AutoDSE: Enabling Software Programmers to Design Efficient FPGA Accelerators

ATEFEH SOHRABIZADEH and CODY HAO YU, Computer Science Department, University of California, Los Angeles, USA
MIN GAO, Falcon-computing Inc., USA
JASON CONG, Computer Science Department, University of California, Los Angeles, USA

Adopting FPGA as an accelerator in datacenters is becoming mainstream for customized computing, but the fact that FPGAs are hard to program creates a steep learning curve for software programmers. Even with the help of **high-level synthesis (HLS)**, accelerator designers still have to manually perform code reconstruction and cumbersome parameter tuning to achieve optimal performance. While many learning models have been leveraged by existing work to automate the design of efficient accelerators, the unpredictability of modern HLS tools becomes a major obstacle for them to maintain high accuracy. To address this problem, we propose an **automated DSE framework**—*AutoDSE*—that leverages a bottleneck-guided coordinate optimizer to systematically find a better design point. *AutoDSE* detects the bottleneck of the design in each step and focuses on high-impact parameters to overcome it. The experimental results show that *AutoDSE* is able to identify the design point that achieves, on the geometric mean, 19.9× speedup over one CPU core for MachSuite and Rodinia benchmarks. Compared to the manually optimized HLS vision kernels in Xilinx Vitis libraries, *AutoDSE* can reduce their optimization pragmas by 26.38× while achieving similar performance. With less than one optimization pragma per design on average, we are making progress towards democratizing customizable computing by enabling software programmers to design efficient FPGA accelerators.

CCS Concepts: • **Computer systems organization** → **High-level language architectures**; • **General and reference** → **Reference works**; **Performance;**

Additional Key Words and Phrases: Bottleneck optimizer, customized computing, HLS, Merlin Compiler

ACM Reference format:
Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. 2022. AutoDSE: Enabling Software Programmers to Design Efficient FPGA Accelerators. *ACM Trans. Des. Autom. Electron. Syst.* 27, 4, Article 32 (February 2022), 27 pages.
https://doi.org/10.1145/3494534

Atefeh Sohrabizadeh and Cody Hao Yu contributed equally to this research.
The authors would like to thank Dr. Peichen Pan for his invaluable support with the Merlin Compiler and Dr. Lorenzo Ferretti and Qi Sun for helping with the comparison to their work. This work is supported by the ICN-WEN award jointly funded by the NSF (CNS-1719403) and Intel (34627365), the CAPA award also jointly funded by NSF (CCF-1723773) and Intel (36888881), and CDSC industrial partners. (https://cdsc.ucla.edu/partners/).
Authors' addresses: A. Sohrabizadeh, C. Hao Yu, and J. Cong, Department of Computer Science, University of California, Los Angeles, Los Angeles, CA 90095, United States; emails: atefehsz@cs.ucla.edu, hyu@cs.ucla.edu, cong@cs.ucla.edu; M. Gao, Falcon Computing Solutions Inc., 3979 Freedom Circle Suite 530 Santa Clara, CA 95054 United States; email: mingao@falcon-computing.com.

1084-4309/2022/02-ART32 $15.00
https://doi.org/10.1145/3494534

1 INTRODUCTION

Due to the rapid growth of datasets in recent years, the demand for scalable, high-performance computing continues to increase. However, the breakdown of Dennard's scaling [19] has made the energy efficiency an important concern in datacenters, and has spawned exploration into using accelerators such as **field-programmable gate arrays (FPGAs)** to alleviate power consumption. For example, Microsoft has adopted CPU-FPGA systems in its datacenter to help accelerate the Bing search engine [38]; Amazon introduced the F1 instance [2], a compute instance equipped with FPGA boards, in its commercial **Elastic Compute Cloud (EC2)**.

Although the interest in customized computing using FPGAs is growing, they are more difficult to program compared to CPUs and GPUs because the traditional **register-transfer level (RTL)** programming model is more like circuit design rather than software implementation. To improve the programmability, **high-level synthesis (HLS)** [13, 59] has attracted a large amount of attention over the past decades. Currently, both FPGA vendors have their commercial HLS products—Xilinx Vitis [53] and Intel FPGA SDK for OpenCL [27]. With the help of HLS, one can program the FPGA more easily by controlling how the design should be synthesized from a high-level view. The main enabler of this feature is the ability to iteratively re-optimize the micro-architecture quickly just by inserting synthesis directives in the form of pragmas instead of re-writing the low-level behavioral description of the design. Because of the reduced code development cycle and the shorter turn-around times, HLS has been rapidly adopted by both academia and industry [3, 20, 30, 45, 49, 65]. In fact, Code 1 shows an intuitive HLS C implementation of one forward path of a **Convolutional Neural Network (CNN)** on Xilinx FPGAs. Xilinx Vitis generates about 5,800 lines of RTL kernel from Code 1 with the same functionality. As a result, it is much more convenient and productive for designers to evaluate and improve their designs in HLS C/C++.

Even though HLS is suitable for hardware experts to quickly implement an optimal design, it is not friendly for most of the general software designers who have limited FPGA domain knowledge. Since the hardware architecture inferred from a syntactic C implementation could be ambiguous, current commercial HLS tools usually generate architecture structures according to specific HLS C/C++ code patterns. As a result, even though it was shown in [13] that the HLS tool is capable of generating FPGA designs with a performance as competitive as the one in RTL, not every C program gives a good performance and designers must manually reconstruct the HLS C/C++ kernel with specific code patterns and hardware specific pragmas to achieve high performance. As a matter of fact, the generated FPGA accelerator from Code 1 is 80× slower than a single-thread CPU. However, the optimized code (shown in Code 3 in Appendix A.1) is able to achieve around 7,041× speedup after we analyze and resolve several performance bottlenecks listed in Table 1 by applying code transformations and inserting 28 pragmas.

It turns out that the bottlenecks presented in Table 1 occur for most C/C++ programs developed by software programmers, and similar optimizations have to be repeated for each new application, which makes the HLS C/C++ design not scalable. In general, there are three levels of optimization that one needs to employ to get to a high-performance FPGA design. The level one is for increasing the data reuse or reducing/removing the data dependency by *loop* transformations, which is common in CPU performance optimizations as well (e.g., for cache locality); therefore, it is well accepted by software programmers and we expect them to apply such transformations manually without any problems. The second level is required to enable repetitive architectural optimizations that most of the designs benefit from, such as memory burst and memory coalescing, as mentioned in reasons 1-2 in Table 1. Fortunately, the recently developed Merlin Compiler[1] [11, 12, 21] from

[1]The Merlin Compiler is open-sourced at https://github.com/Xilinx/merlin-compiler.

Table 1. Analysis of Poor Performance in Code 1

	Reason	Required Code Changes for Higher Performance
①	Low bandwidth util.	Manually apply memory coalescing using HLS built-in type `ap_int`.
②	High access latency to global memory	Manually allocate local buffer and use memcpy to enable memory burst.
③	Does not hide communication latency	Manually create load/compute/store functions and double buffering.
④	Lack of parallelism	Manually create parallel coarse-grained processing elements by wrapping the inner loops as a function and setting proper array partition factors.
⑤	Sequential execution	Apply `#pragma HLS pipeline` and `#pragma HLS unroll` with proper array partition factors for each processing element.

Code 1. CNN HLS C Code Snippet

```
// Skip const variable initialization and macro definitions for brevity
void CnnKernel(const float* input ①, const float* weight①,
               const float* bias ①, float* output ①) {

  float C[ParallelOut][ImSize][ImSize];
  for (int i = 0; i < NumOut / ParallelOut; ++i) { ④
    // Initialization
    for (int h = 0; h < ImSize; ++h) {
      for (int w = 0; w < ImSize; ++w) {
        for (int po = 0; po < ParallelOut; po++)
          C[po][h][w] = bias[(i << shift) + po]; } }
    // Convolution
    for (int j = 0; j < NumIn; ++j) { ⑤
      for (int h = 0; h < ImSize; ++h) { ⑤
        for (int w = 0; w < ImSize; ++w) { ⑤
          for (int po = 0; po < ParallelOut; po++) { ⑤
            for (int p = 0; p < kKernel; ++p) { ⑤
              for (int q = 0; q < kKernel; ++q) ⑤
                C[po][h][w] += weight(i, po, j, p, q) * input(j,h + p,w + q); ② ③ } } } } }
    // ReLU + Max pooling
    for (int h = 0; h < OutImSize; ++h) { ⑤
      for (int w = 0; w < OutImSize; ++w) { ⑤
        for (int po = 0; po < ParallelOut; po++) { ⑤
          output(i,h,w) = max(0.f, C, po, h, w); } } } } }
```

Falcon Computing Solutions [21], which was acquired by Xilinx in late 2020 [51], can automatically take care of these kinds of code transformations.

The final and the most critical level deals with FPGA-specific architectural optimizations, detailed in reasons 3-5 in Table 1, that vary from application to application. Although the Merlin Compiler also helps alleviate this problem to some extent by introducing a few high-level optimization pragmas and applying source-to-source code transformation to enable them, these optimizations are much more difficult for software programmers to learn and apply effectively. More specifically, choosing the right part of the program to optimize, deciding the type of optimization and the pragmas to apply for enabling it, and tuning the pragma to get to the design with the highest quality complicate this level.

Apparently, the requirement of mastering all three levels of optimizations makes the bar for general software programmers to use FPGA extremely high. Hence, general software programmers will lean towards other popular accelerators such as power-consuming GPUs or high-cost ASICs with less consideration over FPGAs. These obstacles consequently result in huge barriers in the adoptions of FPGA in datacenters, the expansion of the FPGA user community, and the advances of

FPGA technology. One possible solution is to apply an automated micro-architecture optimization. Thus, everyone with decent knowledge of programming is able to try customized computing with minimum effort. In order to free accelerator designers from the iterations of HLS design improvement, automated **design space exploration (DSE)** for HLS attracts more and more attention. However, existing DSE methods face the following challenges:

Challenge 1: The large solution space: The solution space grows exponentially by the number of candidate pragmas. In fact, only applying pipeline, unroll, and array partition pragmas to Code 1 produces 10^{20} design points. This huge number of combinations creates a serious impediment to exploring the whole design space.

Challenge 2: Non-monotonic effect of design parameters on performance/area: As pointed out by Nigam, et al. [35], we cannot assume that an individual design parameter will affect the performance/area in a smooth and/or monotonic way.

Challenge 3: Correlation of different characteristics of a design: When different pragmas are employed together in a design, they do not affect only one characteristic of a design. We will use the convolution part of the Code 1 as an example. If we apply **fine-grained (*fg*)** pipeline to `w` loop and parallelize the loop with a factor of 2, it results in a loop with **initiation interval (II)** of 2 when synthesized by Vivado HLS [48]. However, when we change the parallel factor to 4, the HLS tool increases the II to 3 to optimize resource consumption by reusing some of the logic units instead of doubling the resource utilization. The analytical models usually fail to capture these cases. Furthermore, pipelining the `j` loop is part of the best design configuration. However, it does not improve the performance until after the *fg* pipelining is applied on the `w` loop. It suggests that the order of applying the pragmas is crucial in designing the explorer.

Challenge 4: Implementation disparity of HLS tools: The HLS tools from different vendors employ different implementation strategies. Even within the same vendor, the optimization and implementation rules keep changing across different versions. For example, the past Xilinx SDAccel versions consistently utilize *registers* to implement array partitions with small sizes to save BRAMs. However, the latest ones use *dual-port BRAMs* for implementation to support two reads in one cycle for achieving full pipelining, or II = 1, even if the array size is small. Such implementation details are hard to capture and maintain in analytical models and make it difficult to port an analytical model built on a specific tool to the other.

Challenge 5: Long synthesis time of HLS tools: HLS tools usually take 5-30 minutes to generate RTL and estimate the performance—and even longer if the design has a high performance. This emphasizes the need for a DSE that can find the Pareto-optimal design points in fewer iterations.

In this paper, as our first step to lowering the bar for general software programmers to make the FPGA programming universally accessible, we focus on automating the final level of optimization. To solve the challenges 2 to 4 mentioned above, instead of developing an analytical model, we treat the HLS tool as a black-box. Challenges 1 and 5 imply that we need to explore the solution space intelligently. For that, we first apply the coordinate descent with the finite difference method to guide the explorer. However, we show that the general application-oblivious approaches fail to perform well for the HLS DSE problem. As a result, we present the *AutoDSE*[2] framework that adapts a bottleneck-guided coordinate optimizer to systematically search for better configurations. Bottleneck-guided optimization approaches have been used successfully for CPU and GPU compilation optimization [25, 36], however the runtime for evaluation of the optimized code on CPUs and GPUs is much shorter. To speed up the process, we incorporate a flexible list-comprehension syntax to represent a grid design space with all invalid points

[2]All the codes are open-sourced at https://github.com/UCLA-VAST/AutoDSE.

marked which can help us prune the design space. In addition, we also partition the design space systematically to address the local optimum problem caused by Challenge 2.

In summary, this paper makes the following contributions:

- We propose two strategies to guide DSE. One adapts the commonly used coordinate descent with the finite difference method and the other exploits a bottleneck-guided coordinate optimizer.
- We incorporate list-comprehension to represent a smooth, grid design space with all invalid points marked.
- We develop the *AutoDSE* framework on top of the Merlin Compiler to automatically perform DSE using the bottleneck optimizer to systematically close in on high-QoR design points.
- To the best of our knowledge, we are the first ones to evaluate our tool using the Xilinx optimized vision library [52]. Evaluation results indicate that *AutoDSE* is able to achieve the same performance, yet with 26.38× reduction of their optimization pragmas resulting in less than one required optimization pragma per kernel, on the geometric mean.
- We evaluate *AutoDSE* on 11 computational kernels from MachSuite [39] and Rodinia [8] benchmarks and one convolution layer of Alexnet [29], showing that we are able to achieve, on the geometric mean, 19.9× speedup over a single-thread CPU—only a 7% performance gap compared to manual designs.

2 PROBLEM FORMULATION

Our goal is to expedite the hardware design by automating its exploration process. In general, there are two types of pragmas (using Vivado HLS as an example) that are applied to a program. One type is the *non-optimization* pragmas, which are relatively easy for software programmers to learn and apply. The other type is *optimization* pragmas, including `PIPELINE` and `UNROLL` pragmas. These pragmas require knowledge of FPGA devices and micro-architecture optimization experience, which are usually much more challenging for a software programmer to learn and master as explained in Section 1. The goal of this research is to minimize or eliminate the need to apply optimization pragmas *manually* and let *AutoDSE* insert them *automatically*. More formally, we formulate the HLS DSE problem as the following:

Problem 1: Identify the Design Space. Given a C program $\mathcal{P}$ as the FPGA accelerator kernel, construct a design space $\mathbb{R}^K_{\mathcal{P}}$ with K parameters that contains possible combinations of HLS pragmas for $\mathcal{P}$ as design configurations.

Problem 2: Find the Optimal Configuration. Given a C program $\mathcal{P}$, we would like to insert a minimal number of optimization pragmas manually to get a new program $\mathcal{P}'$ as the FPGA accelerator kernel along with its design space set $\mathbb{R}^K_{\mathcal{P}'}$, which is identified in Problem 1, and we let the DSE tool insert the rest of the pragmas automatically. More specifically, having a vendor HLS tool $\mathbf{H}$ that estimates the execution cycle $Cycle(\mathbf{H}, \mathcal{P}')$ and the resource utilization $Util(\mathbf{H}, \mathcal{P}')$ of the given $\mathcal{P}'$ as a black-box evaluation function, the DSE must find a configuration $\theta \in \mathbb{R}^K_{\mathcal{P}'}$ in a given search time limit so that the generated design $\mathcal{P}'(\theta)$ with θ can fit in the FPGA and the execution cycle is minimized. Formally, our objective is:

$$\min_{\theta} Cycle(\mathbf{H}, \mathcal{P}'(\theta)) \tag{1}$$

subject to

$$\begin{gathered}\theta \in \mathbb{R}^K_{\mathcal{P}'} \\ \forall u \in Util(\mathbf{H}, \mathcal{P}'(\theta)), u < T_u\end{gathered} \tag{2}$$

where u is the utilization of one of the FPGA on-chip resources and T_u is a user-available resource threshold on FPGAs. We set all T_u to 0.8, an empirical threshold, in our experiments. Beyond 0.8, the design will suffer from high clock frequency degradation due to the difficulty in placement and routing. In addition, the rest of the resources are left for the interface logic of the vendor HLS tool.

Note that we introduce two optimization objectives; one minimizes the optimization pragmas that have to be inserted manually to obtain $\mathcal{P}'$, and the other maximizes the performance of $\mathcal{P}'$ using *AutoDSE* by applying pragmas automatically. Obviously, there is a trade-off between the two. An expert designer can always get an optimized micro-architecture to achieve the best performance by inserting enough HLS optimization pragmas. However, it is time-consuming and not feasible for software programmers with little or no FPGA design experience. In our evaluation, our goal is to match the performance of well-designed HLS library code (typically written by experts) yet insert much fewer optimization pragmas *manually*. Indeed, our experimental results in Section 6 show that we can achieve this with 26.38× pragma reduction on the geometric mean, requiring less than one optimization pragma per kernel.

3 RELATED WORK

There are a number of previous works that propose an automated framework to explore the HLS design space, and they can be summarized in two categories: model-based and model-free techniques.

3.1 Model-based Techniques

The studies in this category build an analytical model for evaluating the quality of each explored design point by estimating its performance and resource utilization. The authors in [50, 60, 62] build the dependence graph of the target application and utilize traditional graph analysis techniques along with predictive models to search for the best design. Although this approach can quickly search through the design space, it is inaccurate and it is difficult to maintain the model and port it to other HLS tools as explained in Challenge 4 of Section 1. Zhong, et al. [64] develops a simple analytical model for performance and area estimation. However, they assume that the performance/area changes monotonically by modifying an individual design parameter, which is not a valid assumption as we explained in Challenge 2 of Section 1. To increase the accuracy of the estimation model, a number of other studies restrict the target application to those that have a well-defined accelerator micro-architecture template [9, 14, 15, 40, 45, 58], a specific application [55, 61], or a particular computation pattern [10, 28, 37]; hence, they lose generality.

To the same end, there are other studies that build the predictive model using learning algorithms. They train a model by iteratively synthesizing a set of sample designs and updating the model until it gets to the desired accuracy. Later on, they use the trained model for estimating the quality of the design instead of invocations of the HLS tool. To learn the behavior of the HLS tool, these works adapt supervised learning algorithms to better capture uncertainty of the HLS tools [28, 31, 32, 43, 56, 63]. While this technique increases the accuracy of the model, it is still hard to port the model to another HLS tool in a different vendor or version. Often by changing the HLS tool or the target FPGA, new samples should be collected which can be an expensive step. After that, for each of them, a new model should be trained to include the new dataset.

3.2 Model-free Approaches

To avoid dealing with the uncertainty of the HLS tools, in this category, the studies treat the HLS tool as a black box. Instead of learning a predictive model, they invoke the HLS tool every time to evaluate the quality of the design. To guide the search, they either exploit general application-oblivious heuristics (e.g., simulated annealing [33] and genetic algorithm [41]) or they develop

Table 2. Merlin Pragmas with Architecture Structures

Keyword	Available Options	Architecture Structure
parallel	factor=<int>	CG & FG parallelism
pipeline	mode=cg	CG pipeline
	mode=fg	FG pipeline
tiling	factor=<int>	Loop Tiling

CG: Coarse-grained; FG: Fine-grained.

their own heuristics [22, 23, 42]. S2FA [57] employ multi-armed bandit [24] to combine a set of heuristic algorithms including uniform greedy mutation, differential evolution genetic algorithm, particle swarm optimization, and simulated annealing. However, as we will present in Section 5.1, general hyper-heuristic approaches are unreliable for finding the high **quality of result (QoR)** design configuration. Moreover, the authors in [22, 23] claim that Pareto-optimal design points cluster together. They exploit an initial sampling to build the first approximation of the Pareto frontier and require local searches to explore other candidates. However, the cost of initial sampling is not scalable when the design space is tremendously large (e.g., the scale of 10^{10} to 10^{30}), as the ones we have enumerated in this paper are. Sun, et al. [46] adapt a **(Gaussian process) GP**-based **Bayesian optimization (BO)** algorithm to explore the solution space. At each iteration, it improves a surrogate model to mimic the HLS tool, by sampling the design space. Again, as the search space grows, it will require more samples to build a good surrogate model which can limit the scalability. Moreover, the computation of a GP-based BO can be seen to be cubic in the total number of samples (in addition to the time to evaluate the sampled point using the HLS tool), as it wants to calculate the inversion of a dense covariance matrix at each step [44] which can further limit the scalability of the approach.

4 THE AUTODSE FRAMEWORK

To reduce the size of the design space, we build our DSE on top of the Merlin Compiler [11, 12, 21]. Section 4.1 reviews the Merlin Compiler and justifies our choice. Then, we present an overview of *AutoDSE* in Section 4.2.

4.1 Merlin Compiler and Design Space Definition

The Merlin Compiler [11, 12, 21] was developed to raise the abstraction level in FPGA programming by introducing a reduced set of high-level optimization directives and generating the HLS code according to them automatically. It uses a simple programming model similar to OpenMP [17], which is commonly used for multi-core CPU programming. Like in OpenMP, it defines a small set of compiler directives in the form of pragmas for optimizing the design. Table 2 lists the Merlin pragmas with architecture structures. Note that the `fg` option in the fine-grained pipeline mode refers to the code transformation that tries to apply fine-grained pipelining to a loop nest by fully unrolling all its sub-loops; whereas, the `cg` option in the coarse-grained pipelining transforms the code to enable double buffering. Based on these user-specified pragmas, the Merlin Compiler performs source-to-source code transformation and automatically generates the related HLS pragmas such as `PIPELINE`, `UNROLL`, and `ARRAY_PARTITION` to apply the corresponding architecture optimizations.

To reduce the size of the solution space, we chose to utilize the Merlin Compiler as the backend of our tool. Since the number of pragmas required by the Merlin Compiler is much smaller (as it performs source level code reconstruction and generates most of the HLS required pragmas), it defines a more compact design space, which makes it a better fit for developing

Code 2. CNN Code Snippet in Merlin C

```
void CnnKernel(
       const float input[NumIn][InImSize][InImSize],
       const float weight[NumOut][NumIn][kKernel][kKernel],
       const float bias[NumOut], float output[NumOut][OutImSize][OutImSize]) {

  float C[ParallelOut][ImSize][ImSize];
  for (int i = 0; i < NumOut/ParallelOut; i++) {
    // Initialization
          for (int h = 0; h < ImSize; ++h) {
#pragma ACCEL parallel factor=4
            for (int w = 0; w < ImSize; ++w){
              for (int po = 0; po < ParallelOut; po++)
                        C[po][h][w] = 0.f; } }
    // Convolution
#pragma ACCEL pipeline
          for (int j = 0; j < NumIn; ++j) {
            for (int h = 0; h < ImSize; ++h) {
#pragma ACCEL parallel factor=4
#pragma ACCEL pipeline FLATTEN
              for (int w = 0; w < ImSize; ++w) {
                      for (int po = 0; po < ParallelOut; po++){
                        float tmp = 0;
                              for (int p = 0; p < kKernel; ++p) {
                                for (int q = 0; q < kKernel; ++q){
                                        tmp += ... } }
                              C[po][h][w] += tmp; } } } }
    // Skip ReLU + Max Pooling for brevity
} }
```

a DSE as shown in [15, 57]. For instance, Code 2 shows the CNN kernel with Merlin pragmas. With inserting only four lines of pragmas and no further *manual* code transformation, the Merlin Compiler is able to transform Code 2 to a high-performance HLS kernel with the same performance as the manually optimized design written in HLS C which has 28 pragmas (Section A.1).

The Merlin Compiler, by default, applies code transformations to address the bottlenecks 1 and 2 listed in Table 1 and provides high-level optimization pragmas for the rest of them. For example, instead of rewriting Code 1 to test whether double buffering would help the performance as described in reason 3 in Table 1, we just need to use the cg `PIPELINE` pragma and the Merlin Compiler will rewrite the code to satisfy it. As a result, our focus in this work is on finding the best location of each of these high-level pragmas and tuning them, automatically; hence, we can address reasons 3-5 in Table 1 as well by enabling the architectural optimizations along with the best pipelining and parallelization attributes. Consequently, our solution to Problem 1 is defined as in Table 3. We identify the design space for each kernel by analyzing the kernel **abstract syntax tree (AST)** to gather loop trip-counts, available bit-widths, etc. The rules we enforce in building this design space are listed in Section 5.4.

Now that we have defined the design space in Table 3 for ***Problem 1***, we focus on ***Problem 2*** in the remainder of this paper. Although to some extents, Merlin pragmas alleviate the manual code reconstruction overhead, a designer still has to manually search for the best option for each pragma, including position, type, and factors. In fact, choices for the CNN design in Code 1 contain four DRAM buffers and thirteen loops, which result in $\sim 10^{16}$ design configurations. The large design space motivates us to develop an efficient approach to find the best configuration.

4.2 Framework Overview

We develop and implement *AutoDSE*, a push-button framework, as depicted in Figure 1 based on the strategies explained in Section 5. The framework first automatically builds a design space by

Table 3. Design Space Building on Merlin Pragmas

Factor	Design Space (Values)
CG-loop parallel	$\{u \mid 1 < u <= TC(L), u.c = TC(L), c \in \mathbb{Z}\}$
FG-loop parallel	$\left\{u \mid \begin{cases} 1 < u < TC(L),\ u.c = TC(L),\ c \in \mathbb{Z}, & \text{TC(L)} > 16 \\ u = TC(L), & \text{otherwise} \end{cases}\right\}$
CG-loop pipeline	$\{p \mid p \in \{off, cg, fg\}\}$
FG-loop pipeline	$\{p \mid p = fg\}$
loop tiling	$\{t \mid 1 < t < TC(L), t.c = TC(L), c \in \mathbb{Z}\}$

CG: Coarse-grained; FG: Fine-grained; TC: Loop trip-count.

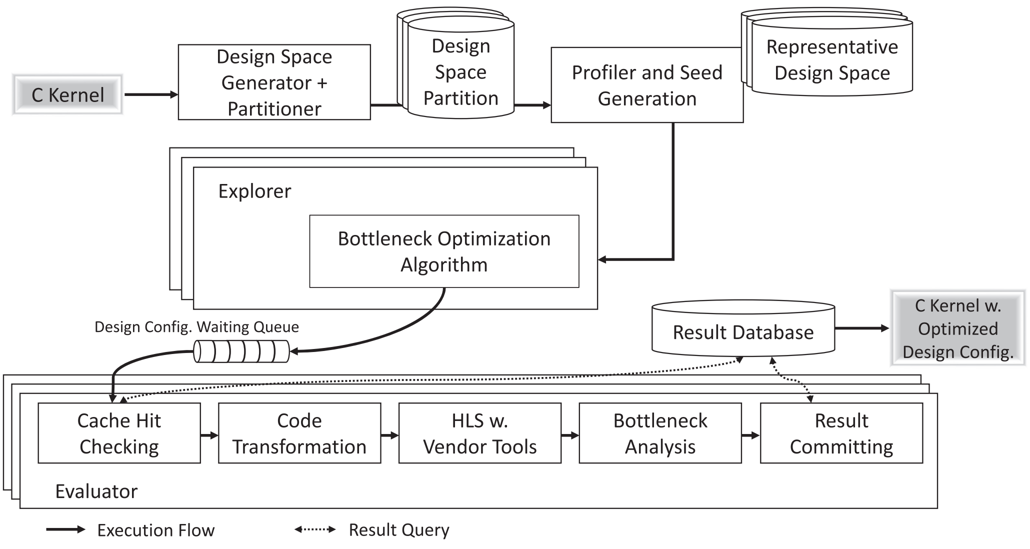

C Kernel
Design Space Generator + Partitioner
Design Space Partition
Profiler and Seed Generation
Representative Design Space
Explorer
Bottleneck Optimization Algorithm
Design Config. Waiting Queue
Result Database
C Kernel w. Optimized Design Config.
Cache Hit Checking
Code Transformation
HLS w. Vendor Tools
Bottleneck Analysis
Result Committing
Evaluator
Execution Flow
Result Query

Fig. 1. The *AutoDSE* framework overview.

analyzing the kernel AST according to the rules and the syntax described in Section 5.4. Then, it profiles and selects representative partitions using K-Means as mentioned in Section 5.5. For each partition, *AutoDSE* explorer performs DSE using the proposed bottleneck-based coordinate strategy in Section 5.2 and the parameter ordering explained in Section 5.3. The explorer can be tuned to evaluate the quality of design points based on different targets such as performance, resource, or finite difference (Equation (5)). When the explorer finishes exploring a partition, it stores the best configuration found by that partition and reallocates the working threads to other partitions to keep the resource utilization high. Finally, when all partitions are finished, *AutoDSE* outputs the design configuration with the best QoR among all partitions.

5 AUTODSE METHODOLOGY

In this section, we first examine the efficiency of application-oblivious heuristics, which were considered in our initial study, in Section 5.1. As we will discuss, the main drawback of these heuristics for the HLS DSE problem is the fact that they do not have any knowledge of the semantics of the program parameter. This problem can potentially linger the DSE process since the explorer may waste a lot of time on parameters with no impact on the results at that stage of optimization. As a result, in Section 5.2, we present a bottleneck-guided coordinate optimizer that can mimic an

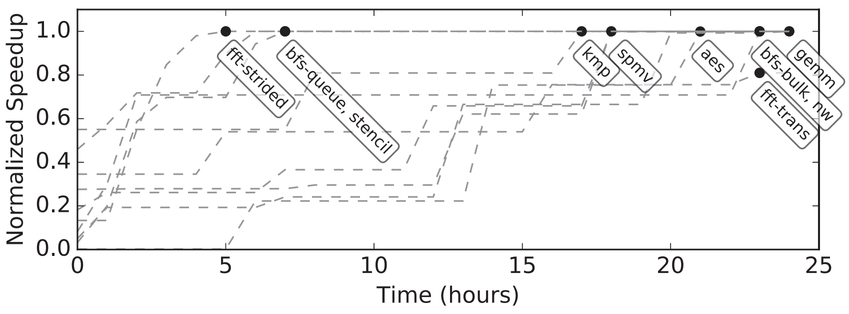


Fig. 2. Speedup over the manual design using S2FA [57].

expert's optimization method and generate high-QoR design points in fewer numbers of iterations. We propose several optimizations in Sections 5.3 to 5.5 to further improve the performance of our framework.

5.1 Application-oblivious Heuristics

In our prior work on DSE, S2FA [57], we adapted a popular search engine called OpenTuner [4]. OpenTuner leverages the **multi-armed bandit (MAB)** approach [24] to assemble multiple meta-heuristic algorithms - including uniform greedy mutation, differential evolution genetic algorithm, particle swarm optimization, and simulated annealing - for high generalization. At each iteration, the MAB selects the meta-heuristic with the highest credit and updates the credit of the selected meta-heuristic based on the QoR, which means the meta-heuristic that can efficiently find high-quality design points will be rewarded and activated more frequently by the MAB, and vice versa. Due to its extensibility, OpenTuner has been adapted to perform DSE for hardware design optimization [54].

S2FA also employs the Merlin Compiler as its backend and further applies more strategies to improve the OpenTuner's efficiency when performing DSE for HLS. We use S2FA to perform the DSE for 24 hours and depict the speedup of our benchmark cases compared to the corresponding manual design over time in Figure 2. The black dot indicates the time that the S2FA finds the overall best design point. We can see that S2FA requires on average 16.8 hours to find the best solution. We further analyze the exploration process and find that most designs have an obvious performance bottleneck (e.g., low utilization of global memory bandwidth, insufficient parallel factors, etc.), which usually dominates more than half of the overall cycle counts and is controlled by only one or two design parameters (pragmas). In this situation, the performance gain of tuning other parameters is often very limited but it is hard for the problem-independent searching algorithm to learn that. In fact, it needs many iterations to identify the key parameter and tune it to resolve the performance bottleneck. After that, it has to spend a large number of iterations again to find the next key parameter. This phenomenon motivates us to develop a new search algorithm that is guaranteed to optimize the key parameter (high-impact parameter) prior to others.

Coordinate descent is another well-known iterative optimization algorithm for finding a locally minimum point. It is based on the idea that one can minimize a multi-variable function by minimizing it along one direction at a time and solving single-variable optimization problems. At each iteration, we generate a set of candidates, Θ_{cand}, as the input to the algorithm. Each candidate is generated by advancing the value of each parameter in the current configuration by one step. Formally, the c-th candidate generated from design point θ_i is:

$$\theta_i^c = [p_0, p_1, \ldots, p_c + 1, \ldots, p_K] \tag{3}$$

where K is the total number of parameters, p_c is the value of c-th parameter in θ_i, $p_c + 1$ denotes the next value of this parameter (the next numeric factor for `PARALLEL` and `TILING` pragma and the next mode of pipelining for `PIPELINE` pragma). Accordingly, we will generate K candidates at each iteration, which means we run the HLS tool K times to determine the next configuration as follows:

$$\theta_{i+1} = \underset{\theta_i^c \in \Theta_{cand}}{argmin}\, g(\theta_i^c, \theta_i) \tag{4}$$

We leverage the finite difference method to approximate the coordinate value by treating the HLS tool as a black-box. That is, given a candidate configuration θ_j deviated from the current configuration θ_i, the coordinate value is defined as:

$$g(\theta_j, \theta_i) \sim \frac{Cycle(\mathbf{H}, \mathcal{P}(\theta_j)) - Cycle(\mathbf{H}, \mathcal{P}(\theta_i))}{Util(\mathbf{H}, \mathcal{P}(\theta_j)) - Util(\mathbf{H}, \mathcal{P}(\theta_i))} \tag{5}$$

We calculate $Util(\mathbf{H}, \mathcal{P}(\theta))$ by taking into account all the different types of resources using the following formula:

$$Util(\mathbf{H}, \mathcal{P}(\theta)) = \sum_u 2^{\frac{1}{1-u}} \tag{6}$$

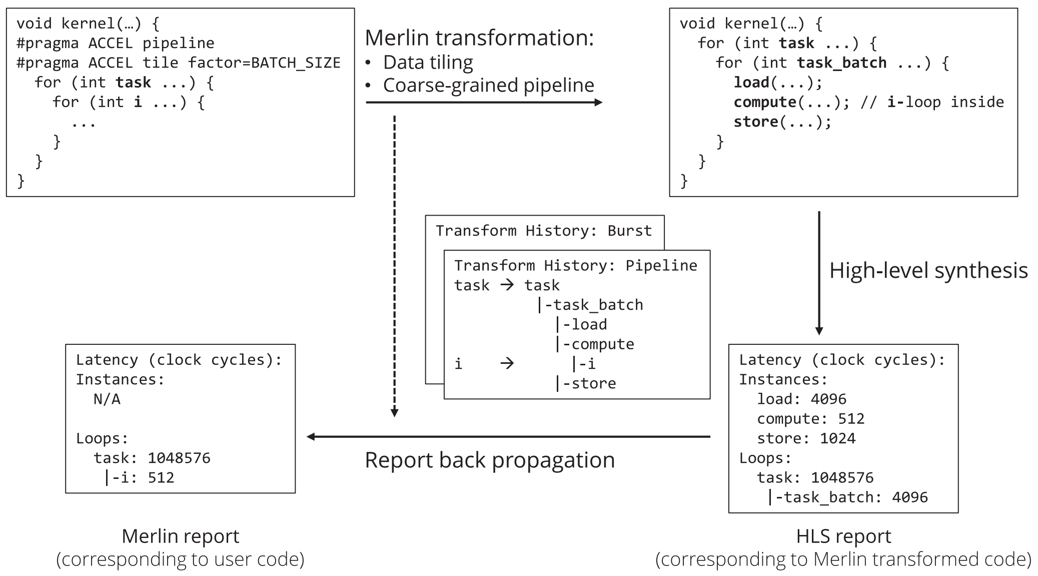
where u is the utilization of one of the FPGA resources. We use exponential function to penalize the over-utilization of FPGA more seriously. Note that Equation (5) considers not only performance gain but also resource efficiency, so it could reduce the possibility of being trapped in a local optimum. For example, we may reduce 10% execution cycle by spending 30% more area if we increase the parallel factor of a loop (configuration θ_1); we can also reduce 5% execution cycle by spending 10% more area if we enlarge the bit-width of a certain buffer (configuration θ_2). Although θ_1 seems better in terms of the execution cycle, it may be more easily trapped by a locally optimal point because it has a relatively limited resource left to be further improved. On the other hand, the finite difference values for the two configurations are $g(\theta_1, \theta_0) = \frac{-10\%}{30\%} = -0.3$ and $g(\theta_2, \theta_0) = \frac{-5\%}{10\%} = -0.5$, so the system prioritizes the second configuration for a better long-term performance.

By leveraging the coordinate descent with a finite difference method, we expect to find a better design point every K HLS runs. Unfortunately, as mentioned in Challenge 2 of Section 1, the performance trend is not always smooth, so the coordinate process can easily be trapped by a low-quality locally optimal design point. Actually, this approach only achieves 2.8× speedup, on the geometric mean, for our **MachSuite** [39] and **Rodinia** [8] **(MR)** benchmarks, which is even worse than the results from S2FA.

Moreover, the efficiency of using the coordinate-based approach for DSE is limited by the number of parameters. More specifically, at each iteration, we need to evaluate K design points, where K is the total number of tuning parameters, to determine the next step. On the other hand, in most cases, only a few of the K tuning parameters have a high impact on the performance, so we should evaluate only the K' impactful parameters at each iteration where $K' < K$. For instance, design space generator will instrument Code 1 with 27 pragmas based on the rules explained in Section 5.4 and the coordinate-based approach proposed in this section needs to assess the quality of 27 new designs in each iteration. However, in the early iterations the convolution part takes more than 85% of the total cycle counts of the kernel. As a result, changing the pragmas outside of this part will have insignificant effect on the performance; hence, it is wasteful to explore them at this stage.

5.2 AutoDSE Exploring Strategy - Bottleneck-guided Coordinate Optimizer

Two main inefficiencies of the approaches reviewed in the previous section are: (1) they must evaluate many design points to identify the performance bottleneck, (2) they have no knowledge



```
void kernel(…) {
#pragma ACCEL pipeline
#pragma ACCEL tile factor=BATCH_SIZE
  for (int task ...) {
    for (int i ...) {
      ...
    }
  }
}
```

```
void kernel(…) {
  for (int task ...) {
    for (int task_batch ...) {
      load(...);
      compute(...); // i-loop inside
      store(...);
    }
  }
}
```

```
Transform History: Burst
```

```
Transform History: Pipeline
task → task
         |-task_batch
           |-load
           |-compute
i    →       |-i
           |-store
```

```
Latency (clock cycles):
Instances:
  N/A

Loops:
  task: 1048576
   |-i: 512
```

```
Latency (clock cycles):
Instances:
  load: 4096
  compute: 512
  store: 1024
Loops:
  task: 1048576
   |-task_batch: 4096
```

Fig. 3. The cycle propagation in the Merlin Compiler.

of the semantics of the parameters, so they have no way to differentiate them and prioritize the important ones. Identifying the key parameters is not straightforward. Although the HLS report may provide the cycle breakdown for the loop and function statements, it is hard to map them to tuning parameters due to the applications of several code transformations applied by the Merlin Compiler. Fortunately, the Merlin Compiler includes a feature that transmits the performance breakdown reported by the HLS tool to the user input code, allowing us to identify the performance bottleneck by traversing the Merlin report and mapping the bottleneck statement to one or few tuning parameters.

Figure 3 illustrates how the Merlin Compiler generates its report of the cycle breakdown. When performing code transformation, the Merlin Compiler records the code change step by step so that it is able to propagate the latency estimated by the HLS tool back to the user input code. In this example, the `i` loop corresponds to the `compute` unit in the transformed code, so the latency of this unit is assigned to it. Note that the latency of all `load`, `compute`, and `store` units are included in the `task_batch` loop which will determine the latency of `task` loop in both the original and transformed codes. This feature is helpful for us to analyze the performance bottleneck and identify the key tuning parameter by running HLS once at each iteration instead of evaluating the effect of all K parameters.

By exploiting the cycle breakdown, we can develop a bottleneck analyzer to resolve the issues mentioned above. We first build a map from the loop or function statements in the user input code to design parameters so that we know which parameters should be focused on for a particular statement. To identify the critical path and type, we start with the kernel top function statement and build hierarchy paths of the design by traversing the Merlin report using **depth-first search (DFS)**. More specifically, for each hierarchy level, we first check to see if the current statement has child loop statements and sort them by their latency. Then, we traverse each of the child loops and repeat this process. In case of a function call statement, we dive into the function implementation to further check its child statements for building the hierarchy paths. Finally, we return a list of paths in order. Note that since we sort all loop statements according to their latency by checking the Merlin report, the hierarchy paths we created will also be sorted by their latency.

Subsequently, for each statement, we check the Merlin report again to determine whether its performance bottleneck is memory transfer or computation. The Merlin Compiler obtains this information by analyzing the transformed kernel code along with the HLS report. A cycle is considered to be a memory transfer cycle if it is consumed by communicating to global memory. As a result, we can not only figure out the performance bottleneck for each design point, but also identify a small set of effective design parameters to focus on. Therefore, we are able to significantly improve the efficiency of our searching algorithm.

When we obtain an ordered list of critical hierarchy paths from the bottleneck analyzer, we start from the innermost loop statement (because of the DFS traversal) of the most critical entry and identify its corresponding parameters as candidate parameters to explore, if they are not already tuned. Based on the bottleneck type, provided by the bottleneck analysis, (i.e., memory transfer or computation), we pick a subset of the parameters mapped to that statement to work on. For example, we may have design parameters of `PARALLEL` and `TILING` at the same loop level. When the bottleneck type of the loop is memory transfer, we focus on the `TILING` parameter for the loop; otherwise, we focus on `PARALLEL` parameter. In other words, we reduce the number of candidate design parameters not only by the bottleneck statement but also by the bottleneck type.

We define each design point as a data structure containing the following information:

```
curr_point = DesignPoint(configuration, tuned, result, quality, children)
```

where *configuration* contains the value of all the parameters and *tuned* lists the parameters which the algorithm has explored for the current point. *quality* stores the quality of design measured by finite difference value and *result* includes all the related information gathered from the HLS tool including the resource utilization and the cycle count. Finally, each design point stores a stack of the configurations for its unexplored children where each child is generated by advancing one of the parameters by one step. The children are pushed to the stack in the order of their importance (from least to most important) as computed by the bottleneck analyzer so that by popping the stack, we get to work with the child who has changed the parameter with the most promising impact.

We define level n as a point where we have fixed the value of n parameters, so the maximum level in our algorithm is equal to the total number of parameters. For each level, we define a heap of the pending design points that can be further explored and push the design points by their *quality* into the heap. Since new design points are sorted by their quality values when they were pushed into the heap, the design point with a better quality value will be chosen for tuning more of its parameters prior to other points. As mentioned above, the next point to be explored is chosen by popping the stack of the unexplored children of this design point so that at each step, we get to evaluate the most promising design point.

Algorithm 1 presents our exploring strategy. As we will explain in Section 5.5, we partition the design space to alleviate the local optimum problem. For each partition, we first get its default point and initialize the heap of the first level (lines 3 to 9). Then, at each iteration of the algorithm, *AutoDSE* gets the heap with the highest level, peeks the first node of the heap, and pops its stack of unexplored children to get the new candidate (lines 11 to 14). Next, each option of the new focused parameter will be evaluated and the result will be passed to the bottleneck analyzer to generate a new set of focused parameters for making new children (lines 16 to 21). Since the number of fixed parameters is increased by one, it will be pushed to the heap of the next level if there is still a parameter left that has not been tuned yet (lines 22 to 26). When the stack of unexplored children of the current design point is empty, it will be popped out of heap (lines 28 to 30). The algorithm continues either until all the heaps are empty or when the DSE has reached a runtime threshold (Line 10).

ALGORITHM 1: *AutoDSE* Explorer: Bottleneck-guided Coordinate Optimizer

Require: A C program $\mathcal{P}$ and a set of design space partitions $\mathbb{P}$.
Ensure: A design configuration θ with the best QoR.

```
1: $top\_func \leftarrow GetTopFunction(\mathcal{P})$
2: parallel for $P \in \mathbb{P}$ do
3:     $best\_cfg = cfg \leftarrow GetDefaultPoint(P)$
4:     $report, hier \leftarrow Evaluate(cfg)$
5:     $parameter\_order \leftarrow BottleneckAnalysis(report, hier, top\_func, \emptyset)$
6:     $children \leftarrow GetChildren(cfg, parameter\_order)$
7:     $LevelHeap \leftarrow \emptyset$
8:     $LevelHeap.append(\emptyset)$
9:     $LevelHeap[0].push(DesignPoint(cfg, \emptyset, report.result, 0, children))$
10:    while $LevelHeap \notin \emptyset$ and $elapsed\_time < DSE\_TIMEOUT$ do
11:        $curr\_level = GetLastLevel(LevelHeap)$
12:        $curr\_point \leftarrow LevelHeap[curr\_level].peek()$
13:        $tuned\_parameters = curr\_point.tuned$
14:        $candidate\_cfg, focused\_parameter \leftarrow curr\_point.children.pop()$
15:        parallel for $option \in focused\_parameter$ do
16:            $new\_cfg \leftarrow Manipulate(candidate\_cfg, focused\_parameter, option)$
17:            $new\_tuned \leftarrow tuned\_parameters + [(focused\_parameter, option)]$
18:            $report, hier \leftarrow Evaluate(new\_cfg)$
19:            $quality \leftarrow CalQuality(report.result, "FiniteDifference")$
20:            $best\_cfg \leftarrow UpdateBest(new\_cfg, quality)$
21:            $parameter\_order \leftarrow BottleneckAnalysis(report, hier, top\_func, new\_tuned)$
22:            if $len(parameter\_order) > 0$ then
23:                $children \leftarrow GetChildren(new\_cfg, parameter\_order)$
24:                $new\_point \leftarrow DesignPoint(new\_cfg, new\_tuned, report.result, quality, children)$
25:                $LevelHeap[curr\_level + 1].push(new\_point)$
26:            end if
27:        end for
28:        if $LevelHeap[curr\_level].peek().NumChildren == 0$ then
29:            $LevelHeap[curr\_level].pop()$
30:        end if
31:    end while
32: end for
33: return $best\_cfg$
```

As an example, when *AutoDSE* optimizes Code 1, it will see that the *convolution* part of the code takes 85.2% of the overall cycle counts. Since that section of the code is a series of nested loops, the parameters of the innermost loop will take the top of the list produced by the bottleneck analyzer. We shall explain in Section 5.4 (Rule 1) that we do not consider loops with trip count of less than or equal to 16 in our DSE since the HLS tool can automatically optimize these loops well. As a result, the `w` loop in Line 15 would be the innermost loop with parameters which the Merlin report tells us it is a computation-bound loop. According to the parameter ordering described in Section 5.3, *AutoDSE* first tries to apply `fg PIPELINE` which would be a successful attempt. In the next iteration, the last level heap will contain the design point that was just optimized and since the *convolution* part is still the bottleneck, *AutoDSE* would try parallelizing the `w` loop and will

Table 4. Performance and Area Compared to The Base Design When Parameters of Line 15 in Code 1 Change

Optimization	Status	Perf	BRAM	LUT	DSP	FF
Pi-`fg`	PASS (24 min)	175×	+7%	+23%	+24%	+15%
PF=4	TIMEOUT	-	-	-	-	-
Pi-`fg` + PF=4	PASS (28 min)	218×	+17%	+44%	+33%	+25%

Pi: Pipeline, PF: Parallel Factor, fg: fine-grained.
TIMEOUT is set to 60 minutes. The results suggest that applying fine-grained optimization first lets the HLS tool synthesize the design easier.

choose `factor=4` since it achieves the highest *quality* value. Although `factor=8` can reduce the cycle count by 11%, it increases the overall area (Equation (6)) by 63% which results in a worse quality; therefore, *AutoDSE* picks `factor=4` to make room for further improvement. By adopting Algorithm 1, *AutoDSE* can improve the performance by 218× very quickly, only after two iterations of the algorithm.

5.3 Parameter Ordering

It often happens that each bottleneck type has more than one applicable design parameter. In these situations, we sort the parameters by a pre-defined priority. For example, if the bottleneck of a loop statement is determined to be its computation, one can apply `fg` or `cg` pipelining/parallelization, in general. In this case, we treat the `PIPELINE` pragma as two different parameters based on its mode and choose the order of applying the pragmas to be `fg PIPELINE`, PARALLEL, and cg PIPELINE which is a heuristic approach to improve the performance by utilizing more fine-grained parallelization units since the HLS tool handles such optimizations better. Here, measuring the quality of design points with the finite difference value (Equation (5)) helps *AutoDSE* not to over-utilize the FPGA. For a configuration, when the gain of the achieved speedup is not comparable to the loss of available resources, the quality of design decreases; hence, *AutoDSE* will not tune that parameter and the resources are left for applying a design parameter with higher impact.

Moreover, as mentioned in Challenge 3 of Section 1, the order of applying the pragmas is crucial in order to get to the best design. Our experiments show that evaluating the fine-grained optimizations first helps *AutoDSE* reach the best design point in fewer iterations. This is mainly because HLS tools schedule fine-grained optimizations better than the coarse-grained ones. Table 4 shows how the performance and resource utilization change when `fg PIPELINE` and PARALLEL pragmas are applied to line 15 in Code 1 compared to the base design where all the pragmas are off. The time limit to run the HLS tool is set to 60 minutes. The results suggest that in order to get to the optimal configuration for this loop, we must first apply the fine-grained pipelining. This way, the HLS tool can better schedule the loop when parallelization is further applied and its synthesis will finish in 28 minutes. However, if we first apply the other pragma which results in a coarse-grained parallelization, the synthesis will be timed out and *AutoDSE* does not tune this pragma at this stage.

Note that we do not prune the other design parameters. We just change the order of the parameters to be explored as these rules cannot be generalized to all cases due to the unpredictability of the HLS tools. If the bottleneck of a design point is memory transfer, *AutoDSE* prioritizes `cg PIPELINE` over `TILING` pragma. The Merlin Compiler, by default, caches the data and the former will further overlap the communication time with computation by applying double buffering; however, the latter can be used to change the size of the cached data.

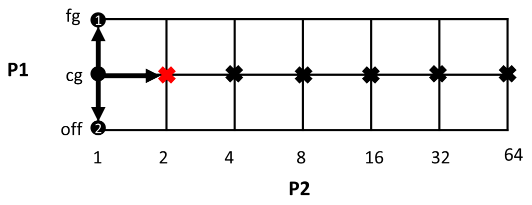


Fig. 4. Proposed design space representation and its impact on DSE. P1 and P2 denote the `PIPELINE` and `PARALLEL` pragmas, respectively.

5.4 Efficient Design Space Representation

To further facilitate the bottleneck-based optimizer, we seek to reduce the ineffective parameters. Intuitively, we can build a grid design space from the Merlin pragmas by treating each pragma as a tuning parameter and search for the best combination. However, many points in this grid space may be infeasible. For example, if we have determined to perform coarse-grained pipelining at the outermost loop of a loop nest, the Merlin Compiler will apply double-buffering on the loop. In this case, the physical meaning of double-buffering at the outermost loop is to transfer a batch of data from DRAM to BRAM, which cannot be further parallelized. As a result, pipeline and parallel pragmas are mutually exclusive in a loop nest. We propose an efficient approach to create a design space that preserves the grid design space but invalidates infeasible combinations.

Figure 4 illustrates the goal of an efficient design space representation. In this example, we attempt to explore the best parameter with the best option for loop `j` of Code 1 with pragma P1 and P2 denoting the `PIPELINE` and `PARALLEL` pragmas, respectively. Pragma $P1$ and $P2$ are exclusive when $P1$ is used in cg mode; therefore, only one of them should be inserted at a time. A good design space representation must preserve the grid design space but invalidate infeasible points. An example of such representation is presented in Figure 4. Assume that we are at the configuration $(P1, P2) = (\text{cg}, 1)$, we only have two candidates to explore in the next step because the configuration $(P1, P2) = (\text{cg}, 2)$ is invalid. This representation is exploration friendly and it is easy to enforce rules on the infeasible points.

To represent a grid design space with invalid points, we introduce a *Python* list comprehension syntax to *AutoDSE*. The *Python* list comprehension is a concise approach for creating lists with conditions. It has the following syntax:

```
list_name = [expression for item in list if condition]
```

Formally, we define the design space representation for Merlin pragmas with list comprehensions as follows:

```
    #pragma ACCEL <pragma-type> <attribute-key>=auto{
      options: parameter_name=list-comprehension-expression;
      default: default-value }
```

For our example, the design space can be represented using list comprehensions as follows:

```
// Skip the rest due to page limit
#pragma ACCEL PIPELINE auto{
  options: P1 = [x for x in [off, cg, fg]];
  default: 'off' }
#pragma ACCEL PARALLEL factor=auto{
  options: P2 = [x for x in [1, 2, 4, 8, 16, 32, 64] if P1!=cg];
  default: 1 }
for (int j = 0; j < NumIn; ++i) {
// Skip the rest due to page limit
```

where line 6 indicates that the two pragmas are exclusive. In other words, when we set $P1$ = `cg`, the available option for $P2$ is only the default value, which is 1 in this case. Note that the default value of each pragma turns it off.

There are three main advantages to adopting list comprehension-based design space representations. First, we are able to represent a design space with exclusive rules to greatly reduce its size. Second, the *Python* list comprehension is general. It provides a friendly and comprehensive interface for higher layers such as polyhedral analysis [66] and domain-specific languages to generate an effective design space in the future. Third, the syntax of this representation is *Python* compatible. This means we can leverage the *Python* interpreter to evaluate the design space and improve overall stability of the DSE framework.

The Design Space Generator, depicted in Figure 1, adapts the Rose Compiler [1] to analyze the kernel AST and extract the required information for running the DSE such as the loops in the design, their trip-count, and available bit-width. Artisan [47] employs a similar approach for analyzing the code. However, it only considers unroll pragma in code instrumentation. Our approach, on the other hand, considers a wider set of pragmas as mentioned in Table 2 and exploits the following rules to prune the design space:

(1) Ignore the fine-grained loops with **trip count (TC)** of less than or equal to 16 as the HLS tool can schedule these loops well.
(2) **Tiling factors (TF)** should be integer divisors of their loop TC.
(3) The allowed **parallel factors (PF)** for a loop are all sub-divisors of the loop TC up to $min(128, TC)$ plus the TC itself. PF of larger than 128 causes the HLS tool to run for a long time and it usually does not result in a good performance.
(4) For each loop, we should have $TF * PF \leq TC$.
(5) When `fg PIPELINE` is applied on a loop, no other pragma is allowed for the inner loops since this parameter wants to unroll all the inner loops completely.
(6) A parallel pragma is invalid for a loop nest when `cg PIPELINE` is applied on that loop.
(7) A tiling pragma is added only to the loops with an inner loop.

5.5 Design Space Partitioning

Unfortunately, the third inefficiency of the approaches reviewed in Section 5.1 also exists in our bottleneck-guided optimizer. We still cannot identify whether the current option of a parameter is locally or globally optimum. The most promising solution is breaking the dependency between options and searching a set of them in parallel. Although we need to evaluate multiple design points at every iteration, each design point will provide the maximum information for improving the performance because we always evaluate the parameters that have the largest impact on the performance bottleneck.

By partitioning the design space based on the likely distribution of locally optimal points and exploring each partition independently, we solve the local optimum issue caused by the non-smooth performance trend (Challenge 2 in Section 1) since each partition starts exploring from a different point. Intuitively, we could partition the design space according to the range of values of every parameter in a design, but it may generate thousands of partitions and result in a long exploration time. Instead, we partition the design space based on the pipeline mode, as `fg PIPELINE` unrolls all sub-loops while the `cg PIPELINE` exploits double buffers to implement coarse-grained pipelining. These two modes apparently have the most significant different influence on the generated architecture and are expected to have non-related performance and resource utilization. According to the pipeline modes in each loop, we use the tree partition and generate 2^m partitions from a design space with m non-innermost loops.

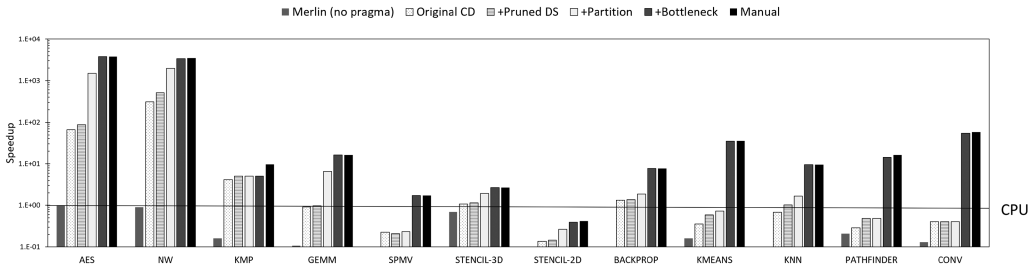


Fig. 5. Speedup of the Merlin Compiler without any pragmas, proposed approach with different optimizations, and the manual design over an Intel Xeon CPU core.

Supposing we use t working threads to perform, at most, h hours DSE for 2^m design space partitions, we need $\frac{2^m}{t} \times h$ hours to finish the entire process. On the other hand, some partitions that are based on an insignificant pipeline pragma may have a similar performance, so it is more efficient to only explore one of them. As a result, we profile each partition by running HLS with minimized parameter values to obtain the minimum area and performance and use K-means clustering with performance and area as features to identify t representative partitions among all 2^m partitions.

6 EVALUATION

6.1 Experimental Setup

Our evaluation is performed on **Amazon Elastic Compute Cloud (EC2)** [2]. We use `r4.4xlarge` instance with 16 cores and 122 GiB memory to perform the DSE and generate accelerator bitstreams. The generated FPGA accelerators are evaluated on an F1 instance (`f1.2xlarge`) with Xilinx Virtex UltraScale+$^{\text{TM}}$ VU9P FPGA. In addition, we choose the commonly-used MachSuite [39] benchmark suite and the FPGA-friendly Rodinia [8] benchmark, along with one convolution layer of Alexnet [29] as our first benchmark. For several common kernels, MachSuite provides C implementation that is programmed without the consideration of FPGA acceleration, which makes it a natural fit for demonstrating our approach. We evaluate the effect of our optimizations and compare the designs generated by our tool to the state-of-the-art works using this benchmark. Furthermore, to the best of our knowledge, we are the first ones to evaluate the performance of our tool on vision kernels of Xilinx Vitis libraries [52] that are optimized for Xilinx FPGAs, based on the OpenCV library [5].

6.2 Evaluation of Optimization Techniques

We first measure the performance of the Merlin Compiler without any pragmas and without the help of AutoDSE to get the impact of its default optimizations. The 1^{st} bar of each case in Figure 5 depicts the speedup gained by the Merlin Compiler with respect to CPU. Then, we evaluate the original **coordinate descent (CD)** method described in Section 5.1 and the proposed optimization strategies explained in Sections 5.4 and 5.5. The 2^{nd} to 4^{th} bars in Figure 5 show the speedup gained after tuning the pragmas by each of these optimizations. Note that the chart is in logarithmic scale. We can see that the default optimizations of the Merlin Compiler are not enough and after applying the candidate pragmas generated by the Original CD, we get 13.52× speedup, on the geometric mean. Moreover, each of the proposed strategies benefits at least one case in our benchmark and together further bring a 2.47× speedup. The list-based design space representation keeps the search space smooth by invalidating infeasible combinations. As a result, we can investigate more design points in a fixed amount of time. This helps `AES`, `NW`, `KMP`, `PATHFINDER`, `KMEANS`, and `KNN`. Design

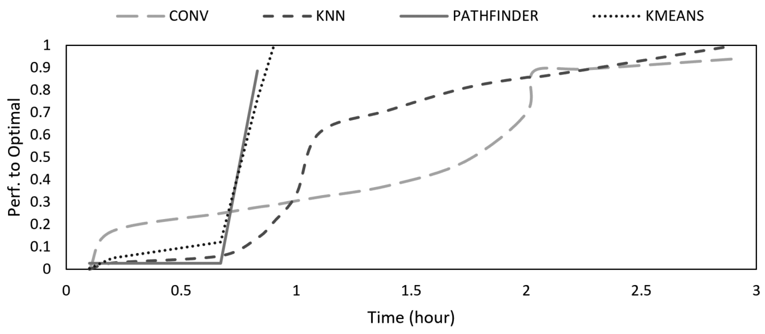


Fig. 6. Speedup over the manual design using AutoDSE for the four cases that the bottleneck-guided optimizer had significant impact.

space partition benefits the designs with many loop nests in which the coordinate process is easily trapped by the local optimum when changing pipeline modes—such as `AES`, `GEMM`, `NW`, `STENCIL-2D`, and `STENCIL-3D`.

The 5^{th} bar shows the speedup of *AutoDSE* when the bottleneck-guided coordinate optimizer detailed in Section 5.2 is adapted along with the parameter ordering explained in Section 5.3, design space representation introduced in Section 5.4, and design space partitioning described in Section 5.5. With this setup, *AutoDSE* further improves the result by 5.5× on the geometric mean bringing the overall speedup compared to when no pragmas are applied to 182.92×. As a result, *AutoDSE* is able to achieve a speedup of 19.9× over CPU and get to 0.93× performance of the manual designs while running only for 1.1 hours on the geometric mean. The manual designs, depicted by the 6^{th} bar, are optimized by applying the Merlin pragmas *manually* without changing the source programs.

Figure 6 depicts the *AutoDSE* process for four cases where the bottleneck-guided optimizer showed significant improvement in the performance. This shows that our approach can rapidly achieve a high performance design. *AutoDSE* does not exactly match the performance of manual designs for all of the cases because the HLS report may not reflect the accurate computation cycles when the kernels contain many unbounded loops or while-loops, which in turn affects the Merlin report. In order to get the importance of the parameters, the bottleneck analyzer (explained in Section 5.2) needs to receive the accurate cycle estimation of the design. In the absence of the true cycle breakdown, it cannot determine the high-impact design parameters. Therefore, our search algorithm may focus on unimportant parameters.

6.3 Comparison with Other DSE Approaches

We further evaluate the overall performance of generated accelerator designs by *AutoDSE* compared to the previous state-of-the-art works including S2FA [57], lattice-traversing DSE [23], and Gaussian process-based Bayesian optimization [46] in Table 5. The numbers show the speedup of the design found by *AutoDSE* compared to the design that their framework found after running the tools for the same allotted time. Note that the performance of the other works are not reported by the authors for all of the kernels we are testing. According to Table 5, by utilizing the bottleneck-based approach, we can outperform S2FA, lattice-traversing DSE, and Gaussian process-based Bayesian optimization by 3.45× (86.56×), 4.23× (5.11×), 17.92× (43.33×), respectively, on the geometric (arithmetic) mean.

As we discussed in Section 5.1, the deficiency of S2FA stems from how hard it is for the problem-independent learning algorithm to find the key parameters. Lattice-traversing DSE needs an initial

Table 5. Speedup of Our Approach Compared to S2FA [57], Lattice-traversing DSE [23], and Gaussian Process-based Bayesian Optimization [46]

Approach	AES	NW	GEMM	KMP	SPMV	STENCIL-3D	**GEO-Mean**	**ARITH-Mean**
Lattice (ICCD'18) [23]	1.63	6.32	7.39	-	-	-	**4.23**	**5.11**
S2FA (DAC'18) [57]	512.86	1	1.52	1.74	1	1.26	**3.45**	**86.56**
Bayesian (DATE'21) [46]	-	-	100.17	-	2.07	27.75	**17.92**	**43.33**

sampling step to learn the design space. This takes a long time for our benchmark due to the size of the design space even though the authors only consider unrolling the loops and function inlining. This constraint makes it hard for the tool to start the exploration process before the time limit for DSE is met. The Gaussian process-based Bayesian optimization also has to spend some time to sample the design space and build an initial surrogate model. However, *AutoDSE* can learn the high-impact directives by exploring the performance breakdown and thus, is able to find a high-performance design in a few iterations.

Moreover, adopting the Merlin Compiler as the backend gives further advantage to *AutoDSE* compared to other DSE tools. This allows the tool to exploit the automatic code transformations for applying the common optimization techniques such as memory burst, memory coalescing, and double buffering; and focus only on high-level hardware changes. Nonetheless, the performance comparison with S2FA shows that adoption of the Merlin Compiler alone is not enough and we still need to explore the design space more efficiently. Another source of inefficiency in the S2FA framework is in translation of code from Scala to a C code to be used by the Merlin Compiler. The generated C code, in general, may be not efficient enough for HLS especially for applications like AES that require bit-level optimizations.

6.4 Comparison with Expert-level Manual HLS Designs

To further evaluate the performance of *AutoDSE*, we use 33 vision kernels from Xilinx Vitis Library [52]. These kernels utilize 14 optimization pragmas, on average (by the geometric mean), which include `UNROLL`, `PIPELINE`, `ARRAY_PARTITION`, `DEPENDENCE`, `LOOP_FLATTEN`, `INLINE`, `DATAFLOW`, and `STREAM`. For each kernel, we remove all the optimization pragmas except for `DATAFLOW` and `STREAM`. The removed pragmas, which are of the first six types mentioned above, are used 13.47 times on average (out of 14). As a result, we require less than one optimization pragma per kernel, on the geometric mean. The only optimization pragmas kept are `DATAFLOW` and `STREAM` pragmas. This is because our search space is built on top of the Merlin Compiler and we do not search for the `DATAFLOW` and `STREAM` pragmas as these pragmas are not among the Merlin-specified pragmas. In the future, we will expand our search engine to HLS pragmas that are not included in Merlin. Furthermore, the `INTERFACE` and `LOOP_TRIPCOUNT` pragmas are also kept which are not among the HLS optimization pragmas. They are rather used to specify the connection to AXI bus and the range of the trip count of the loop, respectively.

To better understand the effect of our optimizer, we tested the performance of the Vitis tool and the Merlin Compiler on the input to *AutoDSE* (which does not include the optimization pragmas mentioned above). The performance comparisons are summarized in Table 6. As the results show, while the Merlin Compiler can get to a speedup of 3.29× compared to the Vitis tool, it still needs the help of *AutoDSE* to get to the manually optimized kernels in the library. In fact, *AutoDSE* could achieve a further speedup of 2.74× by automatically inserting 3.2 Merlin pragmas per kernel, on the geometric mean. As a result, it could improve the performance of the Vitis tool by 9.04× and 1.04× when the code with reduced set of pragmas and the manual code, respectively, are passed to the Vitis tool.

Table 6. Average (Geometric Mean) Speedup of the Vitis Tool, the Merlin Compiler, and *AutoDSE* Over the Manually Optimized Kernels from Xilinx Vitis Libraries

Comparison Scenario \ Compared to	Vitis (Manually Optimized)	Vitis (Default)	Merlin Compiler	*AutoDSE*
Speedup over the Vitis Library with (Original) Manually Inserted Pragmas	1×	0.12×	0.38×	1.04×
Performance Improvement over the Vitis Tool with Default Settings (no pragma)	8.69×	1×	3.29×	9.04×
#pragmas Listed in the Table's Caption	13.47	0	0	0
Total Pragma Reduction	1×	26.38×	26.38×	26.38×

The manual designs are the original kernels from the library. The performance of those designs are compared to when the optimization pragmas we search for (`UNROLL`, `PIPELINE`, `ARRAY_PARTITION`, `DEPENDENCE`, `LOOP_FLATTEN`, and `INLINE`) are removed and the code is passed to three different tools.

Figure 9 in Appendix A.4 depicts the performance comparison of the design points *AutoDSE* generated with respect to Xilinx results along with the number of pragmas that we removed in detail. The results show that *AutoDSE* is able to achieve the same or better performance yet with 26.38× reduction of their optimization pragmas in 0.3 hours, on the geometric mean; therefore, proving the effectiveness of our bottleneck-based approach and the fact that it can mimic the method an expert would take. For the cases that *AutoDSE* does not exactly match the performance of Vitis, *AutoDSE* still finds the best combination of the pragmas. The inequality lies in the different II that Merlin has achieved. For example, the `histEqualize`, `histogram`, and `otsuthreshold` kernels all have a loop that requires the II to be set to 2 when `PIPELINE` pragma is used. Otherwise, Vivado HLS achieves an II=3. However, it is not possible to change the II using the Merlin Compiler. On the other hand, *AutoDSE* is able to outperform the performance of `customConv` and `reduce` kernels significantly by better detecting the choices and locations for pipelining and parallelization.

7 CONCLUSION AND FUTURE WORK

In this paper, we made our first, yet very important, step of lowering the bar of accelerating programs using FPGA for general software programmers to make FPGA universally accessible. We analyzed the difficulty of exploring HLS design space. To address challenges 2 to 4 mentioned in Section 1, we treat the HLS tool as a black-box. We use the synthesis results to estimate the QoR rather than the **placement and routing (***P&R***)** results, because *P&R* is too time-consuming to explore sufficient design points in a reasonable time budget. According to our observation and analysis, we propose a bottleneck-guided coordinate optimizer and develop a push-button framework, *AutoDSE*, based on that to systematically approach a better solution. By exploring the solution space efficiently, we address challenges 1 and 5. We propose a heuristic for ordering the parameters that can further help challenges 3 and 5. To eliminate meaningless design points, we incorporate a list comprehension-based design space representation and prune 24.65× ineffective configurations on average, while keeping the design space smooth; hence, further alleviating Challenge 1. Additionally, we employ a partitioning strategy to address the local optimum problem mentioned in Challenge 2. We show that *AutoDSE* can outperform general hyper-heuristics used in the literature by focusing on high-impact design parameters first. The experimental results suggest that *AutoDSE* lets anyone with a decent knowledge of programming try customized computing with minimum effort.

AutoDSE is built with the assumption that we can get the performance breakdown of the program from the HLS tool. We expect all HLS tools will provide performance breakdown at some point, as it is important for manual performance optimization (such as the need for Intel VTune Profiler [26] in the case of CPU performance optimization). Xilinx HLS is already providing such information that the Merlin Compiler leverages and Intel OpenCL [27] is planning to add this feature. It is likely that other HLS tools [6, 7, 16] will add such information as well in the near future. Hence, we believe, it is reasonable for *AutoDSE* to take advantage of such information to mimic the human performance optimization process to perform bottleneck-driven DSE.

In the future, we plan to include more transformations (design space parameters) for optimizing data access and reuse patterns. We will also extend *AutoDSE* to estimate the QoR based on the *P&R* results by developing a machine learning model to predict them from the synthesis results as in [18].

A APPENDIX

A.1 Optimized HLS Code for CNN

Code 3 shows the optimized HLS code for the CNN algorithm in Code 1 after applying the code transformations and pragmas listed in Table 1.

Code 3. Optimized CNN HLS C Code Snippet

```
// Skip const variable initization for brevity

void CnnKernel(const ap\_uint< 128 > * input, float weight,
                const ap\_uint< 512 > * bias, ap\_uint< 512 > * output){
#pragma HLS INTERFACE m_axi port=input bundle=gmem1 depth=3326977
#pragma HLS INTERFACE s_axilite port=input bundle=control
// Skip the rest for brevity

  float bias_buf[ParallelOut][ParallelOut];
#pragma HLS array_partition variable=bias_buf complete dim=2

  float C[ParallelOut][ImSize][ImSize];
#pragma HLS array_partition variable=C cyclic factor=8 dim=3
#pragma HLS array_partition variable=C cyclic factor=2 dim=2
#pragma HLS array_partition variable=C complete dim=1

  LoadBurst(bias, bias_buf);

  for (int i = 0; i < NumOut / ParallelOut; i++) {
    float weight_buf[NumOut / ParallelOut][NumIn][kKernel][kKernel];
#pragma HLS array_partition variable=weight_buf complete dim=4
#pragma HLS array_partition variable=weight_buf complete dim=3
#pragma HLS array_partition variable=weight_buf complete dim=1

    float output_buf[NumOut / ParallelOut][OutImSize][OutImSize];
#pragma HLS array_partition variable=output_buf cyclic factor=16 dim=3
#pragma HLS array_partition variable=output_buf complete dim=1

    LoadBurst(weight, weight_buf);
    // Initialization
    for (int h = 0; h < ImSize; ++h) {
      for (int w = 0; w < ImSize / 4; ++w) {
#pragma HLS dependence variable=C array inter false
#pragma HLS pipeline
        for (int w_sub = 0; w_sub < 4; ++w_sub) {
#pragma HLS unroll
          for (int po = 0; po < ParallelOut; po++) {
#pragma HLS unroll
            C[po][h][w * 4 + w_sub] = 0.f;
    } } } }
    // Convolution
    for (int j = 0; j < NumIn; ++j) {
      float input_buf[InImSize][InImSize];
#pragma HLS array_partition variable=input_buf cyclic factor=8 dim=2
#pragma HLS array_partition variable=input_buf cyclic factor=5 dim=1
      LoadBurst(input, input_buf);
      for (int h = 0; h < ImSize; ++h) {
        for (int w = 0; w < ImSize / 4; ++w) {
#pragma HLS dependence variable=C array inter false
#pragma HLS pipeline
     for (int w_sub = 0; w_sub < 4; ++w_sub) {
#pragma HLS unroll
      for (int po = 0; po < ParallelOut; po++) {
#pragma HLS unroll
       float tmp = 0.f;
       for (int p = 0; p < kKernel; ++p) {
#pragma HLS unroll
        for (int q = 0; q < kKernel; ++q) {
#pragma HLS unroll
         tmp += ...;
       } }
       C[po][h][w * 4 + w_sub] += tmp;
  } } } } }
  // ReLU + Max pooling
  for (int h = 0; h < OutImSize; ++h) {
   for (int w = 0; w < OutImSize; ++w) {
#pragma HLS dependence variable=output_buf array inter false
#pragma HLS pipeline
    for (int po = 0; po < ParallelOut; po++) {
#pragma HLS unroll
      output_buf(h, w, po) = ...
  } } }
  StoreBurst(output, output_buf);
} }
```

A.2 Challenges of Model-based Approaches

The main problem with the model-based techniques is the fact that they assume the performance and/or area have a smooth trend with the change of pragmas. However, as Figure 7 illustrates this is not a valid assumption with the latest HLS tools. This figure depicts the execution cycle of the NW algorithm [34] from MachSuite benchmark [39] with different parallel factors for its five loops. Although the performance trend of three loops are ideal, the rest of the two loops (CG-loop-2 and FG-loop-1[3]) are not. This is because the optimizations and heuristics employed by the tools are constantly changing so the models should be updated regularly.

A.3 Frequency of Employed Parameters

We explored the number of candidate pragmas and the number of pragmas that *AutoDSE* tuned for optimizing the design to find out the most influential parameters. Figure 8 demonstrates the results for the best design found by *AutoDSE* for each of the target kernels. For illustration purposes, we depict the frequency of employed pragmas for those who either had the highest speedup or utilized the most number of pragmas, meaning that *AutoDSE* spent more iterations for optimizing them. As the results suggest, the tiling pragma has special use case, only CONV and STENCIL-2D utilized it. For the rest of the kernels, it is either not applicable or not necessary. The parallel and pipeline pragmas are equally important. The pipeline pragma is employed more where there is a nested loop, whereas for single loops, the parallel pragma is more preferred. This is because the HLS tools can implement both fg and cg pipelining better than cg parallelization. Also, when a single loop does not have any pragma, it will be pipelined automatically. Looking at the vision kernels from the Xilinx Vitis Library that we targeted, there are 228, 122, and 18 candidate parallel, pipeline, and tiling pragmas in total, respectively. Since these kernels rarely utilize deep nested loops, *AutoDSE* chose 65 and 48 of the parallel and pipeline pragmas in total, respectively; while, it never used any tiling pragmas.

[3]CG and FG mean coarse-grained and fine-grained, respectively.

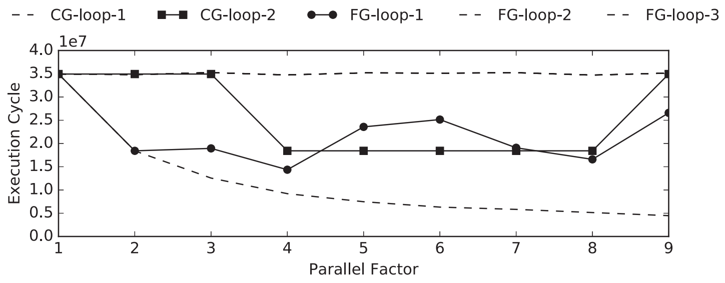


Fig. 7. HLS cycles of NW algorithm [34] with different parallel factors on loops.

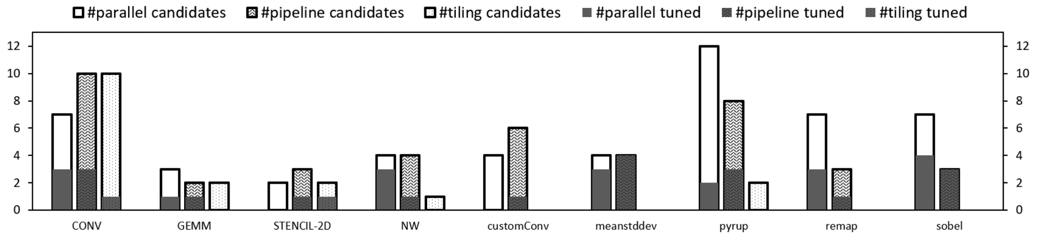


Fig. 8. The number of candidate pragmas vs. the number of chosen pragmas for the optimal designs.

A.4 Detailed Comparison to the Vitis Library

Figure 9 depicts the detailed comparison of *AutoDSE* to the expert-level manual HLS designs from Xilinx Vitis libraries [52]. As explained in Section 6.4, when testing with *AutoDSE*, all the optimization pragmas that the Merlin Compiler can derive with the help of its own pragmas are removed. *AutoDSE* can achieve the same performance while using 26.38× less pragmas, on the geometric mean.

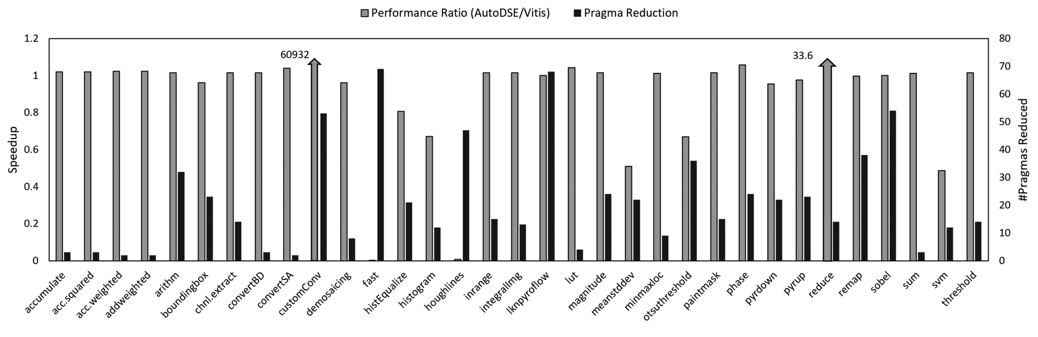


Fig. 9. Speedup and number of reduced pragmas using AutoDSE compared to the vision kernels of Xilinx Vitis Libraries [52].

REFERENCES

[1] [n.d.]. Rose compiler infrastructure. http://rosecompiler.org/.
[2] Amazon EC2 F1 Instance. [n.d.]. https://aws.amazon.com/ec2/instance-types/f1/.
[3] Joao Andrade, Nithin George, Kimon Karras, David Novo, Frederico Pratas, Leonel Sousa, Paolo Ienne, Gabriel Falcao, and Vitor Silva. 2017. Design space exploration of LDPC decoders using high-level synthesis. *IEEE Access* 5 (2017), 14600–14615.

[4] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An extensible framework for program autotuning. In *PACT*. 303–316.

[5] Gary Bradski. 2000. The OpenCV library. *Dr Dobb's J. Software Tools* 25 (2000), 120–125.

[6] Cadence Stratus High-Level Synthesis. [n.d.]. https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html.

[7] Catapult High-Level Synthesis. [n.d.]. https://eda.sw.siemens.com/en-US/ic/ic-design/high-level-synthesis-and-verification-platform/.

[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*. 44–54.

[9] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. 2018. SODA: Stencil with optimized dataflow architecture. In *ICCAD*. 1–8.

[10] Young-kyu Choi and Jason Cong. 2018. HLS-based optimization and design space exploration for applications with variable loop bounds. In *ICCAD*. 1–8.

[11] Jason Cong, Muhuan Huang, Peichen Pan, Yuxin Wang, and Peng Zhang. 2016. Source-to-source optimization for HLS. In *FPGAs for Software Programmers*. 137–163.

[12] Jason Cong, Muhuan Huang, Peichen Pan, Di Wu, and Peng Zhang. 2016. Software infrastructure for enabling FPGA-based accelerations in data centers. In *ISLPED*. 154–155.

[13] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. 2011. High-level synthesis for FPGAs: From prototyping to deployment. In *TCAD*, Vol. 30. 473–491.

[14] Jason Cong and Jie Wang. 2018. PolySA: Polyhedral-based systolic array auto-compilation. In *ICCAD*. 1–8.

[15] Jason Cong, Peng Wei, Cody Hao Yu, and Peng Zhang. 2018. Automated accelerator generation and optimization with composable, parallel and pipeline architecture. In *DAC*.

[16] CyberWorkBench. [n.d.]. https://www.nec.com/en/global/prod/cwb/index.html.

[17] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 5, 1 (1998), 46–55.

[18] Steve Dai, Yuan Zhou, Hang Zhang, Ecenur Ustun, Evangeline FY Young, and Zhiru Zhang. 2018. Fast and accurate estimation of quality of results in high-level synthesis with machine learning. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 129–132.

[19] Robert H. Dennard, Fritz H. Gaensslen, Hwa-Nien Yu, V. Leo Rideout, Ernest Bassous, and Andre R. LeBlanc. 1974. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits* 9, 5, 256–268.

[20] Javier Duarte, Song Han, Philip Harris, Sergo Jindariani, Edward Kreinar, Benjamin Kreis, Jennifer Ngadiuba, Maurizio Pierini, Ryan Rivera, Nhan Tran, and Zhenbin Wu. 2018. Fast inference of deep neural networks in FPGAs for particle physics. *Journal of Instrumentation* 13, 07 (2018), P07027.

[21] Falcon Computing Solutions, Inc. [n.d.]. http://www.falcon-computing.com.

[22] Lorenzo Ferretti, Giovanni Ansaloni, and Laura Pozzi. 2018. Cluster-based heuristic for high level synthesis design space exploration. *IEEE Transactions on Emerging Topics in Computing*.

[23] Lorenzo Ferretti, Giovanni Ansaloni, and Laura Pozzi. 2018. Lattice-traversing design space exploration for high level synthesis. In *ICCD*. 210–217.

[24] Álvaro Fialho, Luis Da Costa, Marc Schoenauer, and Michèle Sebag. 2010. Analyzing bandit-based adaptive operator selection mechanisms. *Annals of Mathematics and Artificial Intelligence* 60, 1–2, 25–64.

[25] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, Prashant Singh Rawat, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and Ponnuswamy Sadayappan. 2018. GPU code optimization using abstract kernel emulation and sensitivity analysis. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 736–751.

[26] Intel. [n.d.]. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html.

[27] Intel SDK for OpenCL Applications. [n.d.]. https://software.intel.com/en-us/intel-opencl.

[28] David Koeplinger, Raghu Prabhakar, Yaqi Zhang, Christina Delimitrou, Christos Kozyrakis, and Kunle Olukotun. 2016. Automatic generation of efficient accelerators for reconfigurable hardware. In *ISCA*. 115–127.

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *NIPS*. 1097–1105.

[30] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. 2019. HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 242–251.

[31] Hung-Yi Liu and Luca P. Carloni. 2013. On learning-based methods for design-space exploration with high-level synthesis. In *DAC*. 1–7.

[32] Shuangnan Liu, Francis CM Lau, and Benjamin Carrion Schafer. 2019. Accelerating FPGA prototyping through predictive model-based HLS design space exploration. In *DAC*. 1–6.

[33] Anushree Mahapatra and Benjamin Carrion Schafer. 2014. Machine-learning based simulated annealer method for high level synthesis design space exploration. In *ESLsyn*. 1–6.

[34] Saul B. Needleman and Christian D. Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Mol. Biol* 48, 443–153.

[35] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. 2020. Predictable accelerator design with time-sensitive affine types. *arXiv preprint arXiv:2004.04852* (2020).

[36] David Parello, Olivier Temam, Albert Cohen, and J-M Verdun. 2004. Towards a systematic, pragmatic and architecture-aware program optimization process for complex processors. In *SC'04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*. IEEE, 15–15.

[37] Raghu Prabhakar, David Koeplinger, Kevin J. Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. 2016. Generating configurable hardware from parallel patterns. *ASPLOS* 51, 4, 651–665.

[38] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In *ISCA*. 13–24.

[39] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. MachSuite: Benchmarks for accelerator design and customized architectures. In *IISWC*. 110–119.

[40] Enrico Reggiani, Marco Rabozzi, Anna Maria Nestorov, Alberto Scolari, Luca Stornaiuolo, and Marco Santambrogio. 2019. Pareto optimal design space exploration for accelerated CNN on FPGA. In *IPDPSW*. 107–114.

[41] Benjamin Carrion Schafer. 2017. Parallel high-level synthesis design space exploration for behavioral IPs of exact latencies. *TODAES* 22, 4, 1–20.

[42] Benjamin Carrion Schafer and Kazutoshi Wakabayashi. 2012. Divide and conquer high-level synthesis design space exploration. *TODAES* 17, 3, 1–19.

[43] B. Carrion Schafer and Kazutoshi Wakabayashi. 2012. Machine learning predictive modelling high-level synthesis design space exploration. In *IET Computers & Digital Techniques*, Vol. 6. 153–159.

[44] Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Mostofa Patwary, Mr Prabhat, and Ryan Adams. 2015. Scalable Bayesian optimization using deep neural networks. In *International Conference on Machine Learning*. PMLR, 2171–2180.

[45] Atefeh Sohrabizadeh, Jie Wang, and Jason Cong. 2020. End-to-end optimization of deep learning applications. In *FPGA*. 133–139.

[46] Qi Sun, Tinghuan Chen, Siting Liu, Jin Miao, Jianli Chen, Hao Yu, and Bei Yu. 2021. Correlated multi-objective multi-fidelity optimization for HLS directives design. In *IEEE/ACM Proceedings Design, Automation and Test in Europe (DATE)*. 01–05.

[47] Jessica Vandebon, Jose GF Coutinho, Wayne Luk, Eriko Nurvitadhi, and Tim Todman. 2020. Artisan: A meta-programming approach for codifying optimisation strategies. In *FCCM*. 177–185.

[48] Vivado HLS. 2021. www.xilinx.com/products/design-tools/vivado.

[49] Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A polyhedral compiler for high-performance systolic arrays on FPGA. In *Proceedings of the 2021 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*.

[50] Shuo Wang, Yun Liang, and Wei Zhang. 2017. FlexCL: An analytical performance model for OpenCL workloads on flexible FPGAs. In *DAC*. 1–6.

[51] Xilinx. 2021. https://www.xilinx.com/about/xilinx-ventures/falcon-computing.html.

[52] Xilinx Vitis Libraries. 2021. www.github.com/Xilinx/Vitis_Libraries.

[53] Xilinx Vitis Platform. 2021. https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html.

[54] Chang Xu, Gai Liu, Ritchie Zhao, Stephen Yang, Guojie Luo, and Zhiru Zhang. 2017. A parallel bandit-based approach for autotuning FPGA compilation. In *FPGA*. 157–166.

[55] Pengfei Xu, Xiaofan Zhang, Cong Hao, Yang Zhao, Yongan Zhang, Yue Wang, Chaojian Li, Zetong Guan, Deming Chen, and Yingyan Lin. 2020. AutoDNNchip: An automated DNN chip predictor and builder for both FPGAs and ASICs. In *FPGA*. 40–50.

[56] Sotirios Xydis, Gianluca Palermo, Vittorio Zaccaria, and Cristina Silvano. 2014. SPIRIT: Spectral-Aware Pareto iterative refinement optimization for supervised high-level synthesis. In *TCAD*, Vol. 34. 155–159.

[57] Cody Hao Yu, Peng Wei, Max Grossman, Peng Zhang, Vivek Sarker, and Jason Cong. 2018. S2FA: An accelerator automation framework for heterogeneous computing in datacenters. In *DAC*. 1–6.

[58] Georgios Zacharopoulos, Lorenzo Ferretti, Giovanni Ansaloni, Giuseppe Di Guglielmo, Luca Carloni, and Laura Pozzi. 2019. Compiler-assisted selection of hardware acceleration candidates from application source code. In *ICCD*. 129–137.
[59] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. 2008. AutoPilot: A platform-based ESL synthesis system. In *High-Level Synthesis*. 99–112.
[60] 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 *ICCAD*. 430–437.
[61] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. FlexTensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *ASPLOS*. 859–873.
[62] Guanwen Zhong, Alok Prakash, Yun Liang, Tulika Mitra, and Smail Niar. 2016. Lin-analyzer: A high-level performance analysis tool for FPGA-based accelerators. In *DAC*. 1–6.
[63] Guanwen Zhong, Alok Prakash, Siqi Wang, Yun Liang, Tulika Mitra, and Smail Niar. 2017. Design space exploration of FPGA-based accelerators with multi-level parallelism. In *DATE*. 1141–1146.
[64] Guanwen Zhong, Vanchinathan Venkataramani, Yun Liang, Tulika Mitra, and Smail Niar. 2014. Design space exploration of multiple loops on FPGAs using high level synthesis. In *ICCD*. 456–463.
[65] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. 2018. Combined spatial and temporal blocking for high-performance stencil computation on FPGAs using OpenCL. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 153–162.
[66] Wei Zuo, Peng Li, Deming Chen, Louis-Noël Pouchet, Shunan Zhong, and Jason Cong. 2013. Improving polyhedral code generation for high-level synthesis. In *CODES+ ISSS*. 1–10.

Received July 2021; revised September 2021; accepted October 2021