Data-driven HLS optimization for reconfigurable accelerators

Aggelos Ferikoglou, Andreas Kakolyris, Vasilis Kypriotis, Dimosthenis Masouros,
Dimitrios Soudris and Sotirios Xydis
National Technical University of Athens, Greece
{aferik,akakolyris,vkypriotis,demo.masouros,dsoudris,sxydis}@microlab.ntua.gr

ABSTRACT

High-Level Synthesis (HLS) has played a pivotal role in making FPGAs accessible to a broader audience by facilitating high-level device programming and rapid microarchitecture customization through the use of directives. However, manually selecting the right directives can be a formidable challenge for programmers lacking a hardware background. This paper introduces an ultra-fast, knowledge-based HLS design optimization method that automatically extracts and applies the most promising directive configurations to the original source code. This optimization approach is entirely data-driven, offering a generalized HLS tuning solution without reliance on Quality of Result (QoR) models or meta-heuristics. We design, implement, and evaluate our methodology using over 100 applications sourced from well-established benchmark suites and GitHub repositories, all running on a Xilinx ZCU104 FPGA. The results are promising, including an average geometric mean speedup of ×7.2 and ×1.35 compared to designer-optimized designs and resource over-provisioning strategies, respectively. Additionally, it demonstrates a high design feasibility score and maintains an average inference latency of 38ms. Comparative analysis with traditional genetic algorithm-based Design Space Exploration (DSE) methods and State-of-the-Art (SoA) approaches reveals that it produces designs of similar quality but at speeds 2-3 orders of magnitude faster. This suggests that it is a highly promising solution for ultra-fast and automated HLS optimization.

CCS CONCEPTS

• **Hardware → Hardware-software codesign**; **Electronic design automation**; **High-level and register-transfer level synthesis**.

KEYWORDS

High-Level Synthesis (HLS), Design Space Exploration (DSE), FPGA Accelerators, Auto-tuning, Data-driven Optimization

ACM Reference Format:
Aggelos Ferikoglou, Andreas Kakolyris, Vasilis Kypriotis, Dimosthenis Masouros, and Dimitrios Soudris and Sotirios Xydis. 2024. Data-driven HLS optimization for reconfigurable accelerators. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3658471

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06.
https://doi.org/10.1145/3649329.3658471

1 INTRODUCTION

While modern edge-cloud architectures expand computing capacity through a vast pool of resources, the inefficiency of traditional CPUs in delivering near real-time execution has driven the adoption of heterogeneous devices to meet the demand for increased computational power. Field Programmable Gate Array (FPGA) devices are claiming a significant share of the "heterogeneous computing" landscape. The acquisitions of Altera by Intel and Xilinx by AMD underscore the ever-growing interest in re-configurable platforms. For instance, Microsoft has adopted CPU-FPGA systems in its data center to help accelerate the Bing search engine [11]. This trend is also reshaping public Cloud offerings, e.g., AWS already offers VM instances equipped with FPGAs to provide hardware acceleration capabilities to end-users.

HLS has democratized FPGAs by enabling high-level device programming and rapid microarchitecture customization, removing the need for complex and potentially error-prone hardware design techniques that utilize specialized hardware description languages [10]. HLS facilitates development by providing a quicker and more adaptable process, allowing developers to guide the HLS compiler in kernel synthesis by adding directives to high-level language source code. Traditional algorithmic HLS tools primarily focus on extracting parallelism from algorithmic descriptions and transforming the result into hardware execution that operates in parallel [12]. As a result, HLS tools empower designers to quickly implement various algorithmic choices, identify high-level tradeoffs concerning area and performance, and avoid early optimizations [16].

Despite HLS providing a higher level of abstraction than hardware description languages, developers still need to master the translation of software/compiler constructs into hardware concepts to optimize the acceleration of applications using FPGAs [14]. For users transitioning from the "cloud world" this challenge becomes even more formidable as they often operate at higher levels of the technological stack and lack familiarity with the specifics of hardware design. Indeed, automatic exploration of an extended compiler and micro-architectural design space is a constant requirement in modern HLS frameworks [14, 21]. On the other hand, selecting manually the suitable HLS directives poses an exceptionally challenging task, even for experienced designers. This complexity arises primarily from *i)* the huge decision design space and *ii)* the inherent correlation of directives with the underlying architecture [10]. As a result, the absence of tools that quickly and automatically provide optimized HLS directive configurations, poses a substantial challenge for harnessing FPGA acceleration by designers with limited experience in the field.

In recent years, numerous studies have delved into various methodologies for conducting DSE of HLS directives [16]. These works primarily fall into two categories: *a)* synthesis-based and *b)* model-based approaches. **Synthesis-based** methods [14, 17, 20]

aim to efficiently navigate the configuration decision space by assessing a set of QoR metrics derived from designs synthesized using HLS tools. **Model-based** approaches [7, 18, 22] build predictive models to estimate QoR metrics instead of executing HLS flows. Especially, Graph Neural Networks (GNN) [6, 15] have gained prominence in performance modeling due to their capacity to capture variations in source code, encompassing size, structural constructs, and optimization directives. Once performance and resource utilization estimates are available, the DSE process can be streamlined and addressed through dedicated heuristics [4]. **Knowledge-based** approaches [5] have also emerged to reduce the decision design space and minimize the costly synthesis evaluations for unknown applications by leveraging knowledge from previously explored ones. However, we've observed that current research solutions face significant challenges when it comes to providing highly efficient designs quickly, making them less accessible to non-experienced designers.

In our view, HLS DSE methodologies should evolve based on three major pillars, i.e., *i)* ***efficiency:*** proposed directives should be close to near-optimal configurations; *ii)* ***fast convergence:*** optimization recommendations should be provided in acceptable time frames; and *iii)* ***interpretability and generalization:*** methodologies should be analyzable and generic enough and combine collaborative knowledge from multiple applications, thus, allowing developers to use them off-the-shelf, even for potentially unknown applications. While efficiency is tackled by prior research works [14, 15], these solutions typically focus on application specific optimizations, thus requiring massive numbers of design evaluations to guide exploration, which, in turn, translate to high operational costs and longer time to market. Static analysis of the source-code characteristics can aid towards fast convergence and generalization, by providing an "execution profile" of an application, without requiring long-running evaluations. Even though latest research [19] is pivoting to this direction, they are restricted to pattern matching approaches [5], thus, relying on a bi-directional knowledge flow between applications, neglecting the potential benefits of collaborative information sharing.

This unexploited opportunity raises the possibility of developing optimization approaches that leverage collective knowledge to quickly and automatically provide optimized designs. The open source movement also assists to this direction, as it has increased the number of HLS repositories available online (e.g., GitHub), making it extremely easy to obtain and analyze designs freely. Methodologies that move in this direction bring the community one step closer to the vision of completely removing the programmer from the optimization cycle.

In this direction, we introduce a cluster-based hierarchical design optimization framework for fast auto-tuning of HLS directives. Unlike existing auto-tuning techniques [14, 15], we present a fully data-driven HLS optimization framework that combines existing knowledge/data from previously characterized applications to extract and apply the most promising directive configurations for unknown applications. Our optimization technique correlates an application's high-level source code characteristics with its low-level hardware Quality-of-Result (QoR) metrics to create programmer agnostic HLS optimizations. We integrate our approach into Vitis-HLS, providing an end-to-end online and zero-delay auto-tuning solution for high-quality FPGA designs. Our approach achieves *i)* an average geometric mean speedup of ×7.2 and ×1.35 compared to designer optimized designs and over-provisioning respectively, *ii)* high design feasibility score, and *iii)* an average directive proposal time of 38msec. Comparisons with traditional genetic algorithm-based DSE, as well as state-of-the-art approaches, reveal that it generates designs of comparable quality 2-3 orders of magnitude faster, suggesting that our approach holds great promise as an ultra-fast and automated solution for HLS optimization.

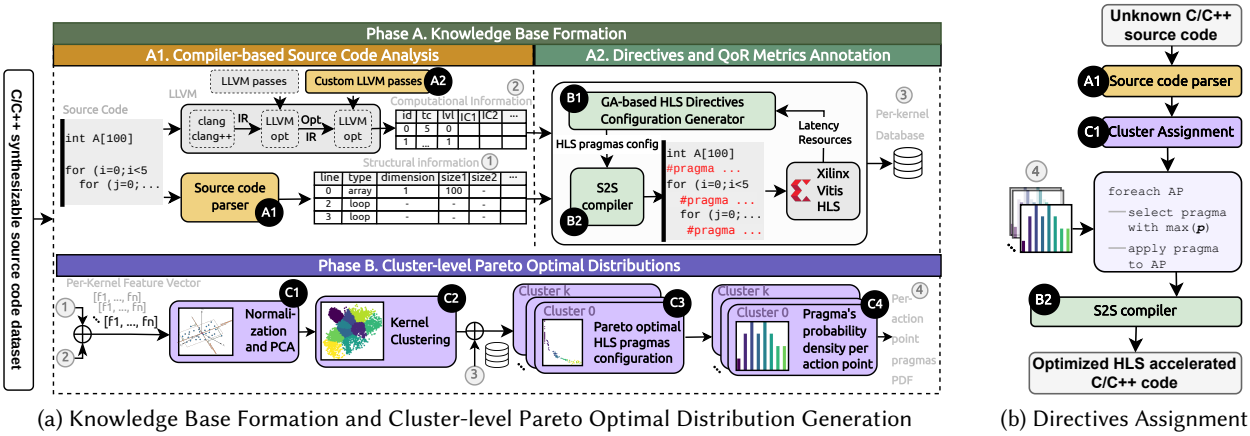
2 A CLUSTER-BASED HIERARCHICAL HLS OPTIMIZATION FRAMEWORK

Our automates the identification and application of optimized directives for HLS designs. Its primary goal is to reduce the latency of proposed designs while ensuring they meet the resource constraints of the target FPGA. This framework adopts a data-driven methodology, gathering and combining insights from previously analyzed applications. Consequently, the optimizations it suggests are derived straight from historical data, rather than relying on complex heuristics or QoR models. Figure 1 provides an overview of the proposed HLS design optimization approach, outlining the framework's architectural structure. This framework involves three phases, which we thoroughly examine in the upcoming sections.

Phase A. Knowledge Base Formation

A1. Compiler-based source code analysis: In this phase, we characterize the applications that constitute our knowledge base. We map these applications into fixed-size feature vectors by capturing high-level characteristics directly from their source code. Each feature vector combines both structural elements (e.g., array dimension sizes) and computational information (e.g., loop trip-counts). The framework identifies all potential "action points" within these knowledge base applications. An "action points" signifies a specific location within the source code where a `pipeline`, `unroll`, or `array partition` HLS directive can be applied. To analyze source code and pinpoint these action points (a.k.a "optimization points"), we employ various components of the LLVM toolchain. A source code parser A1 navigates the Abstract Syntax Tree (AST) generated from the code, identifying loops and array declarations, and an additional set of custom LLVM passes A2 aids in understanding application loop organization (e.g., nesting structures).

Source-code level feature vectors: The information derived from the source code by LLVM serves as the basis for generating a feature vector that characterizes each application. Although modern compiler toolchains provide comprehensive information on memory and data dependencies, our emphasis remains on features that effectively capture similar source code structures found in HLS applications, with a particular focus on in-memory arrays and loop structures. Focusing on array and loop code structures aligns with prior research [5], as these elements have a substantial impact on the performance of HLS applications. A feature vector is structured into three components: *a)* the array structure, *b)* the loop structure, and *c)* the LLVM arithmetic and memory operations. This feature vector's structure aligns with the current knowledge base, which entails determining *a)* the maximum number of array dimensions, *b)* the maximum number of arrays within an application, *c)* the



(a) Knowledge Base Formation and Cluster-level Pareto Optimal Distribution Generation (b) Directives Assignment

Figure 1: Overview of the proposed optimization methodology

maximum number of loop structures in an application, *d)* the maximum nesting level within a loop structure, and *e)* the maximum number of loops at each nesting level. When handling arrays with fewer dimensions than the maximum allowed, a specific identifier is used to denote the absence of the corresponding dimension. Likewise, the same approach is applied to applications lacking the maximum number of arrays, loops, and nesting levels. For unseen applications that deviate from our representation, any additional action points are disregarded during the compile-time directives assignment phase. During the knowledge base formation phase, we simply incorporate the new source code into our knowledge base and make corresponding adjustments to our representation, without the need for re-training, as outlined in *Phase A2*.

A2. Directives and QoR metrics annotation: In this phase, we focus on data collection, specifically the acquisition of QoR metrics for applications subjected to different HLS directives. This knowledge forms the foundation of our collaborative approach, as the proposition of directives in *Phase C* relies entirely on the statistical analysis of this data. While *Phase A2* outlines our strategy for collecting QoR and directives-related information from our applications, it's worth noting that the knowledge base can alternatively be constructed by gathering hand-optimized synthesized designs. After pinpointing the action points in each kernel during *Phase A1*, we apply various directives to each action point and carry out synthesis to obtain latency (measured in clock cycles) and area metrics (BRAM%, DSP%, FF%, and LUT%) for the resulting design.

Given the vast design space, we employ a genetic optimization approach B1 to efficiently explore the configuration space of directives and ensure convergence to optimal solutions. It tackles a dual optimization problem i.e., minimizing both design latency and area while considering the resource constraints of the target FPGA. This methodology yields a set of Pareto Optimal solutions for each application, each with distinct directives for some or all action points, representing a trade-off between performance and area. To traverse the solution space, we utilize the NSGA-II algorithm [3], known for its ability to bypass local optima and quickly converge to efficient solutions. NSGA-II is an evolutionary algorithm that operates across several generations, transferring elite configurations from one generation to the next. In our scenario, each generation entails the following steps: *a)* initializing the configuration population, *b)* applying each configuration from the current population to the source code via a source-to-source compiler B2, and synthesizing the output for the target FPGA using Xilinx Vitis, and *c)* feeding the synthesis results into NSGA-II to generate the next generation of configurations. If a specific directive configuration results in a design that exceeds available resources or requires an impractical amount of time for synthesis (e.g., 1 hour), it is flagged as infeasible. Overall, we set the population size to 40 configurations and the total number of generations to 24.

Overhead of assembling the knowledge base: Given the fact that the designs of one generation are evaluated in parallel, the 1h limit for time-consuming synthesis, and the termination criterion, database formation requires 24h for 960 evaluations in the worst case for a single application. While this process is time-consuming, it occurs only once in our solution (or asynchronously in general) and thus does not form a bottleneck for our approach. Moreover, in production level environments, the proposed architecture enables the incremental expansion of the knowledge base in a seamless manner, without the need for ground-up training or employment of sophisticated retraining techniques typically required in ML-driven approaches, which are prone to catastrophic forgetting and have proven to be ineffective in incrementally assimilating new information [8].

Phase B. Cluster-level Pareto Optimal Distributions

It adopts a cluster-based approach to propose and apply distinct directives for each action point. The underlying concept is that source codes sharing similar structures (indicated by the same feature vectors) tend to exhibit common characteristics and, consequently, respond similarly to the same directives, aligning with the optimization strategy proposed by Ferretti et al. [5]. We validate our hypothesis by examining the correlation between source code similarity and directive effectiveness using *Pearson's coefficient*, specifically considering the Pareto-optimal designs with the lowest latency. For every application, we establish two vectors: one

for the *Hamming distance* of the source code feature vector and another for the assigned directive's *Hamming distance* compared to all other applications. The *Pearson's correlation coefficient* for these vectors reveals an average correlation of 0.62, with a narrow standard deviation of 0.07, signifying a moderate yet existing positive relationship, with values closely grouped around the mean.

We employ the K-Means clustering algorithm C2, where the assignment of different applications to clusters is determined by their fixed-size feature vectors, as captured in *Phase A*. To speed up the convergence of the clustering algorithm, we normalize the feature vectors and reduce their dimensionality using the Principal Components Analysis (PCA) algorithm C1. Finally, to determine the optimal number of clusters, the framework leverages the widely recognized Elbow method [9]. This method involves performing clustering for various cluster numbers and computing the Within-Cluster Sum of Squares (WCSS) i.e., the summary of squared distances of each point and the cluster centroid.

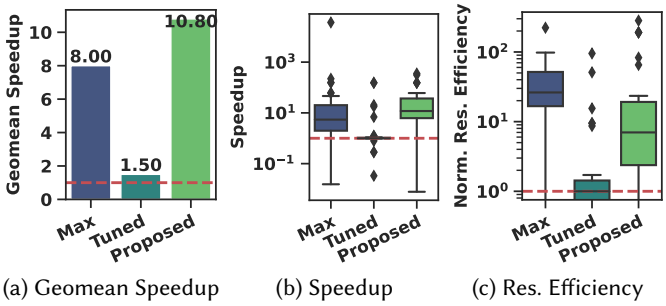
Cluster-wise directives distribution: Following the cluster assignment, it enriches each cluster with QoR and directives-related information. Specifically, for every application within a cluster, the framework collects the specific directives' configurations that led to a Pareto Optimal (PO) solution (*Phase A2*). These configurations may differ in terms of the directives applied per action point. Consequently, this process yields a distribution of "Pareto Optimal directives" per action point, encompassing all distinct directives from all PO solutions across all applications for that specific point C3. Drawing from this distribution, it derives a histogram of directives per action point, representing the likelihood of employing a particular directive at the designated source code location C4.

Phase C. Compile-time Directives Assignment

In the "compile-time directives assignment" phase, it optimizes incoming applications by automatically recommending directives for each identified action point. Figure 1b provides an overview of the steps involved. To begin, the framework takes as input the source code of the application to be optimized. As a first step, the source code parser extracts the essential information needed to characterize the application A1 and generates its feature vector, as outlined in *Phase A*. Subsequently, based on this feature vector, the application is assigned to a specific cluster C2, as formed in *Phase C*. Based on the cluster's histograms, it suggests the directive with the highest probability for each action point, given that the probability surpasses a predefined lower threshold. Notably, increasing the probability threshold results in fewer optimization directives and consequently more feasible designs. In contrast, not using optimizations can lead to sub-optimal performance. After experimenting with various values, we settled on a threshold of 0.15, which strikes an optimal balance between these considerations. Once the directives are proposed, the source-to-source compiler B2 integrates them into the input source code and generates the optimized output.

3 EXPERIMENTAL EVALUATION

To demonstrate and validate the viability of the proposed optimization methodology, we constructed a comprehensive dataset



(a) Geomean Speedup (b) Speedup (c) Res. Efficiency

Figure 2: Leave-One-Out Evaluation

of HLS applications. This dataset is composed of synthesizable applications written in C/C++, sourced from well-established HLS benchmark suites within the community such as Rodinia [2] and Machsuite [13], as well as publicly available HLS repositories on GitHub. Specifically, we analyze over 100 applications spanning diverse domains, including Digital Signal Processing (DSP), Genomics, Machine Learning (ML) and High-Performance Computing (HPC) applications. For our source code analysis, we utilized Clang from LLVM (v14.0.5), and for synthesizing applications within our dataset, we utilized Xilinx Vitis HLS (v2021.1). Our choice of base architecture was the Xilinx MPSoC ZCU104 FPGA, operating at a clock frequency of 300 MHz.

3.1 Leave-One-Out Analysis

In our evaluation, we adopt a Leave-One-Out scenario (LOT), where we exclude each application from the dataset during the knowledge base formation phase. Subsequently, we aim to determine the optimized configuration during the compile-time directives assignment for each omitted application. We compare the design produced by our optimization methodology with *i)* the design derived from the source code with directives, resulting in the maximum feasible use of resources (Max), and *ii)* the design obtained when the source code is fine-tuned by the developers of the benchmark suite, e.g. Machsuite (Tuned). The metric we use to compare baselines with our approach is the design latency speedup over Vitis-HLS. This is computed by dividing the latency of the design synthesized by Vitis-HLS without directives by the latency of the design generated by the respective optimization approach.

In Figure 2a, the geometric mean speedups of the optimization methods discussed over Vitis-HLS are depicted. The proposed approach achieves a geometric mean speedup of ×10.8, surpassing Tuned, which achieves ×1.5, indicating superiority over human-optimized designs. On the other hand, Max baseline results in a geometric mean speedup of ×8, emphasizing that utilizing maximum resources often leads to sub-optimal designs. Figures 2b and 2c provide further insight into the achieved speedup distribution and the normalized average resources utilization. Our approach outperforms Max baseline in terms of design latency speedup, as indicated by the ×1.35 higher geometric mean speedup, utilizing ×3.75 fewer resources, demonstrating its ability to effectively harness the available resources of the target device. Regarding the Tuned baseline, our approach leads to a ×7.2 higher geometric mean speedup while utilizing ×6.1 more resources on average, highlighting that the

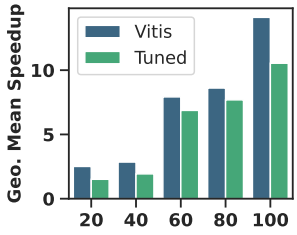


Figure 3: Knowledge Base Scaling

performance gains counteract the increased resources utilization. Given its focus on loop pipelining, loop unrolling, and array partitioning directives, it demonstrates its greatest advantages when applied to highly parallel applications that feature numerous loops and in-memory arrays. Specifically, applications with substantial parallelization potential, such as Machsuite's MD KNN and GEMM NCUBED, experience remarkable reductions in latency when directives from our methodology are incorporated. This results in impressive results, achieving speedups of ×310.5 and ×355 for MD KNN and GEMM NCUBED, respectively.

Design feasibility and compile-time directives assignment delay: The configurations recommended respect the available resources of the examined FPGA architecture, with approximately 86% of the optimized design configurations deemed feasible in terms of resources. This level of feasibility is comparable with the 98% and 93% feasibility rates achieved by the *Vitis* and *Tuned* baseline when employing standard optimizations. Another crucial aspect of our optimization methodology is its rapidity in optimizing an application's source code. In terms of evaluating the delay associated with the compile-time directives assignment, we observed a very narrow delay distribution, with a maximum/minimum inference latency of 50/20 ms (averaging 38 ms). This suggests that our approach can accelerate the process of DSE.

Knowledge Base Scaling: The suggested method is inherently data-driven, meaning that the knowledge base, comprising the used applications, directly influences its ability to provide efficient designs. Varied knowledge base sizes result in diverse clustering patterns of application source codes, consequently yielding differing Pareto-optimal distributions per cluster and distinct directive proposals. To explore the impact of knowledge base size, we systematically revisit the LOT experimental campaign, considering scenarios encompassing 20%, 40%, 60%, 80%, and 100% of the available applications' solution spaces. Figure 3 displays the geometric mean speedup of our approach in comparison to the Vitis and Tuned baseline across the examined knowledge base sizes. The results reveal that the higher the percentage of the knowledge base employed, the greater the geometric mean speedup. This implies that the performance of our approach could be improved by integrating more applications. For instance, expanding the knowledge base from 40% to 60% results in a ×5 and ×5.1 increase in geometric mean speedup for the Vitis and Tuned baselines, respectively. Further increasing the knowledge base by an additional 20% has a more modest effect on the geometric mean speedup, with only a 10% increase for both baselines. This outcome primarily stems from the inclusion of applications that augment the Pareto-optimal points within the distribution of each source cluster, resulting in more precise distributions and, thus, improved directive proposals.

Comparison with Per-Application DSE: We additionally compare our approach with a standard per-application Design Space Exploration (DSE) approach, specifically one that employs the NSGA-II optimizer (GenOpt) to generate approximate optimal Pareto designs for each application within the LOT scenario. The NSGA-II configuration is detailed in section 2. Figure 4 depicts the distributions of design latency speedup, calculated by dividing GenOpt's design latency by our method's design latency, and inference latency speedup, obtained by dividing GenOpt's time for directive proposal and kernel synthesis by the corresponding time from our approach. In Figure 4 (Left), the distribution indicates a maximum/minimum design latency speedup of ×1.2/×0.55, with an average design latency speedup reduction of 19%. Figure 4 (Right) displays a distribution with a maximum/minimum inference latency speedup of ×1710/×13.5, and an average inference latency speedup of ×425. As a result, our proposed optimization scheme yields a latency profile closely resembling that of the near-optimal design, all without the need for time-consuming meta-heuristic investigations.

3.2 Comparison with State-of-the-Art

Comparison with Knowledge-based DSE: Our approach falls into the category of knowledge-based methods within the SoA. To demonstrate the effectiveness of our approach, we compare with Ferretti's [5] knowledge-based HLS optimization (KBOpt), which, to the best of our knowledge, is the most closely related approach in the SoA. We conduct a comparison between KBOpt and our method for each application in the LOT scenario. KBOpt follows a series of steps when optimizing an application. As a first step, it identifies the most similar application in terms of source code features, calculating the *Hamming distance* with other pre-explored applications in *Phase A2*. For the identified application, it retrieves the Pareto-optimal designs and extracts their directives. It translates these directives, similar to the approach in [5], and applies them to the source code of the application being optimized. Finally, the method synthesizes all the different optimized source codes and identifies the designs with the lowest latency. Figure 5 depicts the distributions of design latency speedup and inference latency speedup, calculated as described in the per-application DSE comparison. In Figure 5 (Left), the distribution shows a maximum/minimum design latency speedup of ×60.6/×0.5, with an average design latency speedup of ×4.2. Figure 5 (Right) illustrates a distribution with a maximum/minimum inference latency speedup of ×440.3/×0.4, and an average inference latency speedup of ×99.5. It not only offers much faster directive proposals without the need for synthesizing multiple designs but also consistently delivers better design quality on average. This underscores the superiority of collaborative information sharing over the bidirectional knowledge flow between applications.

Comparison with Synthesis-based DSE: In the final part of our evaluation, we conduct a qualitative comparison[1] with AutoDSE [14], an automated framework built on top of the Merlin compiler [1]. AutoDSE and our approach share the same optimization objective: minimizing application latency while adhering to the resource

[1]While [14] provides an open-source GitHub repository, it is not actively maintained and several build issues arise during installation.

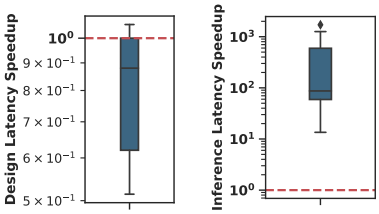


Figure 4: NSGA-II Exploration

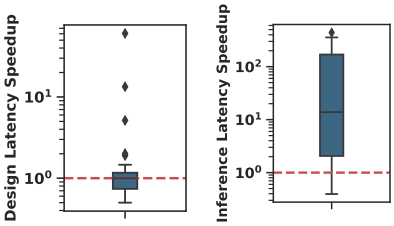


Figure 5: Knowledge-based DSE [5]

Application	AutoDSE	Proposed
GEMM	x100	x107.9
SPMV	x10	x16.8
STENCIL2D	x3	x10.4
KMEANS	x150	x66.6
KNN	x12.5	x100

Table 1: AutoDSE [14]

constraints of the target FPGA. Utilizing data from the original AutoDSE paper [14], we qualitatively discuss the performance of both approaches on common kernels from Rodinia [2] and Machsuite [13]. To facilitate a speedup comparison between AutoDSE and our approach, we normalize our results to Merlin's performance without directives. Specifically, the authors of [14] demonstrated that the Merlin compiler achieves an average speedup of ×3.29 over the default Vitis. To align our speedup results with the Merlin-related, we project them by dividing our speedup values over default Vitis by a factor of ×3.29.

Table 1 presents the speedups of common kernels over the Merlin compiler. On average, it yields comparable speedups, approximately ×1.1 higher when compared to AutoDSE. In terms of the time required for AutoDSE to generate optimized designs, the authors report a geometric mean of 1.1 hours. This highlights a significant distinction from our work, as we can provide the corresponding directives with an average delay of 38ms, as detailed in section 3.1. In summary, our data-driven optimization approach performs competitively with AutoDSE and excels in terms of speed, with orders of magnitude faster inference times.

4 CONCLUSION

We introduced an ultra-fast cluster-based hierarchical HLS design optimization strategy that automatically identifies and applies the most promising directive configurations for minimizing design latency. Our method delivers efficient design solutions, including an average geometric mean speedup of approximately ×7.2 compared to manually optimized designs and ×1.35 compared to over-provisioning. Furthermore, it achieves a high design feasibility score and maintains an average inference latency of 38ms. Notably, it provides design quality comparable to other methods but at speeds 2-3 orders of magnitude faster, making it a highly promising solution for ultra-fast and automated HLS optimization.

ACKNOWLEDGMENTS

This work was partially supported by the EU Horizon Program CONVOLVE under Grant Agreement No: 101070374.

REFERENCES

[1] Cong Jason et al. 2016. Source-to-source optimization for HLS. *FPGAs for Software Programmers* (2016), 137–163.
[2] Che Shuai et al. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 44–54.
[3] Deb Kalyanmoy et al. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation* 6, 2 (2002), 182–197.
[4] Ferretti Lorenzo et al. 2018. Cluster-based heuristic for high level synthesis design space exploration. *IEEE Transactions on Emerging Topics in Computing* 9, 1 (2018), 35–43.
[5] Ferretti Lorenzo et al. 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 (2020), 3736–3747.
[6] Ferretti Lorenzo et al. 2022. Graph Neural Networks for High-Level Synthesis Design Space Exploration. *ACM Transactions on Design Automation of Electronic Systems* 28, 2 (2022), 1–20.
[7] Gautier Quentin et al. 2022. Sherlock: A multi-objective design space exploration framework. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 27, 4 (2022), 1–20.
[8] Kemker Ronald et al. 2018. Measuring catastrophic forgetting in neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32.
[9] Marutho Dhendra et al. 2018. The determination of cluster number at k-mean using elbow method and purity evaluation on headline news. In *2018 international seminar on application for technology of information and communication*. IEEE, 533–538.
[10] Numan Mostafa et al. 2020. Towards automatic high-level code deployment on reconfigurable platforms: A survey of high-level synthesis tools and toolchains. *IEEE Access* 8 (2020), 174692–174722.
[11] Putnam Andrew et al. [n. d.]. Retrospective: A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. ([n. d.]).
[12] Papakonstantinou Alexandros et al. 2011. Multilevel granularity parallelism synthesis on FPGAs. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 178–185.
[13] Reagen Brandon et al. 2014. Machsuite: Benchmarks for accelerator design and customized architectures. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 110–119.
[14] Sohrabizadeh Atefeh et al. 2022. AutoDSE: Enabling software programmers to design efficient FPGA accelerators. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 27, 4 (2022), 1–27.
[15] Sohrabizadeh Atefeh et al. 2022. Automated accelerator optimization aided by graph neural networks. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 55–60.
[16] Schafer Benjamin Carrion et al. 2019. High-level synthesis design space exploration: Past, present, and future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2019), 2628–2639.
[17] Sotirios Xydis et al. 2012. Compiler-in-the-loop exploration during datapath synthesis for higher quality delay-area trade-offs. *ACM Trans. Design Autom. Electr. Syst.* 18, 1 (2012), 11:1–11:35. https://doi.org/10.1145/2390191.2390202
[18] Sotirios Xydis et al. 2015. SPIRIT: Spectral-Aware Pareto Iterative Refinement Optimization for Supervised High-Level Synthesis. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 34, 1 (2015), 155–159. https://doi.org/10.1109/TCAD.2014.2363392
[19] Vanderbauwhede Wim et al. 2018. Domain-specific acceleration and auto-parallelization of legacy scientific code in FORTRAN 77 using source-to-source compilation. *Computers & Fluids* 173 (2018), 1–5.
[20] Yu Cody Hao et al. 2018. S2FA: An accelerator automation framework for heterogeneous computing in datacenters. In *Proceedings of the 55th Annual Design Automation Conference*. 1–6.
[21] Ye Hanchen et al. 2022. Scalehls: A new scalable high-level synthesis framework on multi-level intermediate representation. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 741–755.
[22] Zhong Guanwen et al. 2016. Lin-analyzer: A high-level performance analysis tool for FPGA-based accelerators. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.