digital and Analog IMC architecture. 13

3.1 Desoli Digital IMC architecture. 35

4.1 Template Spatial Architecture without IMC integration. 47
4.2 Template architecture including IMC integration through the newly introduced `CIM Memory` and `CIM Compute` levels. 47
4.3 Simulated 8T SRAM cell for digital In-Memory Computing (IMC) in 22 nm technology. 48
4.4 Simulated digital In-Memory Computing (IMC) array, including the integrated adder-tree for partial-sum accumulation. 49
4.5 Relationship between energy consumption and V_{DD}, illustrating the characteristic quadratic dependency. 51
4.6 Polynomial fitting curve (set of blue points) compared with simulation data (red points) for the write operation with $Vdd = 0.8V$. 51
4.7 Description of CLA 1-bit adder sum circuit block implemented using CMOS current mirror logic. 54
4.8 Description of CLA 1-bit adder carry circuit block implemented using CMOS current mirror logic. 55
4.9 Overview of the setup phase, illustrating the workflow from parameter space sampling to FactorFlow architecture instantiation and Gaussian Process model training. 56

4.10 Evolutionary exploration workflow: population initialization via LHS, surrogate-assisted offspring evaluation (GPR), selection and mutation (NSGA-II). . . 68

5.1 Pareto hypervolume: (a) 2D representation and (b) 3D generalization. . . . 84
5.2 Computation of the minimum Manhattan distances d_i. 87
5.3 Pareto analysis. 92
5.4 Hypervolume behavior. 92
5.5 ADRS comparison. 93
5.6 Spacing comparison. 93
5.7 Pareto analysis. 96
5.8 Hypervolume behavior. 96
5.9 ADRS comparison. 97
5.10 Spacing comparison. 97
5.11 Pareto analysis 50 Offsprings. 100
5.12 Pareto analysis 200 offsprings. 100
5.13 Hypervolume behavior 50 offsprings. 101
5.14 Hypervolume behavior 200 offsprings. 101
5.15 ADRS Comparison. 102
5.16 Spacing comparison. 102
5.17 ADRS distance comparisons across various configurations, using the full search Pareto front as reference (32 samples, no retraining). 104
5.18 Comparison of the maximum energy σ values for different algorithms with retraining optimization enabled and disabled. 107
5.19 Execution time comparison for different algorithms with retraining optimization enabled and disabled. 108
5.20 Pareto analysis. 110
5.21 Hypervolume behavior. 110
5.22 ADRS comparison. 111
5.23 Spacing comparison. 111
5.24 Pareto analysis 50 Offsprings. 112
5.25 Pareto analysis 200 offsprings. 112
5.26 Hypervolume behavior 50 offsprings. 113
5.27 Hypervolume behavior 200 offsprings. 113
5.28 ADRS Comparison. 114
5.29 Spacing Comparison. 114
5.30 Improvement of the NSGA-II Decay and Simulated Annealing Pareto front from a Colonnade-inspired architecture. 116

List of Tables

2.1 Energy per operation with a 45nm and 7nm technology node. 10

4.1 Available design parameters for the target arrays. For the smallest 4×4 arrays, only subsets of activity levels were considered due to limited cells: sparsity 25 %, 50 %, 75 % (*read*/*hold*) and switching 25 %, 50 %, 75 %, 100 % (*write*); other combinations were omitted (NaN). 50

4.2 Read energy consumptions for different tile configurations at varying V_{DD}, array size and sparsity. 52

4.3 Summary of the architectural parameters explored during the DSE. Each entry specifies its data type and admissible domain or set of choices. . . . 59

5.1 Fundamental configuration parameters of the NSGA-II algorithm. 88

5.2 Optimizations configuration parameters. 88

6.1 List of Acronyms 131

6.2 List of Symbols 132

List of Symbols

Table 6.1: List of Acronyms

Acronym	Full Name
AI	Artificial Intelligence
CNN	Convolutional Neural Network
DNN	Deep Neural Network
IMC	In-Memory Computing
MAC	Multiply–Accumulate
DSE	Design Space Exploration
DoE	Design of Experiments
LHS	Latin Hypercube Sampling
NSGA-II	Non-dominated Sorting Genetic Algorithm II
SA	Simulated Annealing
GP	Gaussian Process
MOEA	Multi-Objective Evolutionary Algorithm
AIMC	Analog In-Memory Computing
DIMC	Digital In-Memory Computing
DACE	Design and Analysis of Computer Experiments
PE	Processing Element
CLA	Carry-Lookahead Adder

Table 6.2: List of Symbols

Symbol	Description
F	Pareto front
N	Number of elements in a population
G	Number of generations
O	Number of offsprings per generation
S	Number of survivors per generation
R	Number of generations (or repetitions)
m	Number of objectives
P_t	Population at generation t
Q_t	Offsprings at generation t
R_t	Combined population (parents + offspring) at generation t
p	Element of the population
n_p	Domination count for individual p
S_p	Set of solutions dominated by individual p
$HV(F)$	Hypervolume of the Pareto front F
$ADRS$	Average Distance to Reference Set metric
μ_E	Gaussian Process prediction for energy
μ_L	Gaussian Process prediction for latency
σ_E	Standard deviation (uncertainty) of GP prediction for energy
σ_L	Standard deviation (uncertainty) of GP prediction for latency
η	Mutation power in NSGA-II
r_M	Mutation ratio in NSGA-II
f_i	Objective function i (e.g., energy or latency)
δ	Decay rate

Acknowledgements

Here you might want to acknowledge someone.