Journal of the Brazilian Computer Society, 2025, 31:1, doi: 10.5753/jbcs.2025.4912

Efficient Number of Functional Units and Loop Pipeline Design Space Exploration for High-Level Synthesis

Leandro de Souza Rosa [**Alma Mater Studiorum - Università di Bologna** | ***leandro.desouzarosa@unibo.it***]
Christos-Savvas Bouganis [**Imperial College London** | ***savvas.bouganis@imperial.ac.uk***]
Vanderlei Bonato [**Universidade de São Paulo** | ***vbonato@usp.br***]

Dipartimento di Ingegneria dell'Energia Elettrica e dell'Informazione "Guglielmo Marconi", Alma Mater Studiorum - Università di Bologna, Viale del Risorgimento, 2, 40136 Bologna (BO), Italy.

Received: 07 September 2024 • **Accepted:** 31 May 2025 • **Published:** 05 August 2025

Abstract High-level synthesis compilers offer numerous directives for controlling hardware architecture implementations, leading to highly customized solutions, but also to large design spaces that are impractical to be fully explored due to the time-consuming stages of hardware compilation and synthesis. Traditional design space exploration approaches aim to identify architectures with the best hardware resources-performance balance. However, they usually consider the compilation process as a black box, failing to leverage relationships between directives and evaluation metrics to improve their efficiency. This paper analyses the relationship between the "number of functional units" and "loop pipeline" directives, which allow for balancing hardware area and computation time. For the former, we propose a novel path-based method to solve the shortcomings of traditional exploration approaches. For the latter, we propose a novel incremental exploration flow based on a Pareto-frontier evaluation. Results show improvements in exploration speed and quality of hardware designs when compared to established methods.

Keywords: Design Space Exploration, High-Level Synthesis, Loop, Directives.

1 Introduction

Over the past decade, the improved computation efficiency of hardware accelerators popularized their use on several applications, such as cloud computing [Kachris and Soudris, 2016], System-on-Chip (SoC) [Nane *et al.*, 2016], and edge-machine learning [Samanta *et al.*, 2024]. To overcome the expensive design of hardware accelerators, High-Level Synthesis (HLS) tools offer the capability of compiling high-level applications into custom hardware designs, promising to enable high-quality solutions in a fraction of the time when compared to traditional approaches [Nane *et al.*, 2016].

Hardware accelerators generated through HLS are typically composed of several Functional Units (FUs) (e.g., arithmetic units and memory ports) controlled by an Finite State Machine (FSM), and the number of Functional Units (nFUs) affects the design's throughput and energy consumption. HLS compilers offer a set of directives (pragmas, compiler optimizations, and code styles) to control the hardware architecture generation, allowing for balancing the final hardware metrics trade-offs (silicon area, latency, frequency, and energy consumption). As such, designers must perform a Design Space Exploration (DSE) to fine-tune the optimal directives combinations (configurations) to achieve the desired hardware metrics balance. Given the time-consuming compilation and synthesis stages needed to evaluate hardware designs, this task is laborious. Therefore, having a fast DSE method is key to unlocking the HLS potential.

Methods for speeding up the DSE for Multi-Processor System-on-Chip (MPSoC) are classified in into simulation-based methods, which provide hardware metrics estimations without fully implementing the designs, or analytical models, which compute hardware metrics by modelling the systems explicitly or from data [Pimentel, 2017]. Similarly to simulation approaches, in the HLS scope, estimation-based methods are used to avoid the lengthy synthesis process at the cost of a lower DSE accuracy.

For example, [Perina *et al.*, 2019; Bannwart Perina *et al.*, 2019] presents a latency estimator (number of clock cycles) which considers compiler and device-specific information to achieve precise results; [Wang *et al.*, 2020] creates estimations using a dataset composed of several synthesized kernels.

These approaches achieve the highest acceleration potential [Zhong *et al.*, 2017], but they often require different models for each metric, which are platform-dependent and not portable [Rosa, 2019]. For example, [Perina *et al.*, 2021] proposes to leverage knowledge about the scheduling, allocation and mapping steps to achieve precise estimations of the hardware performance, but they are tightened to the compilation platform used, and [Castro-Godínez *et al.*, 2020] uses models in a library for their estimations.

To reduce the estimation errors, [Fernando *et al.*, 2015; Cong *et al.*, 2017; Ali *et al.*, 2019] propose to map applications into pre-defined architecture templates optimized for the target platform, resulting in efficient designs. However, such solutions reduce flexibility and are restricted to applications and platforms matching the templates. Other interesting approaches use a database of high-level code and their matched pre-optimized designs for easily retrieving quasi-optimal configurations [Wang and Schafer, 2022], or matching designs with past-explorations though a similarity metric

[Ferretti *et al.*, 2020].

Another way to speed up the DSE is by evaluating fewer designs and selecting those with a higher chance of belonging to the Pareto-optimal front. This process is usually iterative, using the results of previous iterations to guide the current one [Schafer, 2016; Xydis *et al.*, 2015; Ferretti *et al.*, 2018b].

Additionally, the literature presents several alternative DSE approaches, and a review of their characterization, methods, merits, and shortcomings is found in [Schafer and Wang, 2020; Reyes Fernandez de Bulnes *et al.*, 2020]. Most methods explore the available directives in a common framework, considering the compilation process as a black box, as illustrated in Figure 1, where the values for each directive (configuration), on the left, lead to an implementation with different hardware metrics (design), on the right.

Configurations are encoded as arrays of attributions to each directive (e.g. `loop unrolling = true/false`, `loop pipeline = true/false`, `number of adders=3`, `number of multipliers=2`, `function_inline = true/false`). Designs are encoded as arrays of hardware metrics (e.g. hardware resource usage (`area`) and number of clock cycles (`latency`)).

The relationships between configurations and designs (highlighted in blue) are complex and unpredictable [Schafer and Wang, 2020], limiting the accuracy of exploration methods and forcing them to explore more designs to achieve high-quality results [Ferretti *et al.*, 2018b].

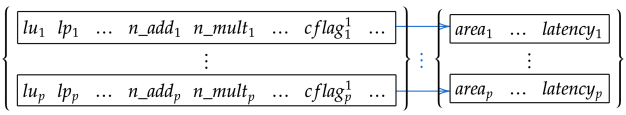

Figure 1. A framework commonly used for DSE. A configuration is an array with the attributions for the HLS directives, such as loop unrolling (`lu`), loop pipeline (lp), number of arithmetic functions (e.g. `n_add` and `n_mult`), and compiler flags (`c_flags`). The generated design is considered as a function of the configuration and is represented as an array with the hardware metrics, such as `area` and `latency`.

Aiming to mitigate the unpredictability of such a relationship, [Schafer, 2016; Schafer and Wang, 2020] propose grouping HLS directives according to their application scope. Each group (called knob) is explored with a different DSE method, and exploration results of a knob are used as input to the next knob's exploration. As such, instead of a fully combinatorial design space, groups of smaller spaces are explored methodologically.

This paper extends the idea of exploring knobs incrementally to a finer granularity by selecting two important directives, which are known for their tight relationship and impacts on the hardware metrics. The first is nFUs, which defines how many functional units (e.g., memory ports, adders, and multipliers) are available to the HLS compiler, hence controlling the balance between hardware area and computation time. The second is the "loop pipeline", which schedules loop operations to the available FUs interleaving loop iterations, maximizing their usage and reducing latency. As such, we have a situation in which the hardware metrics are highly dependent on the combination of these two directives, and by analysing their dependency and relationship, we propose an efficient DSE method for these two directives.

The contributions of this paper are:

- We present a study on the limitations of traditional DSE methods, given the impact of considering all directives in a common framework.
- We highlight the limitations of traditional DSE methods regarding the nFUs and loop pipeline directives' exploration.
- We propose new DSE methods for exploring the nFUs and loop pipeline directives, demonstrating the improvements in exploration speed and hardware quality.

The rest of this paper is organized as follows: Section 2 presents the related works. Section 3 presents the background and definitions. Section 4 evaluates the impact of traditional DSE approaches that consider directives in a common framework. Section 5 presents the analyses and improvements regarding the exploration of nFUs and loop pipeline. Section 6 presents the improved results in speed and accuracy. Finally, Section 7 concludes the paper.

2 Related Works

This section briefly introduces DSE approaches that treat all directives in a common framework. To handle the shortcoming of such an approach, these methods use elaborated estimators, e.g., Machine Learning (ML), usually requiring a large amount of synthesized design samples and target the DSE of a broad number of directives or complex systems. Since the methods presented in this paper specifically target the nFUs and loop pipeline directives, they are better placed as an auxiliary method for a partial DSE or an initial sampling for the methods presented in this section aiming to reduce the number of full synthesis performed.

Early approaches in the literature propose performing the DSE using Genetic Algorithm (GA), which evolves (refines) configurations stochastically to optimise the hardware metrics, focusing on SoCs [Palesi and Givargis, 2002] and MPSoCs [Pimentel, 2017]. On the HLS context, [Krishnan and Katkoori, 2006] encodes the application's data-path and the number of available processing units within a GA, performing the directives DSE concurrently to the scheduling and allocation, capturing their inter-relationships in the optimization process.

Evolving from single-accelerator to complex systems, recent works focus on hierarchical architectures composed of hardware-accelerated components, requiring a global DSE to be fully optimised. Targeting this hierarchical scope, [Siracusa *et al.*, 2021] uses a roof-line model to balance the exploration of system-wide directives (as global shared memory) while performing a traditional DSE for exploring the kernel-related ones; [Mahapatra and Schafer, 2019] focuses on optimising the communication between a master design and multiple-hardware accelerators by including communication-related directives in the DSE. [Bannwart Perina and Bonato, 2018] proposes a learning-based classification to decide whether the code should be accelerated using a Field-Programmable Gate Array (FPGA) or a Graphics Processing Unit (GPU).

Considering complex systems, [Liao *et al.*, 2023] presents a Pareto-based pruning method for reducing the number of explored designs in multi-component systems. The design space explored in these approaches contains parameters that control components integration (e.g., communication) in addition to the ones controlling their internal architecture, resulting in extremely large design spaces. Our solution focuses on loops, which are a snippet of code (likely) present within each component, being an alternative that could be adopted as an intermediate step for multi-component DSE exploration.

Further learning and bio-inspired methods focus on automatically discovering relationships between the configuration space and design space. For example, [Meng *et al.*, 2016] presents a learning-based method to explore the OpenCL directives that control high-level architecture of multiple FPGA accelerators. On the bio-inspired side, MOPSE [Mishra and Sengupta, 2014] presents a particle swarm-based DSE, and [Rajmohan and Ramasubramanian, 2020] presents a memetic DSE method which uses performance models to mitigate the costly synthesis time.

Recently, the use of ML methods for DSE has become popular, e.g., [Goswami *et al.*, 2023] synthesizes a set of configurations to build a predictive model, which is used for the DSE. Similarly, [Ferretti *et al.*, 2022] proposes to use graph neural networks trained on pre-synthesized accelerators, allowing for handling computer programs of arbitrary length. Also, in [Kwon and Carloni, 2020], transfer learning is used to mitigate the large amount of data required by machine learning-based methods, which is impractical to obtain given the time-consuming synthesis step involved. The approaches proposed in this paper can be used for generating optimized accelerators required for fitting ML approaches.

Regarding Reinforcement Learning (RL), [Nardi *et al.*, 2019] leverages random forests classifiers using a constrained Bayesian optimization for handling unknown feasibility constraints statistically and uses prior distributions for speeding-up the learning process. Furthermore, [Wu *et al.*, 2021, 2023] present a Graph Neural Network (GNN)-based framework composed of a hardware metrics estimator learned concurrently to the DSE. The *a priori* distribution estimations used for improving the learning of such methods can be created with lighter DSE approaches, such as the ones proposed in this paper.

Nevertheless, this paper demonstrates that relationships between directives should be better explored to obtain a more efficient DSE. A similar observation is done in [Belwal and Ramesh, 2021], where a regression-forest-based search iteratively refines the Pareto-front when exploring the loop unroll and pipeline directives.

3 Background and Definitions

HLS tools take as input a high-level description (e.g., `C` code) and compile it to a hardware-orientated representation, usually a Register Transfer Level (RTL) one, which can be implemented as an accelerator using FPGAs. The reconfigurable nature of FPGAs allows for controlling the resulting hardware architecture and its performance metrics by attributing values to the available directives. Performing a DSE is the process of finding the values for each directive (a configuration) in order to optimise the desired hardware metrics, which we formally describe next.

Let n be the number of available directives, and c_i a vector holding their attributions given by $c_i = \{[c_i^j] \mid c_i^j \leq a_j \mid \forall\, j \in \{1, \ldots, n\}\}$, where a_j is the maximum value for directive j. The configuration space is the set of all possible c_i combinations, formally $C = \{c_i \mid \forall\, i = \{1, \ldots, p\}\}$, where $p = \prod_{j=1}^{n} a_j$ is the number of possible combinations. Two configurations c_x and c_y are said to be neighbours if they differ by $1u$ (one unit) in one directive, i.e. $||c_x - c_y|| = 1u$.

For each c_i, the source code compilation results in a hardware design, with the respective hardware metrics represented by the vector $d_i = \{[d_i^j] \mid \forall\, j \in \{1, \ldots, m\}\}$, where m is the number of metrics considered. The design space set $D = \{d_i \mid \forall\, i \in \{1, \ldots, p\}\}$ contains all designs d_i for each possible configuration c_i. Commonly used hardware metrics are the hardware resources usage (e.g., Adaptive Logic Modules (ALMs), Digital Signal Processings (DSPs), and Block Random Access Memorys (BRAMs)), energy consumption and latency.

Since the DSE aims to find configurations with the best trade-offs between the hardware metrics, we assume, without loss of generality, that the objective is to minimize all metrics. Given two configurations c_i and c_j, their respective d_i and d_j, c_i is said to dominate c_j if, and only if, $\{d_i^k \leq d_j^k \mid \forall\, k \in \{1, \ldots, m\}\}$. The designs with best trade-offs are the ones which are not dominated by any other design, forming a Pareto-optimal front.

4 Configuration and Design Spaces Relationship

The accuracy of DSE methods (defined here as "how close the selected points are to the actual Pareto-optimal front") is known for being limited by the complex and unpredictable effects of the directives in designs [Schafer and Wang, 2020; Ferretti *et al.*, 2018b]. From now on, this limiting factor will be referred to as the "C and D relationship".

In this paper, we explore the idea that treating all directives in a common framework from traditional DSE approaches creates the complex C and D relationship. Even though the ability to treat different directives in a common framework is desirable for its practicality, the C and D relationship is expected to be complex given that:

1. Different directives act on different scopes. E.g., loop unroll targets a specific loop, while nFUs affects the whole design.
2. Locally optimal directives are not optimal for the whole code. For example, the optimal nFUs configuration for a given loop may differ from the optimal values considering other loops together.
3. Directives scope may intersect. E.g., memory architecture optimizations are code-wide, affecting any loop that references the target memories.
4. Directives are non-orthogonal, meaning that the same

design can be obtained with two different sets of directives.
5. The directives application order impacts the final design. E.g., applying loop interchange before unrolling and vice-versa.
6. Neighbor configurations in C do not necessarily lead to neighbors in D, creating a "false neighborhood relationship". E.g., two configurations which differ only in the loop pipeline turned "on" and "off" are neighbours, but their designs differ significantly in speed and area [de Souza Rosa *et al.*, 2018b];

The following sections present three DSE methods that cover the most common approaches based on gradient, probability, and Lattice methods. The section highlights their shortcomings when considering the C and D relationship, resulting in the information used to improve the proposed exploration of the nFUs and loop pipeline directives described in Section 5.

4.1 Gradient-Based Approaches

Gradient-based approaches are greedy heuristics that, given a configuration c_i, compute (through compilation or estimation) the hardware metrics d_i for c_i and all its neighbours. Then, the next exploration point is defined by the neighbour with steepest gradient, and the search iterates [Prost-Boucle *et al.*, 2014; da Silva and Bampi, 2015].

This class of approaches is suitable when all points in the design space obey a regular area-speed trade-off, i.e., faster designs use more area and vice versa, which is not always valid for general HLS, as directives may impact both metrics positively or not, especially when considering loop structures.

Equation (1) defines the gradient, where δa and δl are the area and latency differences between the current design d_c and d_v^i for a configuration c_c and it's neighbours $c_v^i \mid i \in \{1, \ldots, n\}$. Note that the gradient is typically approximated by considering only one direction at a time; hence, the number of neighbours is exactly equal to the number of available directives n.

$$\nabla G \propto \left[\frac{\delta l}{\delta a}\right] \tag{1}$$

Making DSE decisions based on ∇G has two shortcomings. First, the false neighbourhood relations presented in Section 4 lead to erroneous high-gradient values. Second, Equation 1 raises the following scenarios for the possible δa and δl results:

1. $\delta a = 0$ **and** $\delta l = 0$: $\nabla G = \frac{0}{0}$;
2. $\delta a = 0$ **and** $\delta l > 0$: $\nabla G = \infty$, and d_v^i dominates d_c;
3. $\delta a = 0$ **and** $\delta l < 0$: $\nabla G = \infty$, and d_c dominates d_v^i;
4. $\delta a > 0$ **and** $\delta l = 0$: $\nabla G = 0$, and d_v^i dominates d_c;
5. $\delta a < 0$ **and** $\delta l = 0$: $\nabla G = 0$, and d_c dominates d_v^i;
6. $\delta a > 0$ **and** $\delta l > 0$: $\nabla G > 0$, and d_v^i dominates d_c;
7. $\delta a < 0$ **and** $\delta l < 0$: $\nabla G > 0$, and d_c dominates d_v^i;
8. $\frac{\delta a}{|\delta a|} = -\frac{\delta l}{|\delta l|}$: $\nabla G < 0$, d_v^i and d_c trade-off area and speed.

For scenario 1, ∇G is undefined. When comparing scenarios 2 against 3, 4 against 5, and 6 against 7, ∇G cannot differ between the dominant and dominated designs, leading to inconsistent exploration decisions. Scenario 8 represents a trade-off between area and speed, indicating that the configurations are Pareto-points candidates and should be further explored. However, since $\nabla G < 0$, the exploration selects only one configuration.

The approach presented in [Xydis *et al.*, 2015] reduces these inconsistencies using the area-time product signal model $s[d_c]$ defined in Equation (2), whose minimization tends to the Pareto-optimal curve direction, but does not imply in finding all Pareto points.

$$s[d_c] = Area(d_c) \times Latency(d_c) \tag{2}$$

Equation (3) rewrites $s[d_v^i]$ as a function of d_c, allowing for easily evaluating the $s[d_v^i]$'s behaviour in the 8 scenarios.

$$\begin{aligned} s[d_v^i] &= Area(d_v^i) \times Latency(d_v^i) \\ &= s[d_c] - Area(d_c)\delta l - Latency(d_c)\delta a + \delta a \delta l \end{aligned} \tag{3}$$

Here, $s[d_v^i]$ solves the inconsistencies in scenarios 1 to 5. In scenarios 6 and 7, $s[d_v^i]$ is consistent when assuming $Area(d_c) >> \delta a$ and $Latency(d_c) >> \delta l$, however, this assumption can be invalid, especially for loop pipelines [de Souza Rosa *et al.*, 2018a]. Furthermore, in scenario 8, which represents the finding of Pareto-point candidates, $s[d_v^i]$ can either increase or decrease, leading to inconsistencies in differentiating dominant from dominated designs.

4.2 Probabilistic Approach

The Probabilistic Multi-Knob (PMK) [Schafer, 2016], from Cyber compiler, proposes separating HLS directives into sets, called knobs, and using different methods to explore each knob. Later, [Schafer and Wang, 2020] discusses the impacts of each knob's directives on the resulting hardware design, separating the knobs according to their application scope as follows:

Local knob: Composed by local synthesis directives that affect a specific code region, controlling the final hardware micro-architecture. E.g., loop unroll and pipeline.

nFUs knob: The nFUs available and shared among all high-level code operations. E.g., the number of adders, multipliers, and memory ports. This knob controls the trade-off between hardware area and speed, significantly impacting loop pipelines [de Souza Rosa *et al.*, 2018a] and memory usage [Pilato *et al.*, 2017].

Global knob: Contains global synthesis options applied throughout the whole code. E.g., dead code elimination, tree balancing, and function inline [Zuo *et al.*, 2015].

PMK starts by exploring the local knob using an ant-colony optimization with the nFUs fixed at their minimum and maximum values c_{min} and c_{max}, respectively. The area between c_{min} and c_{max} is considered as its likelihood of containing Pareto-optimal designs (hence the probabilistic name), and larger areas are selected for further exploration.

The limited number of designs explored in this first step balances the trade-off between exploration speed and result quality.

Then, the nFUs knob is explored by synthesizing 10 equally distributed configurations between c_{min} and c_{max} to avoid local-optima and plateaus. However, the uniform distribution might not be adequate to larger codes, leading to a small diversity in the Pareto-front [Reyes Fernandez de Bulnes *et al*., 2020].

The global knob is not explored in [Schafer, 2016], and further research focuses on solving such problems and speeding up the DSE by using multi-scale searches [Jun *et al*., 2023].

4.3 Lattice-Based Approach

Algorithm 1 presents the Lattice-Based DSE [Ferretti *et al*., 2018b], the first approach to consider the relationships between C and D in its development. It starts by using an U-distribution to select m random points from the C (line 1), which are synthesized (line 2). This distribution makes configurations with extreme values more likely to be chosen.

In the iterative part, Pareto-optimal configurations among the evaluated designs are selected for further exploration (line 3). Each configuration has its σ-neighbours synthesized (lines 5 to 6). The selection and synthesis process iterates until no new Pareto-optimal designs are found or a budget is exceeded (lines 10 to 12). Finally, all compiled configurations and metrics are returned (lines 1, 2, 8, and 9).

Algorithm 1: Lattice-Based DSE algorithm presented in [Ferretti *et al*., 2018b].

input : list of possible configurations, maximum number of design evaluations (*budget*)
output list of configurations and metrics
:

1 $evalC \leftarrow m$ U-distributed random configurations c;
2 $evalD \leftarrow$ synthesize and annotate area and latency;
3 $nextConfigs \leftarrow pareto(evalD)$;
4 **do**
5 $\quad configs \leftarrow \sigma$-neighbours of $nextConfigs$;
6 $\quad designs \leftarrow$ synthesize and annotate area and latency $\forall\ c_i \in configs$;
7 $\quad nextConfigs \leftarrow pareto(evalD \cup designs)$;
8 $\quad$ append $configs$ to $evalC$;
9 $\quad$ append $designs$ to $evalD$;
10 $\quad$ **if** $size(evalC) \geq budget$ **then**
11 $\quad\quad$ break;
12 **while** $nextConfigs \neq \emptyset$;
13 return $compiledConfigs$ and $compileDesigns$;

The parameters m and σ in Algorithm 1 control the trade-off between exploration time and results quality, and must be adjusted for the target compiler, directive set, and high-level code.

There are two shortcomings of this approach. The first lies within its U-shaped distribution initialization, which tends to create either fast-and-large or slow-and-small designs, making them susceptible to local minima and plateaus [Rosa, 2019]. The second lies in treating all directives in a common manner, limiting its exploration capabilities, as discussed in Section 4.

5 Number of Functional Units and Loop Pipeline Exploration

Given the observations in Section 4, this section presents the proposed incremental DSE method for exploring the nFUs and loop pipeline directives, highlighting the achieved improvements.

As per motivations, we chose nFUs since it controls the balance between computation time and hardware resources usage, and it is arguably the major contributor to the large exploration design spaces on HLS, typically at thousands of combinations, while other directives usually have few options each. Loop pipeline has been chosen since it is crucial to implement instruction-level parallelism on FPGAs, significantly improving the hardware throughput and energy efficiency, and it is found in the vast majority of DSE works. Furthermore, both directives are known for their tight interaction, as loop pipelining is meant to optimise nFUs usage.

Notice that a pipelined loop requires hardware resources to store and route data, creating an overhead which depends on the nFUs, invalidating the idea of "fewer functional units lead to smaller designs" [de Souza Rosa *et al*., 2018a]. Furthermore, designs with the same nFUs and loop pipeline turned "on" and "off" cannot be considered neighbours.

The main idea in the proposed DSE method is exploring nFUs individually first, incorporating the observations from Sections 4.1, 4.2, and 4.3, as we describe in Section 5.1. Then, we leverage the nFUs exploration results to analyse the relationships between nFUs and loop pipeline, leading to novel exploration flow for both directives, described in Section 5.2.

5.1 Path-Based Number of Functional Units Exploration

When considering the nFUs directives in isolation, the C and D relationship has a more predictable behaviour than when considering more directives simultaneously. As such, we propose a neighbourhood-based path search between the configurations with maximum and minimum nFUs (c_{max} and c_{min}).

Algorithm 2 presents proposed Path-Based approach, which iteratively evaluates configurations focusing solely nFUs. First, c_{max}, and c_{min} constants are defined, δ_{10} is defined as 10% of the interval between c_{max} and c_{min}, along with memory structures to annotate which designs were compiled and discarded (lines 1 to 8), and which are the configurations to be searched next (line 5). The hardware metrics can be obtained through full synthesis or estimation methods (lines 6 and line 3 of Algorithm 4). The iterative part (lines 9 to 24) starts by popping the search pool to get the next exploration, and its neighbours are obtained using Algorithm 3 (line 12), which returns neighbour configurations that were not explored or discarded (more details next).

Algorithm 4 (line 14) iteratively compiles neighbour configurations, adding only Pareto-dominant points among the

current design and its neighbours (i.e. calculating the Pareto frontier [Ščap *et al.*, 2013] in line 18) to the search pool, guaranteeing that only the most promising points are evaluated. If no neighbours are found (local optimum and plateaus), a new configuration is created by reducing 10% of all nFUs, and the algorithm iterates until a configuration that has not been compiled or discarded yet is found, or c_{min} is reached (lines 19 to 23).

Algorithm 2: Proposed Path-based DSE.

```
   input  : data-Flow Graph (DFG)
   output : nFUs and evaluated points metrics
   :
1  c_max ← maximum resources;
2  c_min ← minimum resources;
3  δ_10 ← ⌈0.1 * (c_max − c_min)⌉;
4  c_cur ← c_max;
5  searchPool ← c_cur;
6  d_cur ← compile c_cur;
7  compiled ← [c_cur];
8  discarded ← ∅;
9  do
10    c_cur ← searchPool.pop();
11    d_cur ← metrics of c_cur;
12    neighbors ←
        getNeighbors(c_cur, compiled, discarded) ;
13    if neighbors ≠ ∅ then
14       N_c, N_d ←
           compileNeighbors(d_cur, neighbors);
15       append N_c to compiled;
16       append N_d to discarded;
17       Insert N_c into the searchPool;
      /* remove dominated designs          */
18    searchPool ← pareto(searchPool ∪ c_cur);
      /* apply the 10% rule                */
19    if searchPool == ∅ && c_min ∉ compiled then
20       c_new ← max(c_cur − δ_10, c_min);
21       while c_new ∈ compiled || c_new ∈
           discarded && c_new ≠ c_min do
22          c_new ← max(c_new − δ_10, c_min);
23       Insert c_new into the searchPool;
24 while searchPool ≠ ∅;
```

Algorithm 3 computes the given configuration's neighbours by decrementing a single directive's value from the input configuration (line 2 to 6), avoiding their reduction to less than c_{min} (line 4), returning only non-compiled and non-discarded neighbours (lines 5 and 6).

Finally, Algorithm 4 compiles all neighbours of a given design, evaluating one configuration at a time (lines 2 to 8). If a neighbour dominates the current design or results in the same hardware metrics (line 6), the neighbours evaluation stops, returning only non-compiled designs (line 8).

This approach integrates the strengths of gradient-based, probabilistic, and lattice-based methods while mitigating their weaknesses. By limiting the search to nFUs only and using the Pareto-dominance explicitly, instead of the gradient or area-time signal, we avoid the shortcomings of gradient-based methods. At the same time, our path-based search avoids PMK's low-diversity problems since it is not limited to a fixed number of designs between c_{max} and c_{min}. Opposing the Lattice-Based approach, it can escape possible local minima and plateaus since it forces a connection between c_{max} and c_{min}.

Nevertheless, the path-based search does not have parameters to balance the number of explored designs and the quality of results. Such a mechanism can be implemented by varying the number of neighbours explored per design in Algorithm 3; thus, it will not be considered a shortcoming of the proposed algorithm.

Algorithm 3: $getNeighbors()$ function used by Algorithm 2.

```
   input  : design configuration c_cur, compiled and
            discarded lists.
   output : neighbours configurations rSet.
   :
1  rSet ← ∅;
2  foreach directive j ∈ c_cur do
3     c_new ← c_cur;
4     c_new(j) ← max(c_new(j) − 1, c_min(j));
5     if c_new ∉ compiled && c_new ∉ discarded
        then
6        rSet.insert(c_new);
```

Algorithm 4: $compileNeighbors()$ function used by Algorithm 2.

```
   input  : metrics a design d_cur, its neighbors list.
   output : list of compiled (compNeigh) and
   :
            discarded (discNeigh) neighbours.
1  compNeigh ← ∅;
2  foreach c_i ∈ neighbors do
3     d_i ← compile c_i;
4     calculate da and dt from d_i and d_cur;
5     add c_i to compNeigh;
6     if da ≥ 0 && dt ≥ 0 then
7        break;
8  discNeigh ← neighbors/compNeigh;
```

5.2 Loop Pipeline Exploration Based on the nFUs Exploration

With a DSE method in place to explore the nFUs, the next step is to consider the loop pipeline directive in the DSE. To do so, we first explicitly analyse how the loop pipeline directive affects the *C* and *D* relationship, using the observations to elaborate a novel exploration flow which improves the speed and quality of results when considering nFUs and loop pipeline combined.

The proposed flow divides the configuration space into two subsets containing all possible combinations of nFUs, one with the loop pipeline directive turned "off", and another with it turned "on". Then, one subset is explored, and the exploration results are used as *a priori* knowledge for improv-

ing the second subset's exploration utilising an Lattice-based DSE extension as a Pareto-refinement too.

5.2.1 Lattice-Based DSE Extension

The idea behind this extension is to use a set of configurations to initialise the search instead of the U-shaped distribution in Algorithm 1. This extension, referred as "Seeded Lattice-Based DSE", is achieved with the following modifications in Algorithm 1:

- Add an input configurations set ($seeds$);
- Modify line 1 to "$configs \leftarrow seeds$;".

First, this extension can be used as a DSE method, and using samples from a U-shaped distribution results in the original Lattice-based approach. Second, if the seeds are a Pareto-optimal approximation, this extension works as a Pareto refinement step, which guarantees achieving a Pareto-optimal approximation at least as good as the input one. Third, the search will converges faster for input configurations closer to the Pareto-optimal ones.

Finally, even though the proposed Seeded Lattice-Based approach can be applied for an arbitrary set of directives, we use it mostly as a refinement toll for exploring the nFUs and loop pipeline, and hence we limit our evaluations to those directives.

5.2.2 Proposed Exploration Flow

Let B and A be the subsets of C considering the nFUs with loop pipeline turned `off` (no pipeline) and `on` (fully pipelined with minimum initiation interval), with design metrics b_i and a_i, respectively. A traditional exploration flow (Figure 1) is equivalent to consider the configuration space as ($C = B \cup A$), and then explore it using any DSE method.

The proposed DSE flow, depicted in Figure 2, separates the nFUs and loop pipeline exploration, reducing the number of evaluated designs without compromising the exploration accuracy as consequence of avoiding the complex C-D relationship.

First, a DSE is performed in B, and its Pareto-front are used as seeds to explore A with the seeded lattice-based method, which is expected to speed-up the lattice search conversion given that Pareto-optimal points in B and A are likely to be close w.r.t. the nFUs. Finally, the Pareto-front of both explorations is selected as a result.

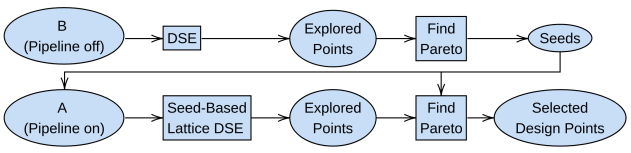

Figure 2. Proposed DSE seeding-based flow for nFUs and loop pipeline directives. The design space is divided into two subsets: loop pipeline "on" (A) and "off" (B). B is explored with any DSE method of choice, and its resulting Pareto-front approximation is used to quick-start the exploration of A using the Seeded Lattice-Based DSE.

The proposed flow is based on two suppositions **s1** and **s2**: **s1 -** given two configurations b_i and b_j, if b_i is faster (or slower) than b_j in B, than the respective a_i is likely to be faster (or slower) than a_j in A as well. **s2 -** configurations with a smaller area in B are likely to have a smaller area in A as well.

The reasoning behind **s1** is the fact that a faster configuration has shorter paths in its Data-Flow Graph (DFG), which tends to result in pipelines with smaller II values [de Souza Rosa *et al.*, 2021]. The reasoning behind **s2** is that the area difference between two designs in B reflects the different nFUs, and it should also be observed in A unless the hardware overheads are larger than the resources used by the FUs.

If **s1** and **s2** were an implication instead of a likelihood, the Pareto-optimal designs from A would also be optimal in B, and exploring only one of them would be sufficient. Nevertheless, we can use the Pareto-front of A (or B) to approximate its counterpart's Pareto-front, and the approximation's quality increases with the suppositions' likelihood to hold, also leading to higher exploration speed-ups.

Table 1 presents the suppositions' validity frequency over the benchmarks used in Section 6 considering the hardware metrics $latency$, $ALMs$, and both concurrently. Considering the $latency$ results in a probability estimation for **s1**; As an "overview" of **s2**, we consider the $ALMs$ usage when compiling the codes with hardcore DSPs disabled, avoiding their influence in the $ALMs$ and Registers.

Table 1. Frequency that designs keep their relative position for design spaces without (B) and with (A) loop pipeline for $latency$, ALMs, and `both`, indicating how often suppositions **s1**, **s2**, and both are likely to hold. Benchmark names abbreviations: Multipliers (**mt**), Adder Chain (**ac**), Dividers (**dv**), Float Adder Tree (**fat**), and Complex (**cp**).

Benchmark	mt	ac	dv	fat	cp
$latency$	1.000	0.930	0.923	0.932	0.897
$ALMs$	0.837	0.897	0.674	0.622	0.448
both	0.837	0.885	0.621	0.572	0.381

Table 1 shows that for the tested benchmarks, at least 89.7% of the designs keep their relative position regarding design speed, corroborating **s1**. Table 1 also shows that **s2** is less likely to hold (as little as 44.8%), which is a consequence of increasing overhead necessary for creating loop pipelines with decreasing nFUs.

As a final remark, the proposed approaches are domain agnostic, being more impactful for applications which highly depend on pipelining for achieving a high throughput and can explore instruction level parallelism with multiple FUs, such as signal processing or streaming applications.

6 Results

This section presents the exploration improvements proposed in Sections 5.1 and 5.2 regarding their accuracy and speed gains. All directives apart from nFUs and loop pipeline are fixed to their default values for comparing approaches directly.

Sections 6.1 to 6.4 describe the experimental setup, nomenclature, evaluated metrics, and benchmarks. Sections 6.5 and 6.6 present results for the nFUs exploration (Section

5.1) and when both the nFUs and loop pipeline directives are considered (Section 5.2), respectively.

6.1 Experimental Setup

The results were obtained on a Ubuntu 14.04 computer with 16 GB of RAM and an Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz, using LegUP 4.0 as HLS compiler and Gurobi 7.5 as Solver. Loop pipelines were created using [Oppermann *et al.*, 2016] with a 10 minutes time budget. All results are the average of 50 repetitions.

The designs were synthesized using Quartus II 16.1, targeting a Stratix V board. We consider as hardware metrics the `latency`, `ALMs`, `registers` and `DSPs`, noting that other hardware metrics can be regarded without loss of generality.

We highlight that no further HLS annotations than the ones for specifying the nFUs and loop pipeline are necessary for the proposed DSE methods, which are implemented along with scripts for generating the configuration files using GNU Octave.

6.2 Nomenclature

For briefness, table 2 summarizes the nomenclature to facilitate the DSE methods identification and the configuration space used for testing. E.g., a test using the lattice-based DSE to explore nFUs and loop pipeline together is written as "$\mathcal{L}^{B \cup A}$"; a test using the proposed approach to explore nFUs and loop pipeline (Figure 2) using the proposed path-based DSE to generate the seeds is written as "$\mathcal{S}^{A}(\mathcal{P}^{B})$".

Table 2. Summary of nomenclature used to identify the exploration methods and configuration spaces.

Short Name	**DSE Method**
$\mathcal{K}$	PMK [Schafer, 2016] (Section 4.2)
$\mathcal{L}$	Lattice-based [Ferretti *et al.*, 2018b] (Section 4.3)
$\mathcal{P}$	Path-based (Section 5.1)
$\mathcal{S}$	Seeded lattice (Section 5.2.1)
Design Space	**Description**
B	Formed by nFUs with loop pipeline "off"
A	Formed by nFUs with loop pipeline "on"
$B \bigcup A$	Formed by nFUs and loop pipeline
Notation	**Description**
$\mathcal{S}(X)$	DSE method X is used to create the seeds for $\mathcal{S}$
X^{Y}	DSE method X is used to explore the design space Y

6.3 Evaluation Metric

We consider the exploration speed proportional to the number of evaluated designs, given that the compilation and synthesis process vastly dominates the computation time.

The exploration's accuracy is measured using the Average Distance from Reference Set (ADRS), which is the most common DSE quality indicator [Reyes Fernandez de Bulnes *et al.*, 2020]. The ADRS compares two sets, a reference one Γ and an approximation one Ω, according to Equation (4). The set $\Gamma = \{\gamma_1, \ldots, \gamma_p\}$ is formed by the Pareto-optimal points of an exhaustive DSE over the design space, while $\Omega = \{\omega_1, \ldots, \omega_q\}$ is formed by the Pareto-optimal estimation obtained by the different exploration methods or flows. The ADRS is measured only between unique points in the reference and approximated sets to avoid its reduction due to repeated points.

$$ADRS(\Gamma, \Omega) = \frac{1}{|\Gamma|} \sum_{\gamma \in \Gamma} \min_{\omega \in \Omega} \left[\max_{j=\{1,\ldots,m\}} \left(\frac{\omega^j - \gamma^j}{\gamma^j} \right) \right] \quad (4)$$

The ADRS is computed over the latency, ALMs, registers and DSPs, using their average as the final ADRS value. As baseline, we consider a sufficiently accurate exploration when achieving ADRS $< 1\%$, which is considered a small value according to [Schafer, 2016].

6.4 Benchmark Selection

The proposed approaches focus on the synthesis of loop structures. Hence, for fairness, the benchmark selection contains codes solely composed of loops, avoiding the bias created by code outside the loop structures in the results. Furthermore, unrolling the nested loops in more complex applications negatively impacts loop pipelining, which can inflate the hardware metrics results [de Souza Rosa *et al.*, 2021]. The adopted benchmarks are the same ones as in [Canis *et al.*, 2014; de Souza Rosa *et al.*, 2018b], which are composed of challenging loops for creating pipelines and that are strongly impacted by the configurations [de Souza Rosa *et al.*, 2018a], making the DSE a compulsory step to achieve optimal hardware implementations [Rosa, 2019].

Table 3 presents design space size formed by considering only the nFUs for each benchmark, which is computed by multiplying the maximum values of all FUs, and when loop pipeline is also considered, which is the former's double since the pipeline directive can assume 2 values ("on" and "off"). Among the codes, **ac** contains a chain of adders, **fat** is a tree of floating-point adders, **cp** contains a diversity of arithmetic operations, and **mt** and **dv** focus on creating and sharing multipliers and dividers.

Table 3. Selected benchmarks design space size formed by only the number of nFUs and when loop pipeline is also considered.

Acronym	**mt**	**ac**	**dv**	**fat**	**cp**
$\lvert C \rvert_{nFUs}$	30	48	128	168	3920
$\lvert C \rvert_{nFUs+pipeline}$	60	96	256	336	7840

We highlight that a full synthesis takes approximately 40 minutes in our setup, resulting in a total time between 2.5 and 217.8 days for an exhaustive search. It is worth noting that the design space cardinality and amount of designs that need to

be synthesized for obtaining a satisfactory Pareto-front estimation depend on the number of possible configurations (and not on the source code complexity), and that the design space size does not reflect the source code complexity.

6.5 Results Optimizing nFUs

Table 4 presents the number of synthesized configurations for $\mathcal{K}$, $\mathcal{P}$, $\mathcal{L}$, and $\mathcal{S}$ for loop pipelining "off" (B) and "on" (A). The (m, σ) parameters for $\mathcal{L}$ and $\mathcal{S}$ are fixed at $(0.05, 0.25)$ (see Section 4.3).

For exploring the nFUs, we found $(m, \sigma) = (0.05, 0.25)$ are adequate for achieving ADRS $< 1\%$, while the experiments presented in [Ferretti *et al.*, 2018b,a] use $(m, \sigma) \geq (0.2, 0.5)$ to achieve the same accuracy. The larger values required in [Ferretti *et al.*, 2018b,a] imply a higher number of compiled configurations, which is a consequence of the more complex design space given that more directives are considered together.

Table 4 shows that the proposed $\mathcal{P}$ evaluates up to 2.3× (geomean) fewer designs and achieves a smaller ADRS than $\mathcal{L}$. Furthermore, $\mathcal{S}$ successfully improves the ADRS compared to $\mathcal{P}$, at the cost of evaluating more designs, demonstrating its usage as a Pareto refinement tool.

The higher ADRS for $\mathcal{L}$ w.r.t. $\mathcal{P}$ is a consequence of $\mathcal{L}$ not being designed to efficiently escape local optimum and plateaus created by its initial U-distribution, while the proposed exploration avoids such problem.

Note that $\mathcal{P}$ is constructed to evaluate more designs than $\mathcal{K}$, but less than $\mathcal{L}$, while improving the ADRS w.r.t. both, and $\mathcal{S}$ improves the ADRS by exploring more designs than $\mathcal{P}$. Overall, both $\mathcal{P}$ and $\mathcal{S}$ achieve a lower ADRS than $\mathcal{L}$ while synthesizing less designs. Finally, $\mathcal{K}$ is not suitable for larger benchmarks, as demonstrated by the ADRS for **fat** and **cp**.

6.6 Optimizing nFU and Loop Pipeline

Table 5 presents the number of compiled designs and ADRS when loop pipeline and nFUs are explored together ($B \cup A$), and with the proposed exploration flow (Figure 2).

Table 5 shows that the proposed flow reduces the number of evaluated designs and the ADRS in all cases when compared against exploring $B \cup A$ with a traditional flow, demonstrating the expected improvements.

Note that $\mathcal{S}^A(\mathcal{P}^B)$ increases the number of evaluated designs when compared to $\mathcal{P}^{B \cup A}$ for **fat**, but this extra exploration results in a smaller ADRS. These results emphasize the adverse effects of combining directives for exploration, especially on $\mathcal{P}$, which is designed solely for exploring the nFUs.

Besides that, the reduction in the number of explored designs grows with the configuration space size (see Table 1), what we speculate is caused by the proposed exploration flow leveraging the fact that supposition **s1** holds frequently.

Note that $\mathcal{L}^{B \cup A}$ achieves ADRS $> 1\%$ with $(m, \sigma) = (0.05, 0.25)$, which were adequate values to achieve $ADRS < 1\%$ while exploring only the nFUs (Section 6.5). Hence, to achieve ADRS $< 1\%$, (m, σ) should be increased, demonstrating that the combination of two directives (nFUs and loop pipeline) forces traditional approaches to increase the exploration to maintain the accuracy as a consequence of the C and D relationship.

It is important to note that the proposed flow always uses $\mathcal{S}$ to explore A; as such, it is sensitive to the (m, σ) parameters. As such, Figure 3 presents the number of evaluated designs by ADRS for the same explorations presented in Table 5 when varying (m, σ), demonstrating the exploration methods' robustness to these parameters. Increasing (m, σ) means to increase the number of explored designs and, consequently, to reduce the ADRS.

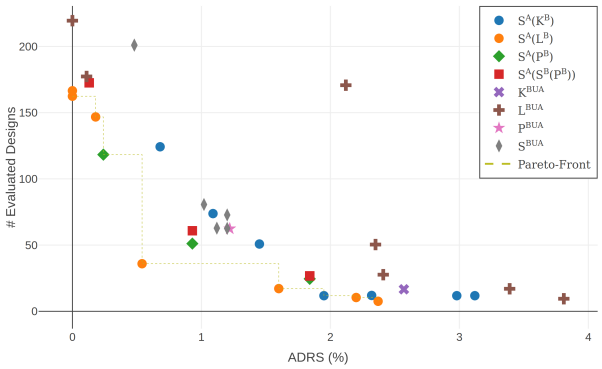

Figure 3. Number of evaluated designs by ADRS for the methods presented in Table 5 (geomean) and $(m, \sigma) = \{(0.01, 0.05), (0.025, 0.075), (0.05, 0.1), (0.05, 0.25), (0.1, 0.5), (0.15, 0.75), (0.2, 0.1)\}$. As m and σ increase, the number of evaluated designs increases, while the ADRS decreases.

Figure 3 shows that the designs obtained with the proposed flow dominate the ones obtained with the traditional flow, demonstrating consistent improvements independently of the (m, σ) tuning and underlying nFUs exploration method. The consistency indicates that the measures taken in exploring the nFUs and loop pipeline mitigate the problems encountered by state-of-the-art approaches, which consider all directives concurrently and the HLS process as a black-box approach.

7 Conclusion

During the high-level synthesis, the design space exploration is crucial in achieving efficient hardware designs. Traditional DSE approaches are based on commonly treating all directives, analysing only the inputs and outputs of the systems. However, this paradigm complicates the relationships between inputs (configuration space) and outputs (design space), limiting the existing approaches' prediction capabilities and performance.

In this paper, we carefully evaluate the shortcomings of state-of-the-art approaches when exploring the nFUs and loop pipeline directives and propose a novel, improved approach for exploring these directives.

For nFUs exploration, we propose a path-based DSE approach (Section 5.1) that improves the exploration by avoiding inconsistencies and local traps identified on different traditional methods. Results show that the proposed approaches speed up the nFUs exploration up to 2.82× while keeping the ADRS smaller than 1%.

For nFUs and loop pipeline directives, we proposed a novel exploration flow based on the relationship between points in

Table 4. Number of compiled configurations and ADRS obtained by $\mathcal{P}$, $\mathcal{L}$, and $\mathcal{S}$ over spaces B and A independently. $(m, \sigma) = (0.05, 0.25)$.

	$\mathcal{K}^B$	$\mathcal{L}^B$	$\mathcal{P}^B$	$\mathcal{S}^B(\mathcal{P}^B)$	$\mathcal{K}^A$	$\mathcal{L}^A$	$\mathcal{P}^A$	$\mathcal{S}^A(\mathcal{P}^A)$
				# designs				
mt	5	6.00	12	12	5	6.00	11	11
ac	9	46	35	52	9	34.22	30	55
dv	8	31.92	30	32	8	33.30	27	32
fat	11	137.88	36	114	11	101.16	40	111
geo.	**8.31**	**63.39**	**27.55**	**44.77**	**8.31**	**60.59**	**30.50**	**59.67**
				ADRS (%)				
mt	0.99	0.99	0	0	0.99	0.63	0	0
ac	0	0	0	0	0.12	0	0	0
dv	0.69	0.011	0.095	0.095	0.42	0.38	0.41	0.41
fat	4.34	0.023	0.24	0.22	2.71	0.14	0.20	0.18
cp	4.09	0.0036	0.16	0.15	5.37	0.031	0.30	0.11
geo.	**2.04**	**0.21**	**0.10**	**0.09**	**1.94**	**0.24**	**0.18**	**0.14**

Table 5. Number of compiled designs and ADRS obtained by DSE methods to explore the nFUs and loop pipeline directives. Results present the traditional and proposed flows using the DSE methods $\mathcal{P}$, $\mathcal{L}$, and $\mathcal{S}$. $(m, \sigma) = (0.05, 0.25)$.

	Proposed flow (Section 5.2.2)				Traditional Flow			
	$\mathcal{S}^A(\mathcal{K}^B)$	$\mathcal{S}^A(\mathcal{L}^B)$	$\mathcal{S}^A(\mathcal{P}^B)$	$\mathcal{S}^A(\mathcal{S}^B(\mathcal{P}^B))$	$\mathcal{K}^{B\cup A}$	$\mathcal{L}^{B\cup A}$	$\mathcal{P}^{B\cup A}$	$\mathcal{S}^{B\cup A}(\mathcal{P}^{B\cup A})$
				# designs				
mt	8	8	12	12	10	8	21	11
ac	22.34	43	34	44	18	64.9	59	65
dv	8	11.98	21	24	16	17.96	28	24
fat	15.47	134.28	130	184	22	81.5	33	75
cp	10.3	474.22	313	357	20	1231.42	344	1369
geo.	**11.79**	**48.28**	**51.11**	**60.82**	**16.62**	**62.26**	**52.37**	**70.66**
				ADRS (%)				
mt	0.65	2.65	0.90	0.90	0.89	1.32	0.00	0.00
ac	0.00	0.00	0.00	0.00	0.76	0.00	0.08	0.00
dv	2.43	0.02	2.31	2.31	3.64	0.06	2.64	2.64
fat	2.74	0.00	1.40	1.40	2.83	9.97	2.38	2.38
cp	3.89	0.00	0.02	0.02	4.69	0.01	0.96	0.02
geo.	**1.95**	**0.54**	**0.93**	**0.93**	**2.57**	**2.35**	**1.22**	**1.02**

the design space. Results show that the proposed flow consistently improves the DSE speed-accuracy trade-off, forming a new Pareto-front when contrasted against traditional exploration flow.

Declarations

Acknowledgements

The authors would like to thank the São Paulo Research Foundation (FAPESP) and the European Union's Next-Generation EU (FAIR) - Future Artificial Intelligence Research project.

Funding

FAPESP - grant number 2014/14918-2. FAIR - Piano Nazionale di Ripresa e Resilienza (PNRR) - Missione 4 Componente 2, Investimento 1.3 - D.D. 1555 11/10/2022, PE00000013. This manuscript reflects only the authors' views and opinions, neither the European Union nor the European Commission can be considered responsible for them.

Authors' Contributions

LSR contributed to the conception, implementation, and experiments and is the main author. VB and CSB guided the methodology development. All authors contributed to the writing of this manuscript. All authors read and approved the final manuscript.

Competing interests

The authors declare that there are no competing interests.

Availability of data and materials

The manuscript does not provide complementary materials.

References

Ali, K., Ben Atitallah, R., Ait El Cadi, A., Fakhfakh, N., and Dekeyser, J.-L. (2019). ViPar: High-Level Design Space Exploration for Parallel Video Processing Architectures. *International Journal of Reconfigurable Computing*, 2019. DOI: 10.1155/2019/4298013.

Bannwart Perina, A., Becker, J., and Bonato, V. (2019). Lina: Timing-Constrained High-Level Synthesis Performance Estimator for Fast DSE. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 343–346. DOI: 10.1109/ICFPT47387.2019.00063.

Bannwart Perina, A. and Bonato, V. (2018). Mapping Estimator for OpenCL Heterogeneous Accelerators. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 294–297. DOI: 10.1109/FPT.2018.00057.

Belwal, M. and Ramesh, T. (2021). Q-PIR: A quantile based Pareto iterative refinement approach for high-level synthesis. *Engineering Science and Technology, an International Journal*, page 101078. DOI: 10.1016/j.jestch.2021.11.004.

Canis, A., Brown, S. D., and Anderson, J. H. (2014). Modulo SDC scheduling with recurrence minimization in high-level synthesis. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. DOI: 10.1109/FPL.2014.6927490.

Castro-Godínez, J., Mateus-Vargas, J., Shafique, M., and Henkel, J. (2020). AxHLS: design space exploration and high-level synthesis of approximate accelerators using approximate functional units and analytical models. In *Proceedings of the 39th International Conference on Computer-Aided Design*, ICCAD '20, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/3400302.3415732.

Cong, J., Wei, P., Yu, C. H., and Zhou, P. (2017). Bandwidth Optimization Through On-Chip Memory Restructuring for HLS. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, pages 43:1–43:6, New York, NY, USA. ACM. DOI: 10.1145/3061639.3062208.

da Silva, J. S. and Bampi, S. (2015). Area-oriented iterative method for Design Space Exploration with High-Level Synthesis. In *Circuits & Systems (LASCAS), 2015 IEEE 6th Latin American Symposium on*, pages 1–4. IEEE, IEEE. DOI: 10.1109/LASCAS.2015.7250447.

de Souza Rosa, L., Bonato, V., and Bouganis, C.-S. (2018a). Scaling Up Loop Pipelining for High-Level Synthesis: A Non-iterative Approach. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 62–69. DOI: 10.1109/FPT.2018.00020.

de Souza Rosa, L., Bouganis, C. S., and Bonato, V. (2018b). Scaling Up Modulo Scheduling For High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1. DOI: 10.1109/TCAD.2018.2834440.

de Souza Rosa, L., Bouganis, C.-S., and Bonato, V. (2021). Non-iterative SDC modulo scheduling for high-level synthesis. *Microprocessors and Microsystems*, 86:104334. DOI: 10.1016/j.micpro.2021.104334.

Fernando, S., Wijtvliet, M., Nugteren, C., Kumar, A., and Corporaal, H. (2015). *(AS)2: Accelerator Synthesis Using Algorithmic Skeletons for Rapid Design Space Exploration*, pages 305–308. DATE '15. EDA Consortium, San Jose, CA, USA. DOI: 10.5555/2755753.2755821.

Ferretti, L., Ansaloni, G., and Pozzi, L. (2018a). Cluster-Based Heuristic for High Level Synthesis Design Space Exploration. In *IEEE Transactions on Emerging Topics in Computing*, volume PP, pages 1–1. IEEE. DOI: 10.1109/TETC.2018.2794068.

Ferretti, L., Ansaloni, G., and Pozzi, L. (2018b). Lattice-Traversing Design Space Exploration for High Level Synthesis. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 210–217. IEEE. DOI: 10.1109/ICCD.2018.00040.

Ferretti, L., Cini, A., Zacharopoulos, G., Alippi, C., and Pozzi, L. (2022). Graph Neural Networks for High-Level Synthesis Design Space Exploration. *ACM Trans. Des. Autom. Electron. Syst.*, 28(2). DOI: 10.1145/3570925.

Ferretti, L., Kwon, J., Ansaloni, G., Guglielmo, G. D., Carloni, L. P., and Pozzi, L. (2020). Leveraging Prior Knowledge for Effective Design-Space Exploration in High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3736–3747. DOI: 10.1109/TCAD.2020.3012750.

Goswami, P., Schaefer, B. C., and Bhatia, D. (2023). Machine learning based fast and accurate High Level Synthesis design space exploration: From graph to synthesis. *Integration*, 88:116–124. DOI: 10.1016/j.vlsi.2022.09.006.

Jun, H., Ye, H., Jeong, H., and Chen, D. (2023). AutoScaleDSE: A Scalable Design Space Exploration Engine for High-Level Synthesis. *ACM Trans. Reconfigurable Technol. Syst.*, 16(3). DOI: 10.1145/3572959.

Kachris, C. and Soudris, D. (2016). A survey on reconfigurable accelerators for cloud computing. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–10. DOI: 10.1109/FPL.2016.7577381.

Krishnan, V. and Katkoori, S. (2006). A genetic algorithm for the design space exploration of datapaths during high-level synthesis. *IEEE Transactions on Evolutionary Computation*, 10(3):213–229. DOI: 10.1109/TEVC.2005.860764.

Kwon, J. and Carloni, L. P. (2020). Transfer Learning for Design-Space Exploration with High-Level Synthesis. In *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*, MLCAD '20, page 163–168, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/3380446.3430636.

Liao, Y., Adegbija, T., and Lysecky, R. (2023). Efficient system-level design space exploration for high-level synthesis using pareto-optimal subspace pruning. In *Proceedings of the 28th Asia and South Pacific Design Automa-*

tion Conference, ASPDAC '23, page 567–572, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/3566097.3567841.

Mahapatra, A. and Schafer, B. C. (2019). Optimizing RTL to C Abstraction Methodologies to Improve HLS Design Space Exploration. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. DOI: 10.1109/ISCAS.2019.8702355.

Meng, P., Althoff, A., Gautier, Q., and Kastner, R. (2016). *Adaptive threshold non-pareto elimination: re-thinking machine learning for system level design space exploration on FPGAs*, page 918–923. DATE '16. EDA Consortium, San Jose, CA, USA. DOI: 10.3850/9783981537079$_0$350.

Mishra, V. K. and Sengupta, A. (2014). MO-PSE: Adaptive multi-objective particle swarm optimization based design space exploration in architectural synthesis for application specific processor design. *Advances in Engineering Software*, 67:111–124. DOI: 10.1016/j.advengsoft.2013.09.001.

Nane, R., Sima, V.-M., Pilato, C., Choi, J., Fort, B., Canis, A., Chen, Y. T., Hsiao, H., Brown, S., Ferrandi, F., *et al.* (2016). A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35:1591–1604. DOI: 10.1109/TCAD.2015.2513673.

Nardi, L., Koeplinger, D., and Olukotun, K. (2019). Practical design space exploration. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 347–358. DOI: 10.1109/MASCOTS.2019.00045.

Oppermann, J., Koch, A., Reuter-Oppermann, M., and Sinnen, O. (2016). ILP-based Modulo Scheduling for High-level Synthesis. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '16, pages 1:1–1:10, New York, NY, USA. ACM. DOI: 10.1145/2968455.2968512.

Palesi, M. and Givargis, T. (2002). Multi-objective design space exploration using genetic algorithms. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, CODES '02, page 67–72, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/774789.774804.

Perina, A. B., Becker, J., and Bonato, V. (2019). ProfCounter: Line-Level Cycle Counter for Xilinx OpenCL High-Level Synthesis. In *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 618–621. DOI: 10.1109/ICECS46596.2019.8964669.

Perina, A. B., Silitonga, A., Becker, J., and Bonato, V. (2021). Fast Resource and Timing Aware Design Optimisation for High-Level Synthesis. *IEEE Transactions on Computers*, 70(12):2070–2082. DOI: 10.1109/TC.2021.3112260.

Pilato, C., Mantovani, P., Di Guglielmo, G., and Carloni, L. P. (2017). System-Level Optimization of Accelerator Local Memory for Heterogeneous Systems-on-Chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(3):435–448. DOI: 10.1109/TCAD.2016.2611506.

Pimentel, A. D. (2017). Exploring exploration: A tutorial introduction to embedded systems design space exploration. *IEEE Design & Test*, 34(1):77–90. DOI: 10.1109/MDAT.2016.2626445.

Prost-Boucle, A., Muller, O., and Rousseau, F. (2014). Fast and standalone Design Space Exploration for High-Level Synthesis under resource constraints. *Journal of Systems Architect*, 60(1):79–93. DOI: http://dx.doi.org/10.1016/j.sysarc.2013.10.002.

Rajmohan, S. and Ramasubramanian, N. (2020). A Memetic Algorithm-Based Design Space Exploration for Datapath Resource Allocation During High-Level Synthesis. *Journal of Circuits, Systems and Computers*, 29(01):2050001. DOI: 10.1142/S0218126620500012.

Reyes Fernandez de Bulnes, D., Maldonado, Y., Trujillo, L., and Acacio Sanchez, M. E. (2020). Development of Multi-objective High-Level Synthesis for FPGAs. *Sci. Program.*, 2020. DOI: 10.1155/2020/7095048.

Rosa, L. d. S. (2019). Fast Code Exploration for Pipeline Processing in FPGA Accelerators. DOI: 10.11606/T.55.2019.tde-21082019-143417.

Samanta, A., Hatai, I., and Mal, A. K. (2024). A survey on hardware accelerator design of deep learning for edge devices. *Wireless Personal Communications*, 137(3):1715–1760. DOI: 10.1007/s11277-024-11443-2.

Schafer, B. C. (2016). Probabilistic Multiknob High-Level Synthesis Design Space Exploration Acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(3):394–406. DOI: 10.1109/TCAD.2015.2472007.

Schafer, B. C. and Wang, Z. (2020). High-Level Synthesis Design Space Exploration: Past, Present, and Future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2628–2639. DOI: 10.1109/TCAD.2019.2943570.

Siracusa, M., Delsozzo, E., Rabozzi, M., Di Tucci, L., Williams, S., Sciuto, D., and Santambrogio, M. D. (2021). A Comprehensive Methodology to Optimize FPGA Designs via the Roofline Model. *IEEE Transactions on Computers*, pages 1–1. DOI: 10.1109/TC.2021.3111761.

Wang, Z., Chen, J., and Schafer, B. C. (2020). Efficient and Robust High-Level Synthesis Design Space Exploration through offline Micro-kernels Pre-characterization. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 145–150. DOI: 10.23919/DATE48585.2020.9116309.

Wang, Z. and Schafer, B. C. (2022). Learning from the Past: Efficient High-level Synthesis Design Space Exploration for FPGAs. *ACM Trans. Des. Autom. Electron. Syst.*, 27(4). DOI: 10.1145/3495531.

Wu, N., Xie, Y., and Hao, C. (2021). Ironman: Gnn-assisted design space exploration in high-level synthesis via reinforcement learning. In *Proceedings of the 2021 Great Lakes Symposium on VLSI*, GLSVLSI '21, page 39–44, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/3453688.3461495.

Wu, N., Xie, Y., and Hao, C. (2023). Ironman-pro: Multiobjective design space exploration in hls via reinforcement learning and graph neural network-based modeling. *IEEE Transactions on Computer-Aided Design of*

Integrated Circuits and Systems, 42(3):900–913. DOI: 10.1109/TCAD.2022.3185540.

Xydis, S., Palermo, G., Zaccaria, V., and Silvano, C. (2015). SPIRIT: Spectral-Aware Pareto Iterative Refinement Optimization for Supervised High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34:155–159. DOI: 10.1109/TCAD.2014.2363392.

Zhong, G., Prakash, A., Wang, S., Liang, Y., Mitra, T., and Niar, S. (2017). Design Space exploration of FPGA-based accelerators with multi-level parallelism. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 1141–1146. DOI: 10.23919/DATE.2017.7927161.

Zuo, W., Kemmerer, W., Lim, J. B., Pouchet, L.-N., Ayupov, A., Kim, T., Han, K., and Chen, D. (2015). A polyhedral-based SystemC modeling and generation framework for effective low-power design space exploration. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 357–364. DOI: 10.1109/ICCAD.2015.7372592.

Ščap, D., Hoić, M., and Jokić, A. (2013). Determination of the Pareto frontier for multiobjective optimization problem. *Transactions of FAMENA*, 37(2):15–28. Available at: `https://hrcak.srce.hr/105312`.