AutoScaleDSE: A Scalable Design Space Exploration Engine for High-Level Synthesis

HYEGANG JUN, HANCHEN YE, HYUNMIN JEONG, and DEMING CHEN, University of Illinois at Urbana-Champaign, USA

High-Level Synthesis (HLS) has enabled users to rapidly develop designs targeted for FPGAs from the behavioral description of the design. However, to synthesize an optimal design capable of taking better advantage of the target FPGA, a considerable amount of effort is needed to transform the initial behavioral description into a form that can capture the desired level of parallelism. Thus, a design space exploration (DSE) engine capable of optimizing large complex designs is needed to achieve this goal. We present a new DSE engine capable of considering code transformation, compiler directives (pragmas), and the compatibility of these optimizations. To accomplish this, we initially express the structure of the input code as a graph to guide the exploration process. To appropriately transform the code, we take advantage of ScaleHLS based on the multi-level compiler infrastructure (MLIR). Finally, we identify problems that limit the scalability of existing DSEs, which we name the "design space merging problem." We address this issue by employing a Random Forest classifier that can successfully decrease the number of invalid design points without invoking the HLS compiler as a validation tool. We evaluated our DSE engine against the ScaleHLS DSE, outperforming it by a maximum of 59×. We additionally demonstrate the scalability of our design by applying our DSE to large-scale HLS designs, achieving a maximum speedup of 12× for the benchmarks in the MachSuite and Rodinia set.

CCS Concepts: • **Hardware → Resource binding and sharing**; **High-level and register-transfer level synthesis;**

Additional Key Words and Phrases: High-Level Synthesis, design space exploration, static analysis

ACM Reference format:
HyeGang Jun, Hanchen Ye, Hyunmin Jeong, and Deming Chen. 2023. AutoScaleDSE: A Scalable Design Space Exploration Engine for High-Level Synthesis. *ACM Trans. Reconfig. Technol. Syst.* 16, 3, Article 46 (June 2023), 30 pages.
https://doi.org/10.1145/3572959

1 INTRODUCTION

High-level Synthesis (**HLS**) has been widely adopted due to its ability to generate **register-transfer level** (**RTL**) implementation of a hardware design from a high-level behavioral description often described in a language like C/C++ [33]. This additional level of abstraction

This work is supported in part by the Xilinx Center of Excellence at UIUC, the BAH HT 15-1158 contract, and the NSF A3D3 (Accelerated Artificial Intelligence Algorithms for Data-Driven Discovery) Institute.
Authors' addresses: H. Jun, H. Ye, and H. Jeong, University of Illinois at Urbana-Champaign, 403 Coordinated Science Lab, 1308 W Main Street, Urbana IL, 61801 USA; emails: hgjun2@illinois.edu, hanchen8@illinois.edu, hyunmin2@illinois.edu; D. Chen, University of Illinois at Urbana-Champaign, 250 Coordinated Science Lab, 1308 W Main Street, Urbana IL, 61801 USA; email: dchen@illinois.edu.

1936-7406/2023/06-ART46 $15.00
https://doi.org/10.1145/3572959

allows hardware engineers to experiment with various design choices simply by changing the structure of the high-level description and applying different compiler directives. Additionally, this extra level of abstraction can also serve as a bridge [15, 16, 24, 25] between languages such as the CUDA [23]/OpenCL [36] programming models and the FPGA implementation. As a result, many applications have been successfully developed using HLS, such as neural network accelerators [40], graph accelerators [6], video processing [22], and even general-purpose CPUs based on the RISC-V ISA [32].

However, for engineers to effectively develop hardware implementations capable of taking full advantage of the allocated resources, the high-level description of the application must capture the desired level of parallelism. This can be done through the use of compiler directives that tell the high-level synthesis compiler to generate RTL with the desired level of parallelism and memory bandwidth. Nevertheless, the level of parallelism achieved through compiler directives alone is often sub-optimal, which leads to under- or over-utilizing the available hardware resources.

Thus, to generate high-quality hardware using HLS, the designer must first transform the initial algorithmic description of the application into a form more suitable for the HLS compiler. This often means removing variable loop bounds, perfecting the loops, and restructuring the design into sub-functions. This initial transformation alone usually requires a high level of understanding of the HLS design process, and various documents and manuals [10, 14, 17] detail the best practices when developing hardware using HLS.

This initial transformation process typically only scratches the surface in terms of achievable performance. For example, an essential design technique is to strike a balance between the memory bandwidth and the level of parallelism. A well-known method is tiling the loops so that data locality can be achieved [21]; an example of this is shown in Figure 2. However, this process can be very time-consuming as it requires the designer to transform the code structure while preserving correctness. This process, often difficult and time-consuming, is further exacerbated when the number of loops to be tiled is numerous, spanning across multiple functions.

Due to the large number of design choices that can be made, it is challenging for designers to find a globally optimal solution. Such challenges warranted the development of design space exploration engines that can automatically find the Pareto optimal solution [4, 5, 39, 41]. However, these implementations have scalability issues where they cannot search large designs with multiple loops and functions.

As real-world HLS designs have multiple loops, arrays, and functions, the design space is exponentially proportional to the number of tunable knobs. These knobs can be the compiler directives that parallelize the loops or increase the bandwidth of memory elements (e.g., through array partitioning). They can also be the tiling strategy of a loop. As can be expected, with the increase in the number of variables, the scalability of **Design Space Exploration** (**DSE**) engines became a significant issue. An initial solution to this problem can be dividing and conquering the whole design space. This approach aims to tame the exponential growth of the design space by exploring the design space for individual loops, finding the Pareto optimal point that corresponds to a specific tiling, loop parallelism, and memory bandwidth strategy.

However, one major flaw in this approach is the fact that the overall design cannot be divided into independent design spaces. As real-world HLS designs have multiple loops that share resources and communicate with each other over common memory elements, an optimal solution for one sub-design space can lead to a worse global solution due to incompatibility between sub-solutions. As a result, a combined global design point cannot simply be constructed from the sum of the optimization strategies and performance estimations for each sub-design.

As a result of this discrepancy, a DSE engine that explores the design space using the theoretical estimations will not be able to produce optimal results due to the vast majority of merged

points not being Pareto Optimal. The best solution to this problem is to evaluate all the design points in the design space through an HLS compiler to estimate the performance characteristics accurately. However, each HLS synthesis invocation is expensive in terms of runtime and computational intensity. Thus, if we were to apply this approach to large designs, the scalability of this approach would be extremely limited due to the sheer number of design points that need to be evaluated.

In the case of ScaleHLS DSE [39], to minimize the runtime of the design space exploration phase, they introduced a **quality of result** estimator (**QoR**), eliminating the need to invoke the HLS compiler for each design point. However, for large-scale designs, the disparity between the QoR estimation results and the HLS compiler results varied by a wide range (detailed in Section 3.1). Other works, such as Chimera [41], and AutoDSE [35], due to their reliance on the HLS compiler to evaluate every intermediate design points, required long DSE runtimes for up to 24 hours. Additionally, all existing DSE works surveyed by the paper by Schafer and Wang [34] lack the ability to transform code leaving a large portion of the potential performance increase on the table.

1.1 A Scaleable Design Space Engine for HLS

As a solution to these problems, we present AutoScaleDSE, a scalable DSE engine capable of finding the Pareto optimal solution under resource constraints for large real-world HLS designs. Our design space exploration engine aimed to balance the time spent on design space exploration and HLS compilation. By identifying a key limitation (Section 3) in accurately estimating the intermediate DSE results without invoking the HLS compiler, we were able to decrease the DSE runtime to a couple of hours while also being able to evaluate thousands of design points. Additionally, by taking advantage of existing works by the ScaleHLS [39] team, we were able to transform the source code of the input design, further improving the quality of the final output design. Furthermore, by developing our DSE to be aware of the code structure of the input design, AutoScaleDSE is able to identify critical dimensions of the design space, allowing us to focus on the aspects most essential to the design (Section 4.3).

The high-level structure of our DSE engine is shown in Figure 1. Initially, during the "Source Code Analysis" phase, a lexical analyzer analyzes the source code of the input design, extracting key information regarding the design. Using this information, a graph representation of the input design is constructed and used to drive the exploration process. Then, during the "Design Space Exploration" phase, based on the information collected from analyzing the input design, we explore the design space using a random forest classifier to predict the quality of the intermediate design points. Subsequently, in the "Design candidate Selection and Transformation" phase, the most optimal design point is selected from the merged design space, and the corresponding code transformations and compiler directives are applied and inserted. Finally, during the "Design Candidate Evaluation and Optimization" phase, based on the quality of the final design candidate by comparing the DSE and HLS compiler estimation results, we either export the final design or further improve the candidate design. This final exploration phase is done through a genetic algorithm in conjunction with the input design analysis information to efficiently evolve the quality of the design with minimal HLS compiler invocations.

The rest of the article is organized as follows. Section 2 introduces the background for the algorithms and tools used in our DSE engine. Section 3 provides experimental results that motivated us to develop our DSE engine. Section 4 describes how the Random Forest algorithm and the Evolutionary Algorithm are used to solve the design space merging problem. Section 5 brings everything together, detailing the building blocks in Figure 1. Finally, in Sections 6 and 7, we present our evaluation results and conclude this article.

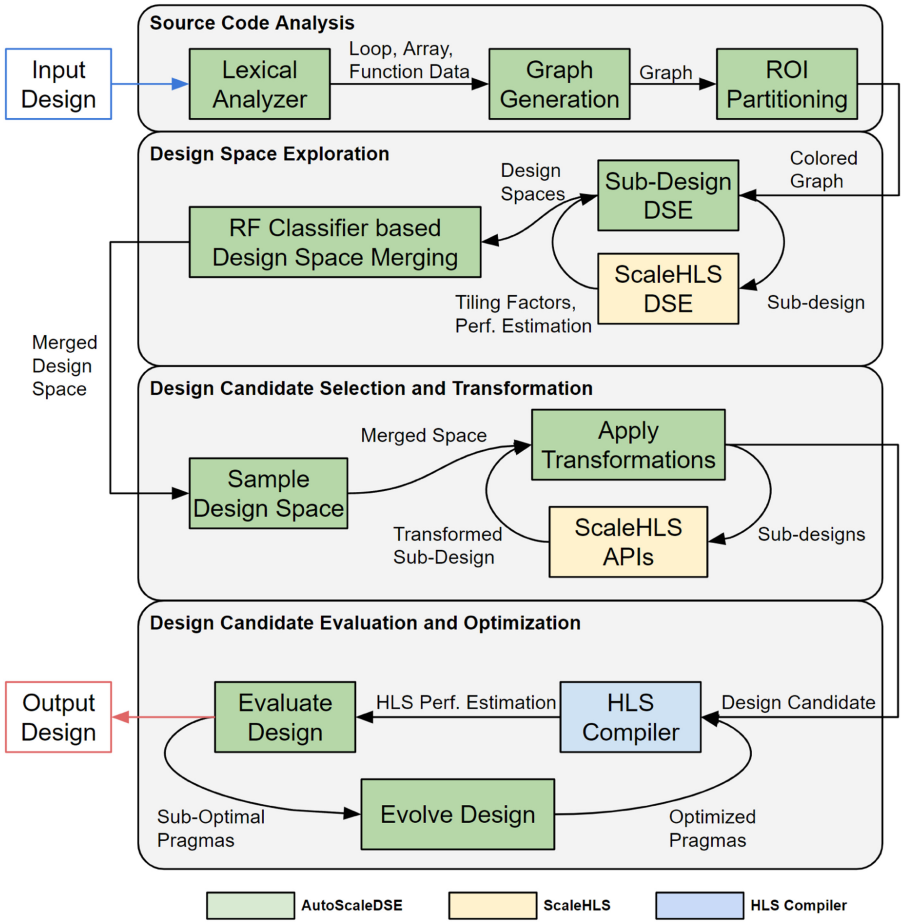


Fig. 1. AutoScaleDSE overview.

2 BACKGROUND

2.1 Definition of a Sub-Design

The smallest quanta of our DSE engine is the loop band, which corresponds to the minimal sub-design from which we build the global design space. In the context of the article, a sub-design refers to a nested loop structure or loop band. For example, in Figure 2(a) a sub-design is the portion of the code between the lines 6 and 14, and in the case of Figure 3(b) the sub-designs correspond to the lines 6–13, 14–21, and lines 22–29; thus the 3mm benchmark has three sub-designs. Throughout the article, we use the terms sub-design, loop nest, and loop band synonymously with each other.

2.2 ScaleHLS

ScaleHLS [39] is a next-generation HLS compilation flow developed using the **multi-level compiler infrastructure** (**MLIR**) [19]. The ScaleHLS project provides a large library of compiler passes developed for Viavdo HLS that automate the transformation and optimization of code.

2.2.1 ScaleHLS Optimization Passes. ScaleHLS is able to represent and optimize high-level hardware designs at three different abstraction levels, including graph, loop, and directive. At the graph

```
void kernel_gemm(float alpha, float beta,
                 float C[20][25], float A[20][30],
                 float B[30][25]) {
#pragma HLS array_partition variable=A complete dim=2
#pragma HLS array_partition variable=B complete dim=1
    for (int i = 0; i < 20; i++) {
        for (int j = 0; j < 25; j++) {
            C[i][j] *= beta;
#pragma HLS PIPELINE II=1
            for (int k = 0; k < 30; k++) {
                C[i][j] += alpha * A[i][k] * B[k][j];
            }
        }
    }
}
```

(a) gemm small data set.

```
void kernel_gemm(float alpha, float beta,
                 float C[20][25], float A[20][30],
                 float B[30][25]) {
#pragma HLS array_partition variable=A cyclic factor=6 dim=2
#pragma HLS ARRAY_PARTITION variable=B cyclic factor=6 dim=1
#pragma HLS ARRAY_PARTITION variable=B cyclic factor=5 dim=2
#pragma HLS ARRAY_PARTITION variable=C cyclic factor=5 dim=2
 for (int v5 = 0; v5 < 20; v5 += 1) {
  for (int v6 = 0; v6 < 5; v6 += 1) {
   for (int v7 = 0; v7 < 5; v7 += 1) {
#pragma HLS PIPELINE II=1
    for (int v8 = 0; v8 < 5; v8 += 1) {
     for (int v9 = 0; v9 < 6; v9 += 1) {
      if ((v9 + (v7 * 6)) == 0) {
       float v10 = C[v5][(v8 + (v6 * 5))];
       float v11 = v10 * beta;
       C[v5][(v8 + (v6 * 5))] = v11;
      }
      float v12 = A[v5][(v9 + (v7 * 6))];
      float v13 = alpha * v12;
      float v14 = B[(v9 + (v7 * 6))][(v8 + (v6 * 5))];
      float v15 = v13 * v14;
      float v16 = C[v5][(v8 + (v6 * 5))];
      float v17 = v16 + v15;
      C[v5][(v8 + (v6 * 5))] = v17;
     }
    }
   }
  }
 }
}
```

(b) Tiled gemm small data set.

Fig. 2. Applying loop tiling to the gemm benchmark by a factor of (1,5,6).

level, the coarse-grained computation graph is legalized and split into a dataflow graph that can be executed in a pipelined manner. At the loop level, each node of the graph is lowered to nested loops, where loop transformations can be applied to improve data locality, parallelism, and pipeline efficiency. Finally, at the directive level, HLS-specific directives and primitives are used to guide the micro-architecture generation and improve the hardware performance appropriately.

2.2.2 ScaleHLS Quality of Result Estimator. In order to evaluate the benefit of different optimization passes of ScaleHLS and guide the automated DSE, ScaleHLS provides a QoR estimator to estimate the latency and resource utilization by analyzing the IR of the design. The QoR estimator leverages an **as-late-as-possible** (**ALAP**) algorithm to schedule the operators in the IR to calculate the clock cycles of the design. During the scheduling phase, the number of on-chip memory banks is considered in order to schedule the memory load and store operations accurately. For pipelined loop nests, loop-carried dependencies, and memory port conflicts, are honored when determining the minimum achievable initiation interval of the pipeline.

2.2.3 ScaleHLS DSE. With the QoR estimator rapidly evaluating each design's latency and resource utilization, ScaleHLS provides an automated DSE to search for the optimized design solutions given the physical constraints of the targeted platform. The ScaleHLS DSE has two phases of searching: (1) local optimizations that search for the Pareto frontier in the area-latency space of each loop nest in the targeted design; (2) global optimization based on a dynamic programming algorithm that merges the Pareto frontier of every loop nest in order to generate the Pareto frontier of the whole design. With the discovered Pareto frontier, the DSE engine is able to determine the optimized design points with the resource constraints.

2.2.4 Example Case. The example shown in Figure 2(a) is of the Polybench [29] gemm kernel using the small dataset (dataset refers to the problem size; changing the dataset changes the size of

```
1  void kernel_2mm(float alpha, float beta,
2                  float tmp[40][50], float A[40][70],
3                  float B[70][50], float C[50][80],
4                  float D[40][80]) {
5      int i, j, k;
6      for (i = 0; i < 40; i++) {
7          for (j = 0; j < 50; j++) {
8              tmp[i][j] = 0;
9              for (k = 0; k < 70; ++k) {
10                 tmp[i][j] += alpha * A[i][k] * B[k][j];
11             }
12         }
13     }
14     for (i = 0; i < 40; i++) {
15         for (j = 0; j < 80; j++) {
16             D[i][j] *= beta;
17             for (k = 0; k < 50; ++k) {
18                 D[i][j] += tmp[i][k] * C[k][j];
19             }
20         }
21     }
22 }
```

(a) 2mm small data set.

```
1  void kernel_3mm(float A[40][60], float B[60][50],
2                  float C[50][80], float D[80][70],
3                  float E[40][50], float F[50][70],
4                  float G[40][70]) {
5      int i, j, k;
6      for (i = 0; i < 40; i++) {
7          for (j = 0; j < 50; j++) {
8              E[i][j] = 0;
9              for (k = 0; k < 60; ++k) {
10                 E[i][j] += A[i][k] * B[k][j];
11             }
12         }
13     }
14     for (i = 0; i < 50; i++) {
15         for (j = 0; j < 70; j++) {
16             F[i][j] = 0;
17             for (k = 0; k < 80; ++k) {
18                 F[i][j] += C[i][k] * D[k][j];
19             }
20         }
21     }
22     for (i = 0; i < 40; i++) {
23         for (j = 0; j < 70; j++) {
24             G[i][j] = 0;
25             for (k = 0; k < 50; ++k) {
26                 G[i][j] += E[i][k] * F[k][j];
27             }
28         }
29     }
30 }
```

(b) 3mm small data set.

Fig. 3. 2mm and 3mm Polybench benchmarks.

the arrays and loops). In the example shown, the loop structure of the gemm benchmark is initially not perfect. A perfect loop structure is defined as a loop structure where only the innermost loop has contents [14]. Thus, due to line 8 of Figure 2(a), this example does not have a perfect loop structure. Various reference manuals recommend this loop structure [14, 17] as it allows for further compiler optimization in the HLS process. Conventionally to perfectivize this example, the user has to rewrite the code manually. However, through ScaleHLS the code transformation process from Figure 2(a) to (b) is as simple as applying the "-affine-loop-Perfectize" and "-affine-loop-tile" compiler passes.

2.3 Loop Tiling

Loop tiling or loop blocking [38] is a widely used compiler technique that improves the data locality of various algorithms. This concept can also be used in HLS designs to improve parallelism and data locality. Figure 2 shows an example where the gemm [29] kernel (Figure 2(a)) has been tiled by a factor of (1,5,6). Corresponding to the tiling factor of (1,5,6), the loops at lines 7 and 10 of Figure 2(a) have been split into the loops at lines 9, 10, 12, and 13 of Figure 2(b). Specifically, the loop at line 7 of Figure 2(a) has been transformed to be represented by the two loops at lines 9 and 12 of Figure 2(b), where the inner point loop (line 12 of Figure 2(b)) has the trip count corresponding to the tiling factor. The same is true for the loop at line 10 of Figure 2(a) that has been transformed into the loops at lines 10 and 13 of Figure 2(b). Note that the loop at line 6 of Figure 2(a) with a corresponding tiling factor of 1 is not transformed into a multi-loop representation in Figure 2(b) only represented by the single loop at line 8 of Figure 2(b).

2.3.1 Loop Tiling—Degree of Parallelization. Conventionally, without loop tiling, the example in Figure 2(a) can only be parallelized by applying the loop pipeline or loop unroll HLS compiler directives to either of the three for-loops in the sub-design. Furthermore, if the "pipeline" directive is used, the "unroll" directive becomes redundant for other loops in the loop nest. For the gemm

example in Figure 2(a), as the loop at line 7 has been pipelined, the pipeline directive implicitly unrolls the portion of the code within the scope of the loop [14]. In this example, the loop at line 10 of Figure 2(a) will be fully unrolled. However, if we wish to increase the kernel's parallelism, we cannot simply unroll the other loops in the loop nest using the unroll pragma. Take the example where the unroll pragma with a factor of 2 is applied to the parent loop of the pipeline loop of line 7 of Figure 2(a). This "unroll" compiler directive instructs the HLS compiler to create two instances of the pipelined loop. As a result, the two instances must simultaneously be able to read the B[k][j] variable as the values are invariant to the unrolled outer "i" loop at line 6 of Figure 2(a), which is not possible without the restructuring of code.

On the other hand, through loop tiling, HLS designs can be parallelized in a more fine-grained manner. For example, in Figure 2 the same level of parallelism of 30 is achieved for both cases by applying the compiler directives as shown. Suppose then we wish to increase the level of parallelization. In that case, for the non-tiled version, the loop at line 7 has to be pipelined, only being able to control the level of parallelism by adjusting the unroll factor of the loop at line 7 while the loop at line 10 is fully unrolled. However, if we were to tile the loops, the level of parallelism can be adjusted by simply changing the tiling factors.

2.3.2 Loop Tiling—Data Locality. Loop tiling also has the added benefit of optimizing for memory. For example, in the instance in Figure 2(a), due to the pipelined loop being fully unrolled, the array partition scheme should be completely partitioned to meet the memory bandwidth requirements. While for the tiled case, due to greater data locality, the array partitioning can be minimized for each array. This locality is beneficial for HLS designs due to not having to partition arrays by a large factor, as large array partition schemes will lead to sub-optimal designs due to complex interconnects and over-utilization of memory resources [17].

2.4 Definition of Our Design Space

Traditional HLS DSE tools aim to search a vast design space that is comprised of "loop-pipeline," "loop-unroll," and "array-partition," each having corresponding factors that have to be fine-tuned. However, this process can often lead to illegal combinations of directives where, for example, an "unroll" directive is applied to a parent loop of a "pipelined" loop (explained in Section 2.3.1). Thus, an HLS DSE engine has to be aware of the legal combination of the directives. As a side effect of this legal combination rule, the degree of parallelism that can be achieved through directives alone is limited, as explained in Section 2.3.

The ScaleHLS team proposed a different way of capturing the desired level of parallelism through loop tiling. By implicitly always pipelining the tiled loops as shown in Figure 2, we can concisely capture the level of parallelism we wish to achieve. This is further aided by systematically finding the best array partition scheme corresponding to the selected tiling strategy using the ScaleHLS' -array-partition API based on the research by Zhao et al. [42]. As a result of these measures, we can concisely capture our desired design using only the tiling strategy and initiation interval of the pipeline compiler directive (pipeline-ii). The conciseness stems from how the tiling strategy implicitly captures loop unrolling and array partitioning. For the "gemm" and "3mm" kernels, the method for how the design points are encoded is shown in Table 1. Throughout the rest of the article, we use the same method of capturing the level of parallelism through tiling factors, improving upon the method initially developed by the ScaleHLS team.

3 DESIGN SPACE MERGING

The design space of real-world HLS applications is exponentially vast, proportional to the number of tunable knobs. This complexity is further compounded by the fact that each loop band has its

Table 1. Example DSE Outputs

(a) gemm kernel in Figure 2(b).

Target Knob	Name	Factors	Pipeline-ii
Tiling Strategy	Sub-design 1	(1,5,6)	1

(b) 3mm kernel in Figure 3(b).

Target Knob	Name	Factors	Pipeline-ii
Tiling Strategy	Sub-design 1	(1,25,5)	10
Tiling Strategy	Sub-design 2	(25,1,5)	25
Tiling Strategy	Sub-design 3	(5,5,5)	16

own set of tunable knobs. Examples of these knobs may include the compiler directives loop unroll, loop pipeline, and array partition, each with a corresponding factor.

As a result, a DSE engine that targets large designs must either shrink the number of knobs or partition the design space into sub-design spaces. As we aimed to develop a scalable DSE engine, we opted for the latter approach, partitioning the whole design space into sub-design spaces. In addition, rather than selecting the optimal combinations of HLS compiler directives, we captured the desired level of parallelism of a design using the tiling strategy mentioned in Sections 2.3 and 2.4.

The ScaleHLS [39] team has demonstrated favorable results by capturing the level of parallelism of a single sub-design using the tiling strategy. Through compiler passes, ScaleHLS can unroll and pipeline the target design to the desired degree of parallelism based on the tiling strategy. As a result, ScaleHLS DSE can better utilize the given resource constraints due to having a finer control of the level of parallelism, leading to a comparatively higher quality of the final design.

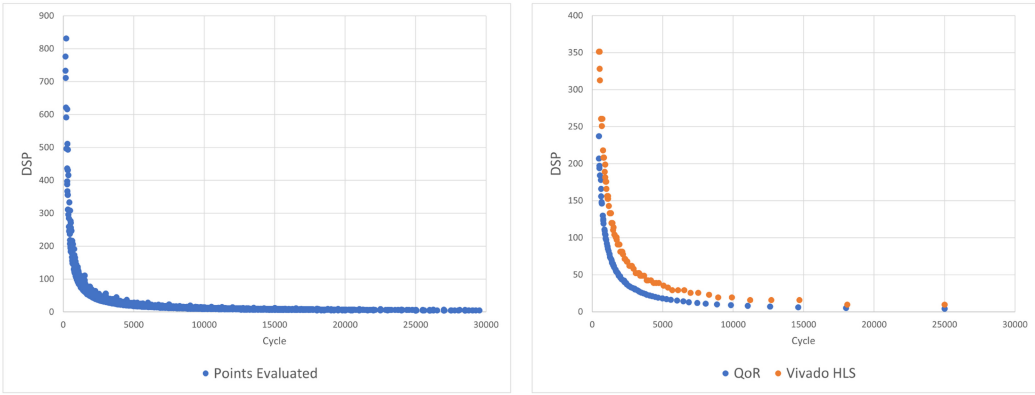
However, although the "tiling" approach was very successful for single sub-design (gemm Figure 2(a)) kernels, when the target design consisted of multiple sub-designs that were sequential and logically linked to each other, the final results were not ideal. Through our experiments, we learned that this was due to the incompatibility of tiling strategies between sub-designs. In this section, we introduce and explore the shortcomings of the tiling strategy method. In Section 3.1, we introduce the method of generating the design space for each sub-design while also evaluating the QoR estimator of ScaleHLS. Subsequently, Section 3.2 explains how the sub-design spaces are merged to build a global design. Finally, in Sections 3.3.1 and 3.3.2, we present and analyze the design space merging problem that limits the overall quality of the final design.

3.1 Sub-Design Design Space Generation

To generate the initial design space for each sub-design, we used the ScaleHLS DSE engine due to its speed and its ability to optimize code using compiler passes automatically. ScaleHLS DSE is a tool included with the ScaleHLS package that can explore the design space consisting of compiler passes, tiling strategies, and conventional compiler directives.

By utilizing ScaleHLS DSE, we can take advantage of the compiler passes developed by the ScaleHLS team (such as loop perfectization, remove variable bound, and loop order optimization), allowing for the initial code structure not to be HLS optimal (having variable bounds and non-affine loops). Additionally, through the QoR estimator, ScaleHLS DSE can explore a massive amount of design points in a short amount of time, not having to evaluate the design points using the complete HLS compilation process. Figure 4(a) plots 1,826 design points explored by ScaleHLS DSE for the gemm [29] kernel (Figure 2(a)), the runtime for the DSE being under 30 seconds. Consistent with the DSE procedure outlined in the ScaleHLS paper [39], the exploration points are clustered close to the Pareto frontier, demonstrating the effectiveness of the exploration process.

Additionally, we evaluated the QoR estimator's accuracy by comparing the QoR estimation results to Vivado HLS for 85 Pareto optimal points. In Figure 4(b), it can be seen that the accuracy of the QoR estimator is comparable to Vivado HLS. On average, the latency estimation results of the QoR estimator had an error rate of 1.8%, while the DSP utilization estimation results had an error



(a) Design Points Explored by ScaleHLS DSE.

(b) QoR accuracy evaluation to Vivado HLS.

Fig. 4. Evaluation of ScaleHLS DSE engine and QoR estimator.

rate of 94%. Notably, for each design, the DSP utilization estimation results between ScaleHLS QoR and Vivado HLS were off by a constant factor. This was because although the ScaleHLS QoR estimator could accurately predict the DSP utilization of a single sub-design, its accuracy diminished due to the resource sharing between sub-designs.

3.2 Sub-Design Design Space Merging

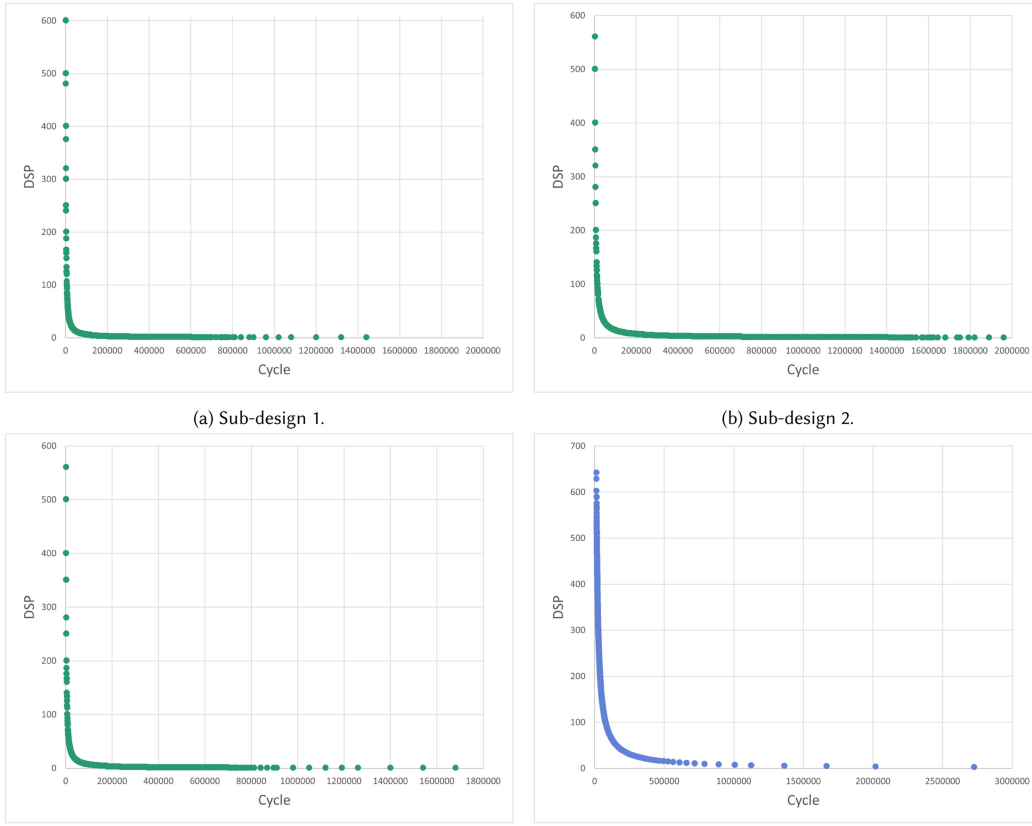
Design space merging refers to the merging of sub-design spaces to create a global design space. A typical method used is as follows. Initially, people can generate the sub-design spaces. Thereafter, they build the global design space by merging sub-designs step by step. During merging, they assume that a point's merged latency and resource utilization is the sum of each sub-design's latency and resource utilization, and they also assume that resources between sub-designs are not shared.

Figure 5 illustrates the merging of the sub-designs for the 3mm [29] (Figure 3(b)) benchmark following the aforementioned merging method. The sub-design spaces correspond to Figures 5(a), (b), and (c), which are merged to create Figure 5(d). As an example, to create the global design point with a latency of 27,229 cycles utilizing 289 DSPs, we would combine the design points (7,208 cycle, 84 DSP) + (12,011 cycle, 117 DSP) + (8,010 cycle, 88 DSP). We note that after merging, non-Pareto optimal points are pruned.

Even though the ScaleHLS QoR estimator can estimate the resource sharing within a single sub-design, the QoR estimator is unable to estimate resource sharing when the number of sub-designs is greater than one due to the dramatic increase in complexity of the design space. For this reason, the ScaleHLS team employed the merging method. This method is shown in the example above, where we estimate the merged design space's resource utilization simply by summing each sub-design's resources.

3.3 Design Space Merging Problem

The performance characteristics of a merged design point did not necessarily correspond to the more accurate performance estimations obtained from synthesizing the merged design. To evaluate how the merged Pareto optimal points translate to the merged design space, we evaluated 300 merged points by synthesizing them using Vivado HLS. The results shown in Figure 6(a)



(a) Sub-design 1.

(b) Sub-design 2.

(c) Sub-design 3.

(d) Merged Design Space.

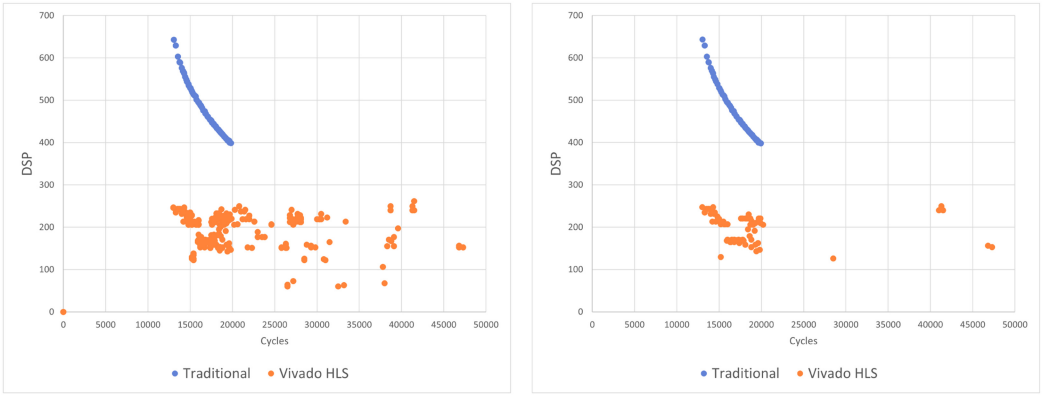
Fig. 5. 3mm sub-design spaces and merged global design space.

were produced using the 3mm [29] Polybench computational kernel (Figure 3(b)) using the small[1] dataset. For each loop band, the sub-design is explored, and a Pareto curve is generated using the ScaleHLS DSE. Afterward, we merge the three Pareto curves to produce a computed merged Pareto curve shown in Figure 6(a) in blue. Next, we evaluate each blue point using Vivado HLS, and the results are the points in orange in Figure 6(a). Figure 6(a) plots 327 data points that we gathered using this approach.

Our experiments showed the fallacy of assuming that a merged sub-design point's performance and resource utilization is the sum of the sub-design points' performance and resource utilization. Additionally, not all merged points could be synthesized by Vivado HLS as the points at (0, 0) correspond to failed HLS synthesis points.[2] The results tell us that we cannot merge Pareto points using the traditional method and assume that the strategies associated with the merged point will actually translate into real-world performance. If we do so, we easily run into making the error of choosing a theoretical Pareto optimal point that is, in truth, far from the actual Pareto frontier or, in the worst case, unsynthesizable.

[1]The small dataset is a dimension of the Polybench [29] benchmark, where the size of the dataset defines the size of the arrays and loops of the particular benchmark.

[2]For the example presented in Figure 6(a), six design points failed HLS syntheses, or approximately 2%.



(a) Traditional Pareto Front Merging.

(b) Enhanced Pareto Front Merging.

Fig. 6. Comparison between Traditional and Merged Pareto optimal points evaluated using Vivado HLS.

3.3.1 Design Space Merging Problem —Initial Analysis. To understand what might be causing this disparity between theoretical and HLS compiler estimated performance results, we evaluated four merged design points using the 2mm [29] Polybench benchmark (Figure 3(a)) using the small dataset. The 2mm kernel was useful for our study due to the simplicity of the design space, as the benchmark only has two sub-designs. Using this to our advantage, we experimented with the merging of sub-design points with different tiling strategies. The results are summarized in Table 2.

Using the methodology outlined in Sections 2.3 and 2.4, the tiling factor of (5,10,1)+(10,10,1) translates the code in Figure 3(a), where the first loop band consisting of three nested loops is transformed into five loops by their corresponding factor; the same is true for the second loop. Additionally, because the "pipeline" directive is applied after the third loop, the inner three loops are subsequently unrolled.[3] Thus, the multiple of the tiling factors effectively refers to the degree of parallelism of a loop band. Additionally, as mentioned in Section 2.4, the array partition scheme that best fits the selected tiling strategy is automatically found using the ScaleHLS's -array-partition API based on the research by Zhao et al. [42]. In cases where an array is shared between sub-designs, the -array-partition API automatically selects the greater partitioning scheme.

The results in Table 2 show that when combining sub-designs that have been tiled, not only do the tiling factors matter, but also the ordering of the factors must be taken into account. For loop1, when it is synthesized independently of loop2, the latency for the following tiling factors (5,10,1) and (10,5,1) are both 2,815 cycles. As for loop2, the independent latency values for the tiling factors (10,10,1) and (8,16,1) are 1,620 and 1,270, respectively. Finally, the target pipeline ii for all experiments is the same at 1.

The first three experiments behaved as predicted when we combined the independent sub-design into a merged design. The total latency was the sum of the individual sub-designs. Interestingly, loop2's performance degraded considerably in "experiment 4." The fourth experiment only differed from "experiment 2" regarding the order of the tiling strategy of loop1. Even though "experiment 4" had the same level of parallelism as "experiment 2," when they were combined using a different ordering of tiling strategies, loop2 could not achieve a pipeline ii = 1 and instead ended up with an ii of 8. Another interesting observation is the results of experiments 1 and 3, in which even though the order of the tiling strategy was different, the overall latency remained the same.

[3]We have omitted the transformed code as the compiler-based transformation automatically unrolls the pipelined loop, making the final code thousands of lines long.

Table 2. Vivado HLS Results of 2mm using Different Tiling Strategies

	Tiling Strategy	loop 1 cyc.	loop2 cyc.	Act. Pipeline-ii	Total cyc.
Exp. 1	(5,10,1) + (10,10,1)	2,815	1,620	(1) - (1)	4,435
Exp. 2	(5,10,1) + (8,16,1)	2,815	1,270	(1) - (1)	4,085
Exp. 3	(10,5,1) + (10,10,1)	2,815	1,620	(1) - (1)	4,435
Exp. 4	(10,5,1) + (8,16,1)	2,815	10,031	(1) - (8)	12,846

Table 3. Tiling Strategy and Array Partition Scheme for Experiments 2 and 4

(a) Experiment 2.

Target Knob	Name	Dims.	Factors	Type/ii
Tiling Strategy	Sub-design 1	N/A	(5,10,1)	ii-1
Tiling Strategy	Sub-design 2	N/A	(8,16,1)	ii-1
Array Partition	tmp	1	8	cyclic
Array Partition	tmp	2	10	cyclic
Array Partition	A	1	5	cyclic
Array Partition	B	2	10	cyclic
Array Partition	C	2	16	cyclic
Array Partition	D	1	8	cyclic
Array Partition	D	2	16	cyclic

(b) Experiment 4.

Target Knob	Name	Dims.	Factors	Type / ii
Tiling Strategy	Sub-design 1	N/A	(10,5,1)	ii-1
Tiling Strategy	Sub-design 2	N/A	(8,16,1)	ii-1
Array Partition	tmp	1	10	cyclic
Array Partition	tmp	2	5	cyclic
Array Partition	A	1	10	cyclic
Array Partition	B	2	5	cyclic
Array Partition	C	2	16	cyclic
Array Partition	D	1	8	cyclic
Array Partition	D	2	16	cyclic

3.3.2 Design Space Merging Problem—Closer Analysis. In the previous section, we demonstrated two experiments where the performance characteristics are drastically different depending on the tiling strategies. To better understand the cause of this degradation in performance, we analyzed the synthesized design using the "Analysis" tab in Vivado HLS. Additionally, we compared the automatically generated array partition schemes between experiments 2 and 4, summarized in Table 3.

Although we expected that the sub-design 2 of "experiment 2" and "experiment 4" would have the same latencies, the synthesis produced different designs with differing characteristics. Upon closer inspection of the synthesized design, we found that sub-design 2 of "experiment 4" only achieved a pipeline-ii of 8, partly due to being unable to read the "tmp" array simultaneously from the on-chip memory element. This was in contrast to "experiment 2," where the same sub-design 2, paired with sub-design 1 with a tiling strategy of (5,10,1), could start reading the "tmp" array simultaneously. From the analysis tab alone, for "experiment 4," the synthesized design appeared unable to read from the on-chip buffers due to a lack of available memory ports, analogous to the case when a design is synthesized with an insufficient number of memory ports.

The lack of "apparent" insufficient memory ports had knock-on effects on subsequent operations within sub-design 2. Most notably, the 128 read operations of array "tmp" (read simultaneously for experiment 2) could not be simultaneously read in the case of "experiment 4" but had to be consecutively read across four control cycles. As a result, subsequent operations could not be scheduled in parallel and needed additional control cycles, leading to the degradation in performance. Curiously, this degradation in performance did not subside simply by increasing the number of memory ports; increasing the array partition factor did not lead to an increase in performance.

Theoretically, the automatically generated array partition schemes appear sufficient for the target design. For example, for experiment 4, with a tiling strategy of (10,5,1) + (8,16,1), the shared memory element "tmp[i][j]" should have an array partitioning scheme of (dim = 1, factor = 10) and (dim = 2, factor = 5) for sub-design 1 corresponding to unrolling the first and second loop in lines 6 and 7 in Figure 3(a) by a factor of 10 and 5 (method outlined in Section 2.3). While for sub-design 2, as the tiling strategy (8,16,1) corresponds to unrolling the first and second loop in lines 14 and 15 in Figure 3(a) by a factor of 8 and 16, the array partition scheme for "tmp[i][k]" should be (dim = 1,

Table 4. Classifier Data Preprocessing and Accuracies

(a) Example of tiling data preprocessing.

Input	(1.1, 1.2, 1.3) + (2.1, 2.2, 2.3) + (3.1, 3.2, 3.3)		
	RFC Model 1	RFC Model 2	RFC Model 3
Grouped	(1.1, 2.1, 3.1)	(1.2, 2.2, 3.2)	(1.3, 2.3, 3.3)
Randomized	(2.1, 1.1, 3.1)	(3.2, 1.2, 2.2)	(2.3, 3.3, 1.3)

(b) Accuracy of various classification methods.

Classification Method	**Accuracy**
GCD	56.39%
Linear Classifier	53.84%
Raw RF Classifier	61.22%
Grouped RF Classifier: Vote ≥ 1	84.61%
Grouped RF Classifier: Vote ≥ 2	72.99%
Grouped RF Classifier: Vote ≥ 3	44.71%
Randomized RF Classifier: Vote ≥ 1	97.03%
Randomized RF Classifier: Vote ≥ 2	82.56%
Randomized RF Classifier: Vote ≥ 3	44.15%

The X.Y notation in Table 4(a) refers first to the sub-design designation (X) and second to the specific loop number (Y) within the sub-design.

factor = 8).[4] When combined, as we select the larger partitioning factor, the final array partitioning scheme for "tmp" (dim = 1, factor = 10) and (dim = 2, factor = 5) (summarized in Table 3(b)) should be sufficient.

The discrepancy between theory and experimental data hints toward a more fundamental reason for the incompatibility between specific tiling strategies. We theorized that depending on the tiling strategy, certain sub-design implementations are compatible or not compatible with other sub-design implementations. From the limited analysis above, we devised two solutions to the sub-design merging problem. Firstly, we aimed to find a method that can predict the compatibility of sub-designs without invoking the HLS compiler. Secondly, we aimed to find a way to increase sub-design compatibility by minimally modifying the overall design after sub-design merging. In the next section, we present our solutions that significantly increase the overall performance of HLS designs that contain more than one sub-design.

4 ENHANCED DESIGN SPACE MERGING

The experiments conducted in Section 3 suggest that the order of the tiling strategy, as well as the tiling factors, are an important metric to consider when combining sub-design spaces. Upon careful deliberation, this is to be expected as the tiling strategy implicitly captures the degree of parallelism and memory bandwidth of a design (by using the "-array-partition" ScaleHLS API, the array partition scheme is algorithmically built based on the tiling strategy). Thus, by specifying a tiling strategy, we constrain the design to a set of resources. It is then up to the HLS compiler to find the optimal design within the given constraints. However, due to the fact that the main operations of high-level syntheses, such as task scheduling and resource binding being NP-hard, the results from the heuristic algorithms may not always be optimal. As a result, certain tiling strategies work well with each other while others do not, as shown in Figure 6(a).

Thus, our goal was to predict the likelihood of whether two sub-designs will be compatible with each other. Additionally, we devised a method for complementing the prediction method. We accomplished this initial goal using a random forest classifier that could, without invoking the HLS compiler, predict the compatibility of two sub-designs. Subsequently, to account for the unavoidable mispredictions of our classifier, we use a genetic algorithm to enhance the quality of our final design by optimizing critical compiler directives. In this section, we present our classifier and our genetic algorithm-based approach.

[4]As the second loop in line 15 in Figure 3(a) does not generate indices for the second dimension of the "tmp" array, partitioning of the second dimension is unnecessary.

4.1 Predicting Sub-Design Compatibility using a Random Forest Classifier

Our initial goal was to prune the non-optimal Pareto points that existed in the design space shown in Figure 6(a). These points correspond to the "degraded" design in experiment 4 mentioned in Section 3.3.2. However, to be able to systematically search the design space in a reasonable amount of time, we could not rely on the HLS compiler to test for the compatibility of the sub-designs as each HLS invocation is computationally intensive and time-consuming.

Thus to solve this problem, we generated 2,601 different tiling combinations for the "3mm [29] small dataset Kernel" and checked their compatibility using Vivado HLS. Out of the 2,601 data points, 59% were valid compatible points while 41% were non-valid incompatible points.[5] We selected the 3mm kernel as it was both small in the respect that we can generate the ground truth dataset in a reasonable amount of time while also being complex enough that insights gained from the dataset are generally relevant. The 3mm kernel is a reasonable estimation of interconnected loops that exist in many applications, such as the matrix multiplication in convolutional neural nets or the Floyd-Warshal [18] algorithm used in many graph-based applications.

Initially, we tried to analytically find a pattern between the tiling strategies using the **greatest common divisor** (**GCD**) method. However, this approach proved to be barely better than randomly guessing. Subsequently, we experimented with machine learning, initially starting with a linear classifier but ultimately using the Random Forest Classifier because of the classifier's high accuracy of 97.03% in predicting the compatibility of sub-designs.

4.1.1 The Greatest Common Divisor Method. Initially, we attempted to predict the compatibility of tiling strategies based on the GCD of the product of the tiling factors by pruning points when the GCD is less than 3. For example, in experiment 1 in Table 2, we first find the products of the tiling factors, which in this case are 50 and 100, corresponding to sub-designs 1 and 2. As the GCD between 50 and 100 is 50, we keep experiment 1. However, for experiment 4, as the GCD between 50 and 128 is 2, we prune experiment 4. However, this approach had a lot to desire as this approach pruned valid points such as experiment 2 that also had a GCD of 2. We quickly abandoned this approach when we found that when we applied the the GCD method to the 2,601 dataset, the accuracy of the GCD method was only 56.39%.

4.1.2 Taking Advantage of Machine Learning. At this point, we realized that this problem could be solved using well-developed machine learning algorithms. To train and test the machine learning models, we randomly split 80% of the 2,601 dataset to create the training set, while the remaining 20% was used as the test set. Using this dataset, our first attempt at machine learning used the Linear layer in the Pytorch [26] library. However, the accuracy at best was only 53.84%, which was worse than the GCD method. From this, we concluded that the low accuracy is due to the problem set not being linearly separable [8]. Having learned of the non-linearity of our dataset, we initially experimented with **Convolutional Neural Nets** (**CNNs**)[6] [20] but quickly transitioned to using the Random Forest Classifier [12].

4.1.3 Random Forest Classifier. We implemented the **Random Forest** (**RF**) Classifier [12] using the Python package scikit-learn [27]. This decision was based on fast training times and its ability to learn non-linear relationships from a comparatively small dataset. Random forest models also minimize the over-fitting issue of traditional decision trees by using multiple randomly generated

[5] This statistic again shows the severity of the degradation in performance when blindly combining sub-designs.
[6] Although CNNs have proved to be extremely powerful in the classification of data, due to the comparatively small dataset size and the small number of features of our data, we concluded that CNNs were not suitable for our use case [31].

ALGORITHM 1: Classifier

```
procedure CLASSIFIER(combined_space, new_point)
    recent ← get_2most_recent_stratergies(combined_space)
    for stratergy in {recent, new_point} do
        if 3 < num_factors then
            stratergy ← get_most_significant_factors(stratergy)
        else
            stratergy ← pad_with_ones(stratergy)
        end if
    end for
    prediction_1 ← RF_model_1(randomize(order1_tiling_factors, order1_new_point))
    prediction_2 ← RF_model_2(randomize(order2_tiling_factors, order2_new_point))
    prediction_3 ← RF_model_3(randomize(order3_tiling_factors, order3_new_point))
    if prediction_1 + prediction_2 + prediction_3 ≥ 1 vote then
        prediction ← True
    else
        prediction ← False
    end if
    return prediction
end procedure
```

decision trees. The inputs to the random forest classifier are the tiling strategies, as they essentially encapsulate the target design of our design.

The initial implementation of the compatibility predictor using the Random Forest Classifier consisted of a single model trained using the training set mentioned above. However, the accuracy of the predictor without any preprocessing of the data could not significantly outperform the GCD method, with an accuracy of only 61.22%.

With the intuition gained from the experiments in Section 3.3.2, we theorized that the order of the tiling strategy per sub-design might have a more direct impact on the final design as the tiling order corresponded to the loop nesting order, and subsequently, the array partition scheme. As a result, we preprocessed the tiling data by grouping them into groups corresponding to their order. Afterward, we assigned a random forest model to each group and trained them using the training set. Using the three random forest models, the prediction of whether the set of tiling strategies was compatible with each other was based on a threshold of the number of votes between the models. For example, for the 2mm example in Figure 3(a), the for-loops are grouped according to the level, i.e., (lines 6 and 14), (lines 7 and 15), and (lines 9 and 17).

When we evaluated them using the testing set, we achieved an accuracy range of 44%–84%, depending on the number of votes. After this initial success, to minimize the overfitting of the models to the training set, we randomized the order of tiling strategies within a group and, with subsequent hyper-parameter tuning, were able to achieve an accuracy range of 44%–97%. The data preprocessing steps and the various model accuracies are summarized in Table 4(a) and (b).

4.2 Using the Random Forest Classifier for Different Designs

As it is incredibly inefficient and resource-intensive to train the random forest classifier during the runtime of the DSE engine, we implemented our flow so that it could discern non-Pareto optimal points using a pre-trained classifier. The pre-training of the random forest classifier can be done

Table 5. Merging of Sub-Design Points Using Our Random Forest Classifier

Step	Combined Design Space	Classifier Input	New Points
1		1 1 1 1 1 1 1 1 1	s1.t1 s1.t2
2	s1.t1 s1.t2	s1.t1 s1.t2 1 s2.t1 s2.t2 1 1 1 1	s2.t1 s2.t2
3	s1.t1 s1.t2 s2.t1 s2.t2	s1.t1 s1.t2 1 s2.t1 s2.t2 1 s3.t1 s3.t2 s3.t3	s3.t1 s3.t2 s3.t3
4	s1.t1 s1.t2 s2.t1 s2.t2 s3.t1 s3.t2 s3.t3	s2.t1 s2.t2 1 s3.t1 s3.t2 s3.t3 s4.t1 s4.t2 s4.t3	s4.t1 s4.t2 s4.t3 s4.t4
5	s1.t1 s1.t2 s2.t1 s2.t2 s3.t1 s3.t2 s3.t3 s4.t1 s4.t2 s4.t3 s4.t4		

For elements denoted as sx.ty, "x" refers to the specific sub-design while "y" refers to the specific tiling factor.

using any dataset that generally best represents the target HLS design we are optimizing (in our case, the 2,601 dataset generated using the 3mm benchmark).

The pre-trained random forest classifiers mentioned above guided the merging process for all the benchmarks in this study. Nevertheless, many situations existed where the problem we tried to classify did not precisely have nine tiling strategies aligning with the inputs of the pre-trained random forest classifiers. We note that the nine tiling strategies correspond to the number of loops the 3mm benchmark shown in Figure 3(b) has. Also, we could not rely on the sub-designs to always have three loops, where the number of loops in a sub-design could be greater than or less than three.

In instances where the number of tiling strategies we had for input to the classifier was less than nine, padding the inputs with 1s so that they totaled nine would allow the classifier to work as intended with minimal accuracy loss. When padding, care was taken to preserve the order of the loops. For example, in the case of a sub-design with two loops with a tiling strategy of (5,10), the 1 is padded last to preserve the correct order for 5 and 10 so that the final tiling strategy is (5,10,1) rather than (1,5,10). This solution was possible as a tiling strategy of 1 does not capture any degree of parallelism and is ignored by the classifier. However, when the number of inputs to the classifier was greater than nine, care was needed so that more essential tiling factors were not lost.

As outlined in Section 4.1.3, the key reason our random forest classifier successfully predicts the compatibility of different sub-design strategies is mainly due to how the tiling strategy is grouped based on the order of the loops. As a result, when the number of loops in the sub-design and the corresponding tiling factors are greater than three, we drop the tiling factors of the deeper loops so that the first three outer loops are correctly grouped. For example, in the case of a sub-design with a strategy of (1,2,3,4,5), we drop 4 and 5 rather than 1 and 2 so that the outer loops are appropriately grouped. In the case in which the number of sub-designs is greater than three, we only use the most recent two sub-designs to predict the compatibility of the incoming merged sub-design. That is, when predicting the compatibility between the existing merged design space with the incoming sub-design space, we used the two most recently merged sub-designs to predict compatibility with the incoming sub-design space.

The algorithm for preprocessing input data during DSE runtime is outlined in Algorithm 1; additionally, an example case for merging four sub-designs is shown in Table 5. This example demonstrates the process of generating a merged design point for an input design that consists of four sub-designs. As a result, it consists of step one where the sub-design point is initially added to the merged design space. Subsequently, the following three steps correspond to the merging of a sub-design point to the existing merged design point. Thus, the example consists of four steps, while the fifth step is shown to illustrate the final merged design point.

In step 1 of Table 5 as the combined design space is empty, the new sub-design points are directly inputted into the combined space circumventing the classifier. In step 2, we predict the compatibility of the new design point and the existing combined space using the classifier. In this example, as the number of tiling strategies for both design points is two, we pad the empty dimensions with

Table 6. Enhanced Worst Case Design Points Using an Evolutionary Algorithm

		Latency	Speedup	DSP	FF	LUT
2mm	Non-Pareto	784 µs	1×	78%	45%	65%
	Evolved-Point	143 µs	5.4×	58%	42%	78%
3mm	Non-Pareto	1,755 µs	1×	27%	22%	67%
	Evolved-Point	414 µs	4.2×	41%	54%	87%

1s taking care to preserve the order. In step 3, this process is repeated, but in this case, as the number of tiling strategies for sub-design point 3 is three, no padding is needed. Finally, in step 4, as the number of tiling strategies the new sub-design point has is four, we drop the least significant strategy, which in this case is s4.t4, as this corresponds to the tiling strategy lowest in the loop hierarchy. Additionally, due to the existing merged design point consisting of more than two sub-designs, the two most recent sub-design points are used as inputs to the classifier. Finally, in step 5, we can observe the contents of the merged design points. We note that if at any point during the merging process, the classifier predicts the new point to be incompatible with the existing point, the new sub-design point is dropped.

Through this approach, we could minimize the number of non-valid/non-Pareto points mentioned in Section 3.3 and shown in Figure 6(a). The result is shown in Figure 6(b); compared to Figure 6(a), the number of non-Pareto points has significantly decreased. Out of the 327 points in Figure 6(a), after the classifier was applied, the number of points that survived the merging process was a little over 80 points and are the points shown in Figure 6(b). Additionally, when our pre-trained RF classifier was applied to other benchmarks in the Polybench [29] suite, the accuracy of the predicted results did not drastically decrease, with an accuracy of 91.37% and 87.55% for the 2mm and correlation benchmarks, respectively. The key takeaway from the comparison was that the classifier effectively reduces the number of non-Pareto points but at the cost of losing total design points as the number of sub-design space mergings increases. In other words, we can decrease the number of non-Pareto points at the cost of shrinking the design space, limiting the valid points that could be compatible with subsequent sub-designs.

4.3 Complementing the Limits of the Random Forest Classifier

Although the random forest classifier was able to reduce the number of non-Pareto optimal points, it was unsuccessful in completely removing them. Additionally, due to the use of a pre-trained Random Forest Classifier, we had to pre-process the input, either having to pad or drop data. As a result, the quality of the merged design space degraded as the number of sub-designs grew. We theorize that the reason behind this degradation is firstly due to the random forest classifier not being able to capture the relationships between tiling strategies due to the padding and dropping of tiling factors. Secondly, due to the grouping of tiling factors based on their order, we theorize that this is at the cost of losing the relationship information that may exist across groups.

Based on the intuition gained through the experiments in Section 3.3.2, we suspected that some sub-design combinations were incompatible with each other due to memory bandwidth issues between logically connected sub-designs. Thus, we hoped that by fine-tuning the array partition schemes of a design, we would be able to alleviate the problems of this memory bandwidth issue. However, through analysis alone, it was extremely difficult to find a more optimal array partition scheme as theoretically, the automatically generated partition schemes should have been sufficient (Section 3.3.2).

We speculated that the reason that some merged design points are not Pareto optimal was due to the HLS process not being effective for certain design constraints. The DSE problem for HLS

designs is a well-studied topic in HLS research [3, 9, 11, 28]. In the paper by Ferrandi et al. [9], they advocated for and proposed using Evolutionary algorithms for HLS DSE as the sub-tasks of high-level synthesis (scheduling, resource binding, etc.) are notoriously NP-complete. Additionally, through their implementation of the Evolutionary Algorithm, the Chimera [41] team was able to report favorable results for Vivado HLS. Inspired by these works, we implemented an evolutionary algorithm that was able to augment our flow so that non-Pareto optimal points could be enhanced for use in the final design.

Our implementation of the evolutionary algorithm, specifically the **genetic algorithm** (**GA**) [2], shares the basic building blocks of crossover and mutation. However, unlike previous works by Yu et al. and Ferrandi et al., which used GA for HLS DSE, the knobs to be explored are limited to the array partition schemes of arrays shared between sub-designs. This decision to restrict the knobs to the arrays partition schemes shared between sub-designs stems from bandwidth limitations we observed in Section 3.3.2. This restriction also has an added benefit where we could limit the number of knobs explored by the GA as the complexity of the search space is a well-known limitation of genetic algorithms.

For a single sub-design, our experiments found that the genetic algorithm approach to DSE could not beat the array partition schemes produced by analyzing the memory access patterns [42]. This was an expected outcome as the memory bandwidth contention was not present in a single sub-design. However, for merged designs, due to the sub-designs competing for the same resources during HLS synthesis, in some instances, the result from the HLS compiler was many times worse than the sum of the performance characteristics of the sub-designs (Section 3.3.1). As a result of these factors, when we only targeted the shared arrays using the genetic algorithm, significant gains in performance and resource utilization could be achieved.

We evaluated the effectiveness of our genetic algorithm using two non-Pareto points, each from the 2mm, and 3mm Polybench [29] benchmarks, and is reported in Table 6. The two cases are examples of non-Pareto optimal design points, such as the points at the 40,000 cycle mark in Figure 6(b). Using the proposed genetic algorithm, we were able to increase the design points' performance by finding a more optimal array partition scheme for shared arrays used in the design.

Two examples best describe the effectiveness of our GA algorithm and the counterintuitive solutions it produces. The performance characteristics are shown in Table 6, while the array partition schemes before and after the GA method was applied are summarized in Table 7. In the case of the 2mm (Figure 3(a)) kernel, we employed the GA to the shared array "tmp" with an initial array partition scheme of (cyclic, factor = 5, dim = 1) and (cyclic, factor = 10, dim = 2). However, after the GA was used, the new array partition scheme for "tmp" became (cyclic, factor = 5, dim = 1), eliminating the array partition scheme for the second dimension of the array. Similarly, in the case of the 3mm (Figure 3(b)) example, before the GA was applied, the array partition scheme for "E" was (cyclic, factor = 8, dim = 1) and (cyclic, factor = 5, dim = 2) while the array partition scheme for "F" was (cyclic, factor = 10, dim = 1) and (cyclic, factor = 5, dim = 2). After the GA was applied, the array partition scheme for E became (cyclic, factor = 5, dim = 2) while F became (cyclic, factor = 10, dim = 1). Surprisingly, in both examples, we increased the overall design's performance by decreasing the memory bandwidth of the share elements.

Upon closer inspection of the synthesized design, using the "Analysis" tab in Vivado HLS, the implemented design before and after the GA was applied was completely different. For example, in the 2mm example, although initially, the reading of data from memory was slower due to the decrease in memory ports, performance gains were made due to subsequent tasks being scheduled concurrently. In comparison, before GA was applied, even though the reading of data from memory was faster, the design's overall performance suffered due to subsequent tasks being sequentially

Table 7. Array Partition Schemes for Shared Arrays for the 2mm and 3mm Kernels before and after the GA Algorithm is Applie

(a) 2mm.

Target Knob	Name	Dimensions	Factors	Type
Before GA: Array Partition	tmp	1	5	cyclic
Before GA: Array Partition	tmp	2	10	cyclic
After GA: Array Partition	tmp	1	5	cyclic

(b) 3mm.

Target Knob	Name	Dimensions	Factors	Type
Before GA: Array Partition	E	1	8	cyclic
Before GA: Array Partition	E	2	5	cyclic
Before GA: Array Partition	F	1	10	cyclic
Before GA: Array Partition	F	2	5	cyclic
After GA: Array Partition	E	2	5	cyclic
After GA: Array Partition	F	1	10	cyclic

scheduled. This observation is in line with the arguments for the genetic algorithm in the paper by Ferrandi et al. [9].

4.4 A Two-Pronged Solution to the Merging Problem

The effectiveness of the GA algorithm hints at the underlying reason why merged design points are sometimes not the sum of the sub-design points explained in Section 3 and shown in Figure 6(a). From the evidence presented in Sections 3 and 4, we conclude that for certain resource constraints, the HLS compiler is unable to produce an optimal design. Thus, to ensure the scalability of our DSE engine, we attempt to solve this problem with a two-pronged approach. Firstly, we minimize the number of non-Pareto points in the design space through the use of a pre-trained random forest classifier. The minimization of non-Pareto optimal points has the added benefit of increasing the quality of the merged design space by limiting the likelihood of non-Pareto optimal points being subsequently merged. Secondly, in the event that a merged design point is not Pareto optimal (due to the RF classifier's margin of error), we apply the genetic algorithm to the shared memory elements to find a solution that is more compatible with the HLS compiler. Through these methods, we can increase our DSE engine's scalability, allowing us to develop the DSE engine presented in the next section.

5 A SCALABLE DESIGN SPACE ENGINE

In this section, we bring everything together, detailing how the building block fits together to become AutoScaleDSE shown in Figure 1. Initially, the lexical analyzer extracts the necessary information about the design needed for our flow, such as the array sizes, loop trip count, and the number of function calls. Based on this information, a function call graph is generated, embedded with information such as the hierarchical structure of the design, the computational intensity of a sub-design, and the dependencies between sub-designs. The overall design is then divided into separate **regions of interest** (**ROIs**) based on the function call graph. The ROI may consist of a single function or span across functions, based on the dependency analysis. Each ROI has its local resource allocation and is optimized as though it is a distinct HLS design using the methods presented in Section 4. Using the colored graph that partitions the design, we explore the design space of each sub-design using ScaleHLS DSE [39], capturing the explored design space only using the tiling factors, pipeline-ii, and performance estimation. Subsequently, we merged the sub-design spaces using the pre-trained random forest classifier. Having constructed a merged design space for the input design, we select the most optimal design point and then apply the corresponding strategies (transforming code and inserting compiler directives) using ScaleHLS APIs. At this point, we evaluate the design candidate using an HLS compiler to gain a more accurate performance estimation. We export the design if the HLS compiler estimations are within a margin of our computed performance estimations (Section 3.2). However, if the results are sub-optimal, we identify critical compiler directives exporting the final design after the critical compiler directives have been further optimized using the genetic algorithm. More details on this flow will be discussed next.

```
kernel_3mm
├── Loop0
│   └── Loop1
│       └── Loop2
├── Loop3
│   └── Loop4
│       └── Loop5
└── Loop6
    └── Loop7
        └── Loop8
```

(a) Loop Structure.

```
['Loop0', 'for (i = 0; i < 40; i++) {', 'kernel_3mm', '40']
['Loop1', 'for (j = 0; j < 50; j++) {', 'kernel_3mm', '50']
['Loop2', 'for (k = 0; k < 60; ++k) {', 'kernel_3mm', '60']
['Loop3', 'for (i = 0; i < 50; i++) {', 'kernel_3mm', '50']
['Loop4', 'for (j = 0; j < 70; j++) {', 'kernel_3mm', '70']
['Loop5', 'for (k = 0; k < 80; ++k) {', 'kernel_3mm', '80']
['Loop6', 'for (i = 0; i < 40; i++) {', 'kernel_3mm', '40']
['Loop7', 'for (j = 0; j < 70; j++) {', 'kernel_3mm', '70']
['Loop8', 'for (k = 0; k < 50; ++k) {', 'kernel_3mm', '50']
```

(b) Loop Information.

```
('kernel_3mm', 3, '-4-5', 1400000)
('kernel_3mm', 6, '-7-8', 700000)
('kernel_3mm', 0, '-1-2', 600000)
```

(c) Computational Intensity.

```
['Array0', 'A[40][60]', 'A', 'kernel_3mm', 2, '40', '60', 'dependencies', 0]
['Array1', 'B[60][50]', 'B', 'kernel_3mm', 2, '60', '50', 'dependencies', 0]
['Array2', 'C[50][80]', 'C', 'kernel_3mm', 2, '50', '80', 'dependencies', 3]
['Array3', 'D[80][70]', 'D', 'kernel_3mm', 2, '80', '70', 'dependencies', 3]
['Array4', 'E[40][50]', 'E', 'kernel_3mm', 2, '40', '50', 'dependencies', 0, 6]
['Array5', 'F[50][70]', 'F', 'kernel_3mm', 2, '50', '70', 'dependencies', 3, 6]
['Array6', 'G[40][70]', 'G', 'kernel_3mm', 2, '40', '70', 'dependencies', 6]
```

(d) Array Information.

Fig. 7. An example of the information gathered by the Lexical Analyzer.

5.1 Lexical Analyzer and Graph Generator

5.1.1 Lexical Analyzer. We developed a lexical analyzer that captures the necessary information for generating the function call graph that guides the entire DSE process. The data extracted from the input code include the number of loop bands, loop bounds, array dimensions/size, array accesses within loops, number of operations per loop, function calls, and function declarations. An example of the information collected by the lexer is summarized in Figure 7 pertaining to the 3mm benchmark (Figure 3(b)).

Figure 7(a) is a printout of the initial call graph used to represent the loop structure of the design. Figure 7(c) contains information on the loops associated with each sub-design and the computational intensity of each sub-design. The computational intensity is estimated by counting the number of operations and multiplying it by the trip count of the loops. For example, the third entry in Figure 7(c) corresponds to the portion of the code in Figure 3(b) between lines 6 and 13.[7] The first element, "kernel_3mm" refers to the scope of the sub-design. The second element, "0" is the unique identifier given to the outermost loop in the sub-design, while the third element, "-1-2" is the identifier of the loops nested within the sub-design. Finally, the element "600000" refers to the computational intensity of the sub-design, which is proportionate to the least intensive sub-design in the 3mm benchmark of the sub-design of lines 6–13 of Figure 3(b).

Figure 7(b) is the printout of the information regarding the loops, keeping track of the loop identifiers, scope, and loop bound for each loop in the design. Finally, Figure 7(d) is the printout of the information about arrays, such as unique identifiers, array names, scope, dimension, and size of the arrays. Most importantly, we keep a record of the sub-designs where the particular array has been accessed. For example, array "E," which is accessed by sub-designs in Figure 3(b) between

[7]We note that the first sub-design is recorded as the third entry in Figure 7(c) due to the list being sorted based on computational intensity.

(a) Graph representation of the Back-Propagation Benchmark.

(b) The ROIs for the Back-Propagation Benchmark.

Fig. 8. The Back-Propagation MachSuite Benchmark graph representation and ROI partitioning.

lines 6 and 13 and 22 and 29, has been correctly identified by the lexer to be accessed by the sub-designs identified by indices 0 and 6 corresponding to the index given to the outermost loop of a sub-design.

5.1.2 Graph Generation. We generate a function call graph representing the input design by processing the information passed from the lexer. Most importantly, we embed information regarding how closely correlated each sub-design is to each other. For example, in Figure 7 sub-design, 0 and 6 are more closely linked to each other than 0 and 3 due to the sharing of the "E" array. This

ALGORITHM 2: Partition ROI

```
procedure PARTITION_ROI
    CI ← get_computational_intensity
    IP ← get_inflection_point
    ROI_list ← get_ROI_seed(IP)
    while ROI_list ! = coverdesign do
        for ROI in ROI_list do
            ROI ← expand_ROI_towards_correlated_nodes
        end for
    end while
    return ROI_list
end procedure
```

information is critical to the following ROI partition phase. An automatically generated graph is shown in Figure 8 that represents the structure of the MachSuite backprop [30] benchmark.

5.2 ROI Partitioning

5.2.1 Motivation. In Section 4, we introduce our method for merging sub-designs. Although we succeeded in improving the quality of the merged design space, the gains we achieved diminished as the size and complexity of the design increased. Initially, this degradation was so prevalent that merging all the sub-design spaces was not possible for large designs with multiple sub-designs, such as the example shown in Figure 8 of the MachSuite [30] backprop benchmark.

The sub-design merging was not feasible for large designs with multiple subroutines and sub-design because the number of design points sharply decreased during the merging process. Analysis showed that when our classifier was applied to two disparate sub-designs that were not directly linked, the lack of shared resources and dissimilar computational patterns would lead to the two sub-designs being classified as incompatible. As a result, these incompatible points would, in subsequent mergings, misclassify the whole merged design point as not valid, whereas, in some instances, would classify the entire design space as non-compatible. For example, in Figure 8, Loop21 is more closely correlated to Loop27 than to Loop21, because Loop27 exists within the same scope sharing resources and being computationally similar, while Loop6 exists in a different scope, not directly being linked to Loop21.

As a result of this phenomenon, to successfully merge the sub-designs using the methods outlined in Section 4, we needed to merge the sub-designs while being aware of the context that the sub-design fits into. Thus, the problem boiled down to appropriately grouping the sub-designs into different ROIs that consisted of closely linked sub-designs. Our attempt to solve this problem was the primary motivation for generating the function call graph outlined in Section 5.1 so that we could appropriately group the sub-designs into different ROIs by partitioning the generated graph.

5.2.2 Partition. Initially, we were able to find a couple of examples of computational graph partitioning methods developed for GPU streaming applications [13, 37]. However, our graph, not being a computational dataflow graph but a function call graph that only captured the relationships between subroutines and sub-designs, was unsuitable for graph partitioning methods developed for GPU applications. As a result, such prior methods were unsuitable for the graph we generated while also not being able to solve the "grouping of linked sub-design" problem that we needed to solve to apply the sub-design merging method of our DSE to large designs.

However, still inspired by the existing graph partitioning works [13, 37] and the maximum independent set problem [1] in graph theory, we implemented a solution that was able to group closely linked sub-designs into different ROIs. Ultimately, our partitioning/grouping algorithm was able to group sub-designs using an expanding method from a seed node so that we could successfully apply our sub-design merging algorithm to large designs with multiple subroutines.

We employ a simple algorithm for partitioning the overall design into distinct ROIs summarized in Algorithm 2. We initially select the seed sub-designs from which we expand. The number of seeds is determined by the distribution of the computational intensity of sub-designs. For example, if the sorted computational intensity of sub-designs is in the set CI = {9,000, 8,000, 1,000, 500, 400}, as there is a sharp decline in the computational intensity after 8,000, the sub-designs corresponding to 9,000 and 8,000 are selected to be the seeds. If a sharp decrease in computational intensity is not present, computationally intensive sub-designs are chosen proportionally based on the total number of sub-designs in the input design. Afterward, we expand the ROI in each step, giving weight to expand toward more closely correlated sub-designs. Eventually, the ROI partition algorithm terminates when all sub-designs are covered.

Figure 8(b) shows how the backprop benchmark has been divided into different ROIs, represented in different colors. For this example, the starting sub-design seeds were Loop6, Loop17, and Loop27. We can observe that the sub-designs in the red ROI are in the same scope while ROIs in green and blue transcend function boundaries. This is because, during the expansion phase, Loop17 did not expand toward Loop27, while Loop27 gave greater weight to sub-designs that share the same scope. Thus, through our ROI expansion algorithm, we could find an approximate solution to the grouping problem.

5.3 Design Space Exploration

5.3.1 ROI Design Space Exploration. The ROI Design Space Exploration phase of our flow is based on the methods outlined in Section 4. During this phase, each ROI is explored as if it is an independent HLS design. Thus, each ROI has a resource constraint that it needs to satisfy, which is a percentage of the overall resource constraint calculated based on the ROI's computational intensity.

We start the DSE process by initializing the sub-design space using ScaleHLS DSE. Subsequently, we merge the sub-designs using the same random forest classifier we developed to create a merged design space. At this stage, we sample a merged design that we evaluate using Vivado HLS to correct for the merging of sub-designs not being able to account for resource sharing. As shown in Figure 6(a), the error between the traditional merged design space (blue) and the Vivado HLS estimation results (orange) in the DSP dimension is off by a certain factor. Thus, we can correct the traditional merged design space by shifting the space globally by the error factor between the theoretical and HLS evaluated results.

Based on the corrected merged design space, we select a design point that best satisfies the given resource constraints, whether latency or resource utilization (area). Afterward, we again evaluate the selected design point using an HLS compiler to decide whether using the genetic algorithm is warranted. If the estimated results by the HLS compiler closely match the theoretical performance we computed, we forgo using the genetic algorithm to explore the ROI further. If, on the other hand, the estimated results differ by a certain predetermined margin, we apply the genetic algorithm to improve the selected design point further.

5.3.2 Combined Design Space Exploration. We repeat this process for all ROIs until we are left with a design candidate for each ROI. Subsequently, using the ROI design candidates, we construct the overall preliminary design by merging the ROI design points and then applying the

Table 8. AutoScaleDSE vs. ScaleHLS DSE Using the Polybench [29] Medium Dataset

	No. Designs	Prob. Sizes	Implementation	**Latency**	**Speedup**	**DSP**	**FF**	**LUT**
bicg	2	390, 410	Baseline	16 ms	1×	0%	0%	0%
			ScaleHLS	42.050 µs	380×	50%	24%	37%
			AutoScaleDSE	42.050 µs	380×	50%	24%	37%
correlation	4	240, 260	Baseline	1.367 s	1×	0%	0%	1%
			ScaleHLS	0.942 s	1.5×	15%	18%	51%
			AutoScaleDSE	15.467 ms	88×	4%	7%	27%
2mm	2	180, 190, 210, 220	Baseline	1.542 s	1×	0%	0%	0%
			ScaleHLS	8.281 ms	186×	55%	21%	46%
			AutoScaleDSE	1.722 ms	895×	76%	17%	46%
3mm	3	180, 190, 200, 210, 220	Baseline	2.054 s	1×	0%	0%	0%
			ScaleHLS	72.930 ms	28×	35%	20%	33%
			AutoScaleDSE	1.996 ms	1,029×	61%	17%	37%

corresponding tiling and array partition strategies using ScaleHLS APIs. Finally, we apply the genetic algorithm to fine-tune the combined design so that the HLS compiler can generate the best possible designs based on the constraints. Ultimately, we output the final design, the result from our DSE flow.

6 EXPERIMENTAL RESULTS

We evaluated the quality of our DSE engine using four Polybench [29] benchmarks with the addition of two large-scale benchmarks from the MachSuite [30] and Rodinia [7] benchmark sets. The presented results in Tables 8 and 11 have been collected from the report generated by Vivado HLS 2019.2 targeting the "xc7z045-ffg900-2" device.

6.1 Evaluation of Small-Scale Kernels

The four different computational kernels presented in Table 8 are designs that consist of multiple sub-designs that best demonstrate the strengths and weaknesses of our DSE. We refrained from presenting evaluation results of single sub-design kernels due to the ScaleHLS [39] team already demonstrating the quality and scalability for these cases. Additionally, single sub-design kernels have been omitted because the results of AutoScaleDSE and ScaleHLS DSE are identical due to the former using the latter as a building block. We used the dataset/problem size "medium" as the array sizes and loop bounds that correspond to the problem size allowing us to demonstrate the capability of our DSE while also satisfying the resource constraints of the target device. Accordingly, we can see that array sizes and loop bounds vary by the factors in the set {40, 50, 60, 70, 80}.

Table 8 presents the comparison results between AutoScaleDSE and ScaleHLS DSE. This comparison was possible due to ScaleHLS DSE using the traditional approach to merge the sub-designs. Thus, by comparing our approach outlined in Section 4 to ScaleHLS DSE, we can measure the effectiveness of our merging strategy. We note that the "No. Designs" column in Table 8 refers to the number of sub-designs, while the "Prob. Sizes" column relates to the sizes of the arrays and the loop bounds of loops used in the presented kernel.

6.1.1 Bicg. For this case, both ScaleHLS DSE and AutoScaleDSE were able to produce a result that improved the latency for this kernel by 380× compared to the baseline that is not optimized. This dramatic increase in performance is due to the relative simplicity of the "bicg" kernel. The "bicg" kernel consists of only two sub-designs, one of which is a single for-loop while the other is a nested for-loop with a depth of two. As can be seen in the summary of the tiling strategy and array partition scheme in Table 9(a), both ScaleHLS and AutoScaleDSE chose the same design point in

Table 9. Summary of the Tiling Strategy and the Accompanying Array Partition Strategy Chosen by ScaleHLS and AutoScaleDSE for the Bicg and Correlation Benchmarks

(a) bicg.

	Target Knob	Name	DIM	Factors	Type
ScaleHLS	Tiling Strategy	SD1	N/A	(65)	ii-1
	Tiling Strategy	SD2	N/A	(5,65)	ii-8
	Array Partition	A	1	5	cyclic
	Array Partition	A	2	65	cyclic
	Array Partition	s	1	65	cyclic
	Array Partition	q	1	5	cyclic
	Array Partition	p	1	65	cyclic
	Array Partition	r	1	5	cyclic
AutoScaleDSE	Tiling Strategy	SD1	N/A	(65)	ii-1
	Tiling Strategy	SD2	N/A	(5,65)	ii-8
	Array Partition	A	1	5	cyclic
	Array Partition	A	2	65	cyclic
	Array Partition	s	1	65	cyclic
	Array Partition	q	1	5	cyclic
	Array Partition	p	1	65	cyclic
	Array Partition	r	1	5	cyclic

(b) correlation.

	Target Knob	Name	DIM	Factors	Type
ScaleHLS	Tiling Strategy	SD1	N/A	(40,4)	ii-2
	Tiling Strategy	SD2	N/A	(30,5)	ii-2
	Tiling Strategy	SD3	N/A	(20,6)	ii-1
	Tiling Strategy	SD4	N/A	(1,1,20)	ii-3
	Array Partition	data	1	20	cyclic
	Array Partition	data	2	40	cyclic
	Array Partition	mean	1	40	cyclic
	Array Partition	stddev	1	30	cyclic
AutoScaleDSE	Tiling Strategy	SD1	N/A	(40,4)	ii-2
	Tiling Strategy	SD2	N/A	(40,4)	ii-2
	Tiling Strategy	SD3	N/A	(26,4)	ii-1
	Tiling Strategy	SD4	N/A	(1,20,1)	ii-3
	Array Partition	data	1	26	cyclic
	Array Partition	data	2	40	cyclic
	Array Partition	mean	1	40	cyclic
	Array Partition	stddev	1	40	cyclic

SDx refers to specific sub-designs, while x denotes the specific sub-design. DIM refers to the dimension of the array that is being partitioned.

the merged design space, and thus the performance difference between the two DSE engines is identical.

Analysis showed that the two DSE engines chose the same design points because of the benchmark's relative simplicity. In the case of bicg, the two sub-designs minimally shared resources only being linked through array "s." As a result, for this case, the merging problem was not as pronounced as in the other cases, such as the example shown in Figure 6(a). Thus, due to the merged design space being mostly Pareto optimal, the results produced by AutoScaleHLS and ScaleHLS DSE were identical.

6.1.2 Correlation. In the case of the correlation kernel, ScaleHLS DSE was only able to increase the latency speedup by a minimal 1.5× compared to the baseline without any optimizations. Having analyzed the synthesized results of ScaleHLS DSE and the code, the minimal latency gain is due to ScaleHLS DSE's inability to find the tiling strategy for a sub-design effectively. This shortfall is due to the loops in one sub-design having a loop bound of 239. The loop bound of 239 being a prime is thus not able to be tiled in any other way other than 239, for which case, due to the large tiling factor, the synthesized design is extremely inefficient. The additional use of "sqrtf" library functions further complicates the exploration of the design, contributing to the under-performance of the ScaleHLS DSE. On the other hand, AutoScaleDSE achieved an increase in latency speedup of 88× compared to the baseline with considerably less resource utilization. The latency characteristics is a 59× improvement to ScaleHLS DSE and is due to AutoScaleDSE's ability to select tiling strategies that are more compatible with each other.

Further analysis of the tiling strategies demonstrates the severity of the design space merging problem in the case of the correlation kernel, where the sub-designs are closely linked and share numerous memory resources. We can calculate the degree of parallelism of the design by multiplying the tiling factors summarized in Table 9(b). For the ScaleHLS design, the degree of parallelism is calculated to be 57,600,000, while for the AutoScaleHLS design, the degree of parallelism works out to be 53,248,000. We can see that even though ScaleHLS DSE has a higher degree of parallelism of approximately 8%, the corresponding design's latency was only 2% when compared to the final design of AutoScaleDSE. We theorize that this is due to the "design space merging problem," where as a consequence of the incompatibility between tiling strategies, ScaleHLS DSE results in a design that is inferior to the result produced by AutoScaleDSE that has a lower degree of parallelism.

Table 10. Summary of the Tiling Strategy and the Accompanying Array Partition Strategy Chosen by ScaleHLS and AutoScaleDSE for the 2mm and 3mm Benchmarks

(a) 2mm.

	Target Knob	Name	DIM	Factors	Type
ScaleHLS	Tiling Strategy	SD1	N/A	(5,38,1)	ii-2
	Tiling Strategy	SD2	N/A	(1,10,19)	ii-2
	Array Partition	tmp	1	5	cyclic
	Array Partition	tmp	2	38	cyclic
	Array Partition	A	1	5	cyclic
	Array Partition	B	2	38	cyclic
	Array Partition	C	1	19	cyclic
	Array Partition	C	2	10	cyclic
	Array Partition	D	2	10	cyclic
AutoScaleDSE	Tiling Strategy	SD1	N/A	(9,19,1)	ii-2
	Tiling Strategy	SD2	N/A	(9,10,19)	ii-2
	Array Partition	tmp	1	9	cyclic
	Array Partition	tmp	2	19	cyclic
	Array Partition	A	1	9	cyclic
	Array Partition	B	2	19	cyclic
	Array Partition	C	1	19	cyclic
	Array Partition	D	1	9	cyclic

(b) 3mm.

	Target Knob	Name	DIM	Factors	Type
ScaleHLS	Tiling Strategy	SD1	N/A	(12,2,48)	ii-3
	Tiling Strategy	SD2	N/A	(10,14,1)	ii-2
	Tiling Strategy	SD2	N/A	(1,10,19)	ii-3
	Array Partition	A	1	12	cyclic
	Array Partition	A	2	18	cyclic
	Array Partition	B	1	8	cyclic
	Array Partition	B	2	2	cyclic
	Array Partition	C	1	10	cyclic
	Array Partition	D	2	14	cyclic
	Array Partition	E	1	12	cyclic
	Array Partition	E	2	19	cyclic
	Array Partition	F	1	19	cyclic
	Array Partition	F	2	14	cyclic
	Array Partition	G	2	10	cyclic
AutoScaleDSE	Tiling Strategy	SD1	N/A	(18,2,4)	ii-1
	Tiling Strategy	SD2	N/A	(2,42,2)	ii-1
	Tiling Strategy	SD2	N/A	(12,6,2)	ii-1
	Array Partition	A	1	18	cyclic
	Array Partition	A	2	4	cyclic
	Array Partition	B	1	4	cyclic
	Array Partition	B	2	2	cyclic
	Array Partition	C	1	2	cyclic
	Array Partition	C	2	2	cyclic
	Array Partition	D	1	2	cyclic
	Array Partition	D	2	42	cyclic
	Array Partition	E	1	18	cyclic
	Array Partition	E	2	2	cyclic
	Array Partition	F	1	2	cyclic
	Array Partition	F	2	42	cyclic
	Array Partition	G	1	12	cyclic
	Array Partition	G	2	6	cyclic

SDx refers to specific sub-designs, while x denotes the specific sub-design. DIM refers to the dimension of the array that is being partitioned.

6.1.3 2mm and 3mm. These two examples best demonstrate the benefits of the enhanced merging methods described in Section 4. 2mm and 3mm are more complex in comparison to the "bicg," and "correlation" as all sub-designs for these kernels are nested loops with a depth of 3 as shown in Figure 3. Additionally, the variation in array sizes and loop bounds are greater as the 2mm kernel has four different variations while the 3mm kernel has five. This complexity all contributes to the traditional merged design space not being accurate. Thus, for these cases, AutoScaleDSE is able to improve the quality of the final design dramatically.

For the 2mm kernel summarized in Table 10(a), the ScaleHLS DSE results in a speedup of 186× in latency compared to the baseline without any optimizations. AutoScaleDSE is further able to increase the latency speedup factor by approximately 5× with a minimal increase in resource utilization. In the case of the 3mm kernel summarized in Table 10(b), AutoScaleDSE is able to considerably improve upon the latency speedup results of ScaleHLS DSE by 38× with a minimal increase in resource utilization. This result is more surprising when we compare the degree of parallelism between ScaleHLS DSE and AutoScaleDSE for the 2mm and 3mm kernels using the tiling strategies summarized in Table 10. In the case of the 2mm kernel, the degree of parallelism (product of the tiling factors) for ScaleHLS DSE is 36,100 while AutoScaleDSE is 29,241. While for the 3mm kernel, the degree of parallelism for ScaleHLS DSE is 30,643,200 while AutoScaleDSE is 3,483,648. For the 2mm case, even though the ScaleHLS DSE design has a higher degree of parallelism, it is outperformed by AutoScaleDSE by 5×. This disparity is even more pronounced in the 3mm case, where the degree of parallelism for ScaleHLS DSE is greater than AutoScaleDSE by nearly an order of magnitude while being outperformed by AutoScaleDSE by 38×.

Table 11. AutoScaleDSE Evaluation Results for MachSuite [30] and Rodinia [7] Benchmarks

	No.	Array Sizes	Implementation	Latency	Speedup	BRAM18K	DSP	FF	LUT
backprop MachSuite	27	3, 64, 192, 489, 832, 2,119, 4,096	Baseline	754 ms	1×	3%	24%	19%	32%
			AutoScaleDSE	62.669 µs	12×	11%	87%	41%	75%
lud - tiled Rodinia	20	256, 65536	Baseline	1.650 s	1×	0%	1%	1%	3%
			AutoScaleDSE	0.846 µ s	2×	38%	15%	21%	55%

Interestingly and predictably, this increase in performance between ScaleHLS DSE and AutoScaleDSE is in line with the complexity of the two kernels. As the number of sub-designs grew and the complexity of the design in terms of variation in array sizes and loop bounds grew, the gains from ScaleHLS DSE decreased; at the same time, the quality of AutoScaleDSE results increased, thus providing further evidence that the merging problem (Section 3) is intensified as the complexity and the number of sub-designs increase.

6.2 Evaluation of Large-Scale Kernels

We present the results of applying AutoScaleDSE to large-scale benchmarks with multiple sub-designs across multiple functions shown in Table 11. Using these benchmarks, we demonstrate the efficacy of the flow presented in Section 5. AutoScaleDSE was able to automatically identify sub-designs, merging each design space and selecting the optimal global strategy based on the resource constraints. Additionally, the transformation of code was entirely automated using the APIs provided by ScaleHLS to eliminate the human element in design space exploration.

Unfortunately, we could not find a comparable DSE engine as ScaleHLS DSE cannot parse multifunction designs while other DSEs such as Chimera [41] or Comba [42] are unable to transform code. Additionally, we note that although the complexity of the large-scale benchmarks is vastly more complex compared to the 2mm and 3mm kernels, the increase in performance is not as dramatic. We attribute this disparity to the large-scale benchmarks having sub-designs (loops) with variable bounds not optimizable by ScaleHLS DSE. We also note that the sub-designs primarily consist of nested loops with a depth of 2.

As a result, in Table 11 we can see that the performance gains from using AutoScaleDSE on large-scale designs are not as dramatic as the results presented in Table 8. For the backprop MachSuite [30] benchmark, we achieved a latency gain of 12× compared to the unoptimized baseline, while for the lud-tiled Rodinia [7] benchmark, the performance gains were only 2×. We attribute the extent of variable loops used within the benchmarks to the diminishing gains in latency. This is consistent with the lud-tile benchmark using variable loops in sub-designs that were computationally critical to the overall design.

Despite these challenges, the results shown demonstrate the viability of the flow presented in Section 5. AutoScaleDSE was able to automate the process of design exploration and code transformation, producing a result superior to the initial design, an extremely time-consuming process if done manually.

7 CONCLUSION

To the best of our knowledge, AutoScaleDSE is the first to identify and effectively solve the sub-design merging problem for HLS design space exploration. We present experimental results showing that the merging problem is indeed a limiting factor to the divide-and-conquer approach to HLS design space exploration for multi-loop designs. Having presented the sub-design merging problem, we formulate and implement a solution using a pre-trained random forest classifier, achieving an accuracy of 97%, and the genetic algorithm to appropriately merge sub-design points without

having to evaluate them using an HLS compiler individually. Building upon this, we present a flow that can automatically explore large-scale designs using a top-down approach, being aware of the code structure. Finally, we report experimental results demonstrating the efficacy of the developed methods.

REFERENCES

[1] Brenda S. Baker. 1983. Approximation algorithms for NP-complete problems on planar graphs. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science (SCFS 1983)*. 265–273. https://doi.org/10.1109/SFCS.1983.7

[2] Wolfgang Banzhaf, Frank D. Francone, Robert E. Keller, and Peter Nordin. 1998. *Genetic Programming: An Introduction: On the Automatic Evolution of Computer Programs and Its Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

[3] M. C. Bhuvaneswari, D. S. Harish Ram, and R. Neelaveni. 2015. *Design Space Exploration for Scheduling and Allocation in High Level Synthesis of Datapaths*. Springer India, New Delhi, 69–92. https://doi.org/10.1007/978-81-322-1958-3_5

[4] Deming Chen, Jason Cong, Yiping Fan, and Lu Wan. 2010. LOPASS: A low-power architectural synthesis system for FPGAs with interconnect estimation and optimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 18, 4 (2010), 564–577. https://doi.org/10.1109/TVLSI.2009.2013353

[5] Deming Chen, J. Cong, and Junjuan Xu. 2005. Optimal module and voltage assignment for low-power. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'05)*, Vol. 2. 850–855. https://doi.org/10.1109/ASPDAC.2005.1466475

[6] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2021. ThunderGP: HLS-based graph processing framework on FPGAs. In *Proceedings of the 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'21)* (Virtual Event, USA). Association for Computing Machinery, New York, NY, 69–80. https://doi.org/10.1145/3431920.3439290

[7] Jason Cong, Zhenman Fang, Michael Lo, Hanrui Wang, Jingxian Xu, and Shaochong Zhang. 2018. Understanding performance differences of FPGAs and GPUs. In *Proceedings of the 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'18)*. 93–96. https://doi.org/10.1109/FCCM.2018.00023

[8] G. Cybenko. 1989. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)* 2, 4 (Dec. 1989), 303–314. https://doi.org/10.1007/BF02551274

[9] Fabrizio Ferrandi, Pier Luca Lanzi, Daniele Loiacono, Christian Pilato, and Donatella Sciuto. 2008. A multi-objective genetic algorithm for design space exploration in high-level synthesis. In *Proceedings of the 2008 IEEE Computer Society Annual Symposium on VLSI*. 417–422. https://doi.org/10.1109/ISVLSI.2008.73

[10] Michael Fingeroff. 2010. *High-Level Synthesis Blue Book*, Xlibris Corporation.

[11] Xiaohao Gao and Takeshi Yoshimura. 2013. Genetic Algorithm based pipeline scheduling in high-level synthesis. In *Proceedings of the 2013 IEEE 10th International Conference on ASIC*. 1–4. https://doi.org/10.1109/ASICON.2013.6811982

[12] Tin Kam Ho. 1995. Random decision forests. In *Proceedings of the 3rd International Conference on Document Analysis and Recognition*, Vol. 1. 278–282. https://doi.org/10.1109/ICDAR.1995.598994

[13] Huynh Phung Huynh, Andrei Hagiescu, Weng-Fai Wong, and Rick Siow Mong Goh. 2012. Scalable framework for mapping streaming applications onto multi-GPU systems. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*. Association for Computing Machinery, New York, NY, 1–10. https://doi.org/10.1145/2145816.2145818

[14] Xilinx Inc. 2020. *Vivado Design Suite User Guide UG902 (v2020.1)*. https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2020_2/ug902-vivado-high-level-synthesis.pdf.

[15] Xilinx Inc. 2022. *SDAccel: Enabling Hardware-Accelerated Software*. https://www.xilinx.com/products/design-tools/legacy-tools/sdaccel.html.

[16] Intel. 2022. *Intel® FPGA SDK for OpenCL™ Software Technology*. https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html.

[17] R. Kastner, J. Matai, and S. Neuendorffer. 2018. Parallel programming for FPGAs. arXiv:1805.03648. https://arxiv.org/abs/1805.03648.

[18] Gary J. Katz and Joseph T. Kider. 2008. All-pairs shortest-paths for large graphs on the GPU. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware (GH'08)*. Eurographics Association, Goslar, DEU, 47–55.

[19] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A compiler infrastructure for the end of Moore's law. arXiv:2002.11054. https://arxiv.org/abs/2002.11054.

[20] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 86, 11 (1998), 2278–2324. https://doi.org/10.1109/5.726791

[21] Junyi Liu, John Wickerson, and George A. Constantinides. 2017. Tile size selection for optimized memory reuse in high-level synthesis. In *Proceedings of the 2017 27th International Conference on Field Programmable Logic and Applications (FPL'17)*. 1–8. https://doi.org/10.23919/FPL.2017.8056810

[22] Xinheng Liu, Yao Chen, Tan Nguyen, Swathi Gurumani, Kyle Rupnow, and Deming Chen. 2016. High level synthesis of complex applications: An H. 264 video decoder. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 224–233.

[23] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. 2020. CUDA, release: 10.2.89. https://developer.nvidia.com/cuda-toolkit.

[24] Alexandros Papakonstantinou, Karthik Gururaj, John A. Stratton, Deming Chen, Jason Cong, and Wen-Mei W. Hwu. 2009. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. In *Proceedings of the 2009 IEEE 7th Symposium on Application Specific Processors*. 35–42. https://doi.org/10.1109/SASP.2009.5226333

[25] Alexandros Papakonstantinou, Yun Liang, John A. Stratton, Karthik Gururaj, Deming Chen, Wen-Mei W. Hwu, and Jason Cong. 2011. Multilevel granularity parallelism synthesis on FPGAs. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. 178–185. https://doi.org/10.1109/FCCM.2011.29

[26] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

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

[28] Nam Khanh Pham, Amit Kumar Singh, Akash Kumar, and Mi Mi Aung Khin. 2015. Exploiting loop-array dependencies to accelerate the design space exploration with high level synthesis. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE'15)*. 157–162.

[29] Louis-Noël Pouchet and others. 2012. Polybench: The polyhedral benchmark suite. 437 (2012), 1–1. https://scholar.google.com/citations?view_op=view_citation&hl=en&user=TCppIZYAAAAJ&citation_for_view=TCppIZYAAAAJ:Y0pCki6q_DkC.

[30] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. MachSuite: Benchmarks for accelerator design and customized architectures. In *Proceedings of the 2014 IEEE International Symposium on Workload Characterization (IISWC'14)*. 110–119. https://doi.org/10.1109/IISWC.2014.6983050

[31] Shahbaz Rezaei and Xin Liu. 2019. Deep learning for encrypted traffic classification: An overview. *IEEE Communications Magazine* 57, 5 (2019), 76–81. https://doi.org/10.1109/MCOM.2019.1800819

[32] Simon Rokicki, Davide Pala, Joseph Paturel, and Olivier Sentieys. 2019. What you simulate is what you synthesize: Designing a processor core from C++ specifications. In *Proceedings of the 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'19)*. 1–8. https://doi.org/10.1109/ICCAD45719.2019.8942177

[33] Kyle Rupnow, Yun Liang, Yinan Li, and Deming Chen. 2011. A study of high-level synthesis: Promises and challenges. In *Proceedings of the 2011 9th IEEE International Conference on ASIC*. 1102–1105. https://doi.org/10.1109/ASICON.2011.6157401

[34] Benjamin Carrion Schafer and Zi Wang. 2020. High-level synthesis design space exploration: Past, present, and future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2020), 2628–2639. https://doi.org/10.1109/TCAD.2019.2943570

[35] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. 2022. AutoDSE: Enabling software programmers to design efficient FPGA accelerators. *ACM Transactions on Design Automation of Electronic Systems* 27, 4 (Feb. 2022), Article 32, 27 pages. https://doi.org/10.1145/3494534

[36] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering* 12, 3 (2010), 66–73. https://doi.org/10.1109/MCSE.2010.69

[37] Dani Voitsechov and Yoav Etsion. 2014. Single-graph multiple flows: Energy efficient design alternative for GPGPUs. In *Proceedings of the 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA'14)*. 205–216. https://doi.org/10.1109/ISCA.2014.6853234

[38] M. Wolfe. 1989. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing (Supercomputing'89)*. Association for Computing Machinery, New York, NY, 655–664. https://doi.org/10.1145/76263.76337

[39] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. 2021. ScaleHLS: Scalable high-level synthesis through MLIR. *CoRR* abs/2107.11673 (2021). arXiv:2107.11673. https://arxiv.org/abs/2107.11673.

[40] Hanchen Ye, Xiaofan Zhang, Zhize Huang, Gengsheng Chen, and Deming Chen. 2020. HybridDNN: A framework for high-performance hybrid DNN accelerator design and implementation. In *Proceedings of the 2020 57th ACM/IEEE Design Automation Conference (DAC'20)*. 1–6. https://doi.org/10.1109/DAC18072.2020.9218684

[41] Mang Yu, Sitao Huang, and Deming Chen. 2021. Chimera: A hybrid machine learning-driven multi-objective design space exploration tool for FPGA high-level synthesis. In *Intelligent Data Engineering and Automated Learning – IDEAL 2021: Proceedings of the 22nd International Conference (IDEAL'21)* . Springer-Verlag, Berlin, 524–536. https://doi.org/10.1007/978-3-030-91608-4_52

[42] Jieru Zhao, Liang Feng, Sharad Sinha, Wei Zhang, Yun Liang, and Bingsheng He. 2017. COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications. In *Proceedings of the 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'17)*. IEEE, 430–437.

Received 18 June 2022; revised 30 September 2022; accepted 1 November 2022