A Unified Framework for Automated Code Transformation and Pragma Insertion

Stéphane Pouget
pouget@cs.ucla.edu
University of California, Los Angeles
Los Angeles, CA, USA

Louis-Noël Pouchet
pouchet@colostate.edu
Colorado State University
Fort Collins, CO, USA

Jason Cong
cong@cs.ucla.edu
University of California, Los Angeles
Los Angeles, CA, USA

Abstract

High-Level Synthesis compilers and Design Space Exploration tools have greatly advanced the automation of hardware design, improving development time and performance. However, achieving a good Quality of Results still requires extensive manual code transformations, pragma insertion, and tile size selection, which are typically handled separately. The design space is too large to be fully explored by this fragmented approach. It is too difficult to navigate this way, limits the exploration of potential optimizations, and complicates the design generation process.

To tackle this obstacle, we propose Sisyphus, a unified framework that automates code transformation, pragma insertion, and tile size selection within a common optimization framework. By leveraging Nonlinear Programming, our approach efficiently explores the vast design space of regular loop-based kernels, automatically selecting loop transformations and pragmas that minimize latency.

Evaluation against state-of-the-art frameworks, including AutoDSE, NLP-DSE, and ScaleHLS, shows that Sisyphus achieves superior Quality of Results, outperforming alternatives across multiple benchmarks. By integrating code transformation and pragma insertion into a unified model, Sisyphus significantly reduces design generation complexity and improves performance for FPGA-based systems.

CCS Concepts

• **Hardware → High-level and register-transfer level synthesis**; • **Software and its engineering → Compilers**.

Keywords

HLS, code transformation, pragma insertion, non-linear problem

ACM Reference Format:
Stéphane Pouget, Louis-Noël Pouchet, and Jason Cong. 2025. A Unified Framework for Automated Code Transformation and Pragma Insertion. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '25), February 27–March 1, 2025, Monterey, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3706628.3708873

1 Introduction

High-Level Synthesis (HLS) compilers and source-to-source compilers are indispensable tools in accelerating hardware design by automating the translation of high-level programming languages like C/C++ or Python into hardware descriptions. They offer notable advantages, such as reduced development time and enhanced performance for hardware designs [4, 5, 11–14, 21, 28, 34, 35, 37–40]. However, achieving a good quality of results (QoR) often requires manual code transformations and pragma insertions, which can be facilitated with either Design Space Exploration (DSE) [7, 26, 27, 29, 30, 30, 31, 41] or source-to-source compilers [13, 36, 37, 39] to guide the synthesis process. These pragmas aid in optimizing the generated hardware code.

Despite their advantages, existing HLS workflows face several significant challenges. Previous frameworks [13, 36, 37, 39] have integrated pragma insertion and code transformation, but these efforts are somewhat limited, focusing primarily on loop permutations based on loop properties. Additionally, code transformations and pragma insertion are often treated as separate processes. Transformations such as loop tiling, splitting, and permutation are typically performed in advance or as part of a pre-processing stage, followed by pragma insertion for optimization, like loop unrolling, pipelining and array partitioning. This separation introduces inefficiencies, as transformations can affect the effectiveness of pragmas, and vice versa. Moreover, the immense design space—encompassing millions of potential transformation and pragma combinations—makes exhaustive exploration unfeasible. Ensuring the correctness of these transformations while adhering to resource constraints further complicates the process.

Our primary research objective is to develop a system capable of autonomously conducting code transformation, tile size selection to cache the data on-chip and integrating hardware pragmas for HLS to enhance parallelization. We seek to attain a favorable QoR, especially for computation-bound designs, where maximizing parallelism is crucial for optimizing QoR. To address this challenge, we introduce Sisyphus, a framework built on top of HLS compilers. This framework automates code transformation, including loop splitting, permutation, tiling, and pragma insertion for unrolling, pipelining, and on-chip data caching while partitioning arrays to ensure efficient parallelization, all within a single optimization framework. Even though several exploration methods could be used, we chose to employ a Nonlinear Programming (NLP) approach. This method facilitates rapid exploration of the entire theoretical space. We have developed an analytical model that integrates considerations of latency and resource utilization, building upon previous research [26, 27]. Our focus lies specifically on regular loop-based kernels [23], ensuring meticulous control over the correctness of code transformations and the accuracy of cost models. This model relies on parameters derived from both the program's schedule

FPGA '25, February 27–March 1, 2025, Monterey, CA, USA

ACM ISBN 979-8-4007-1396-5/25/02
https://doi.org/10.1145/3706628.3708873

and the pragma configuration. To facilitate seamless code transformation while respecting constraints and pragma insertion, we have designed a novel template tailored to these objectives. In our template, we incorporate a two-level tiling strategy, where each tiling level corresponds to particular HLS optimizations such as fine-grained unrolling, pipelining, coarse-grained unrolling, or tiling. As a result, the loop trip counts become variables, with each loop associated with a specific pragma determined by the tiling level. By solving the NLP problem, we determine these trip counts, as well as other parameters like the array partitioning factor, enabling the automatic generation of the corresponding C++ code.
In summary, we introduce the following contributions:

- A novel optimization template that integrates code transformation, pragma insertion, and tiles size selection for data caching by consolidating these tasks into a single optimization problem. This approach not only simplifies the search space but also ensures that only legal transformations are considered.
- A novel NLP-based approach specifically designed to explore the joint design space of regular loop-based kernels. Unlike traditional methods that involve time-consuming DSE, our approach provides solutions within minutes. Plus, the NLP is user-friendly, allowing users to easily configure parameters, such as on-chip buffer size, to meet the specific requirements of the kernel.
- Our framework acts as a comprehensive, fully automated system, offering end-to-end functionality. With it, we can conduct thorough evaluations and attain QoR designs that are comparable or superior to those achieved by AutoDSE, NLP-DSE, ScaleHLS and HARP – *all without requiring DSE to perform multiple HLS compiler syntheses*. Furthermore, Sisyphus identifies the correct transformation, even when no transformation is needed.

The paper is organized into the following sections. Section 2 explains our approach and solution. Following that, Section 3 covers code transformation and pragma insertion. Section 4 introduces a nonlinear formulation based on this model to automatically discover schedule and pragma configurations through NLP optimization. Section 5 elaborates on our post-optimizations. Finally, sections 6, 7, 8 and 9 are dedicated to evaluating our method, presenting related work, addressing limitations and drawing conclusions.

2 Background and Motivation

2.1 Design Space to Explore

The exploration space for optimizing an HLS design includes pragma insertion, code transformations, and data caching, all within the limits of resource constraints. Although different objective functions can be explored, this work specifically focuses on minimizing latency while respecting resource constraints. This objective function is especially potent for computation bound kernels because it addresses the bottleneck caused by insufficient parallelization.

The various components of the exploration space—pragma insertion, code transformations, and data caching—are inherently interdependent. For example, unrolling pragmas affect array partitioning, which in turn influences BRAM utilization. Similarly, loop scheduling and pragmas impact the potential for on-chip data caching, while the choice of schedule dictates how loops should be unrolled based on their characteristics, such as whether they are reduction loops. A reduction loop is a loop where an operation, like addition or multiplication, accumulates values across iterations to produce a single output, often by reducing a multi-element array to a scalar (e.g., loop L2 in Listing 1). The choice of schedule for reduction loops, as opposed to non-reduction loops (e.g., L0, L1, and L3 in Listing 1), influences unrolling strategies and overall optimization because of the dependencies inherent to the loop's characteristics.

Selecting one optimization affects others, often requiring backtracking when spaces are handled separately. This interdependence complicates exploration and demands constant verification of transformation legality.

```
for (i = 0; i < 200; i++) { // L0
  for (j = 0; j < 220; j++) // L1
    C[i][j] *= beta; // S0
  for (k = 0; k < 240; k++) // L2
    for (j = 0; j < 220; j++)// L3
      C[i][j]+=alpha*A[i][k]*B[k][j]; } // S1
```

Listing 1: Original gemm Code

```
for (i = 0; i < 200; i++) // L0
  for (j = 0; j < 220; j++) // L1
    C[i][j] *= beta; // S0
for (i = 0; i < 200; i++) // L2
  for (k = 0; k < 240; k++) // L3
    for (j = 0; j < 220; j++) // L4
      C[i][j]+=alpha*A[i][k]*B[k][j]; // S1
```

Listing 2: Fully Distributed Original gemm Code

```
for (i0 = 0; i0 < I0_S0; i0++) // L0
 for (i1 = 0; i1 < I1_S0; i1++) // L1
  for (i2 = 0; i2 < I2_S0; i2++) // L2
    for (j0 = 0; j0 < J0_S0; j0++) // L3
     for (j1 = 0; j1 < J1_S0; j1++) // L4
      for (j2 = 0; j2 < J2_S0; j2++) // L5
        C[i][j] *= beta; // S0
for (i = 0; i < I0_S1; i++) // L6
 for (i = 0; i < I1_S1; i++) // L7
  for (i = 0; i < I2_S1; i++) // L8
    for (k = 0; k < K0_S1; k++) // L9
     for (k = 0; k < K1_S1; k++) // L10
      for (k = 0; k < K2_S1; k++) // L11
        for (j = 0; j < J0_S1; j++) // L12
         for (j = 0; j < J1_S1; j++) // L13
          for (j = 0; j < J2_S1; j++) // L14
            C[i][j]+=alpha*A[i][k]*B[k][j];// S1
```

Listing 3: Fully Distributed Original gemm Code with Loop Strip-Mined Twice

2.2 Limitation of the Current DSE Methods

Several frameworks [2, 13, 24, 36, 37, 39, 43] enable code transformations and pragma insertion, but have limited exploration spaces [24, 36, 37, 39, 43] or lack automatic DSE [2, 13]. Transformations are often restricted to loop property-based permutations [13, 24, 36, 37, 39] or rely on Pluto [1], which is optimized for CPUs but overlooks FPGA-specific techniques like pipelining and array partitioning. Additionally, Pluto minimizes memory transfer from off-chip to on-chip communication (Comm), which can limit parallelism on FPGAs, unlike our focus on reducing global latency (Lat). Meanwhile, other DSE frameworks [27, 29–31] explore pragmas insertion for fixed loop orders, but predicting friendly transformations beforehand is challenging. Exploring every possible transformation is also impractical for large design spaces. The key differences between the frameworks are summarized in Table 1.

We will now examine examples that highlight the limitations of current methods. For this, we use two different HLS DSE methods,

```
load_C(C, C_off_chip);
/**** Level 0 of S0 ****/
for (j0 = 0; j0 < 1; j0++) // L0
  for (i0 = 0; i0 < 4; i0++) // L1
/**** Level 1 of S0 ****/
    for (j1 = 0; j1 < 22; j1++) // L2
#pragma HLS pipeline
/**** Level 2 of S0 ****/
      for (j2=0; j2<10; j2++) { // L3
#pragma HLS unroll
        for (i2=0; i2<50; i2++) // L4
#pragma HLS unroll
          i = ...;
          C[i][j] *=beta; } // S0
/**** Level 0 of S1 ****/
for (k0 = 0; k0 < 48; k0++) // L5
  load_B_S1(B, B_off_chip, k0);
  load_A_S1(A, A_off_chip, k0);
  for (j0 = 0; j0 < 1; j0++) // L6
    for (i0 = 0; i0 < 4; i0++) // L7
/**** Level 1 of S1 ****/
      for (j1=0; j1 < 220; j1++) // L8
#pragma HLS pipeline
/**** Level 2 of S1 ****/
        for (j2=0; j2<1; j2++) // L9
#pragma HLS unroll
          for (i2=0; i2<50; i2++) // L10
#pragma HLS unroll
            for(k2=0;k2<5;k2++){ // L11
#pragma HLS unroll
              i = ...;
              C[i][j]+=alpha*A[i][k2]*B[k2][j];}// S1
store_C(C, C_off_chip);
```

Listing 4: Result found for gemm kernel using the NLP with the constraints: DSP = 2,000, On-chip mem = 320 kB

```
load_C(C, C_off_chip);
load_A(A, a_off_chip);
load_B(B, B_off_chip);
/**** Level 0 of S0 ****/
for (i0 = 0; i0 < 1; i0++) // L0
  for (j0 = 0; j0 < 1; j0++) // L1
/**** Level 1 of S0 ****/
    for (j1 = 0; j1 < 55; j1++) // L2
#pragma HLS pipeline
/**** Level 2 of S0 ****/
      for (i2 = 0; i2 < 200; i2++) // L3
#pragma HLS unroll
        for (j2=0; j2<4; j2++) { // L4
#pragma HLS unroll
          i = ...;
          C[i][j] *= beta;} // S0
/**** Level 0 of S1 ****/
for (i0 = 0; i0 < 1; i0++) // L5
  for (j0 = 0; j0 < 1; j0++) // L6
    for (k0 = 0; k0 < 60; k0++) // L7
/**** Level 1 of S1 ****/
      for (j1 = 0; j1 < 220; j1++) // L8
#pragma HLS pipeline
/**** Level 2 of S1 ****/
        for (i2=0;i2<200;i2++) // L9
#pragma HLS unroll
          for (j2=0; j2<1; j2++) // L10
#pragma HLS unroll
            for(k2=0;k2<4;k2++){ // L11
#pragma HLS unroll
              i = ...;
              C[i][j]+=alpha*A[i][k]*B[k][j];}//S1
store_C(C, C_off_chip);
```

Listing 5: Result found for gemm kernel using the NLP with the constraints: DSP = 6,840 On-chip mem = 7.2 MB

	Poly-Opt-HLS	POL-SCA/ Pluto	Auto-DSE/ HARP	Hetero-CL / Allo	NLP-DSE	Scale-HLS/ POM	Sisy-phus
Tiling	✓	✓	✗	✓	✗	Limit.	✓
Permutation	Limit.	Limit.	✗	✓	✗	Limit.	✓
Pragma Insert.	✓	✗	✓	✓	✓	✓	✓
Code Trans + Pragma Insert. (unified)	✓	✗	✗	✗	✗	✗	✓
Off-chip comm. gen.	✓	✓	✓	✓	✓	✗	✓
Enum. (AI, heuristics,...)	✗	✗	✓	Manual	✗	✓	✗
Objective	Comm	Comm	Lat	-	Lat	Lat	Lat

Table 1: Comparison of different frameworks

AutoDSE [31] and NLP-DSE [26, 27]. A comparison with HARP [30] and ScaleHLS [37] will be presented in Section 6. AutoDSE treats the source-to-source compiler Merlin [35] and the associated HLS tools as a black box, adjusting pragmas based on the identified bottlenecks from previous iterations. In contrast, NLP-DSE employs Nonlinear Programming DSE, utilizing a lower bound-based objective function to achieve high QoR within a short timeframe. Both of this DSE use the AMD source-to-source compiler Merlin [35]. The compiler employs different code transformations such as strip-mining via the *TILE* pragma or for partial loop unrolling. It typically avoids permutations, except when partially unrolling the two innermost loops, in which case the compiler strip-mines these loops and applies permutations to the resulting fully unrolled ones. A more detailed description of this compiler can be found in [27].

In the following examples, we use the HLS compiler Vitis 2023.2 with the unsafe-math option disabled. The targeted FPGA is the AMD/Xilinx Alveo U200. We evaluate two scenarios: the gemm kernel from Polybench [23], as shown in Listing 1, and a convolutional neural network (CNN) layer where the problem size and original loop order of CNN are I,J=256 (Output and Input Channels), H,W=224 (Height, Weight) and P,Q=5 (Filter Height and Weight).

The gemm kernel's original loop order, as illustrated in Listing 1, has a non-reduction loop at the center of the second statement (L3). This arrangement facilitates AutoDSE and NLP-DSE in unrolling this loop and pipelining the reduction loop (L2). In this scenario, if the reduction loop is pipelined **and** partially unrolled by a factor of uf, performance will degrade. While partial unrolling allows for parallel execution of the multiplications, it increases the pipeline depth due to the loop's dependencies, as the uf additions must still be performed sequentially. Consequently, when the reduction loop is pipelined, the initiation interval (II) will significantly increase, becoming a multiple of both uf and the latency of the reduction operation. This occurs because the next iteration of the loop can only begin after the uf additions are completed and the output is fully written. Their inability to further increase parallelism due to loop order limits the best designs to a throughput of 20 GFLOPs per second (GF/s). In such instances, a transformation becomes imperative to augment parallelization.

A single CNN layer with 6 loops results in an enormous design space that cannot be explored exhaustively. Loop permutations alone represent 6! = 720 possibilities, and tiling is required to manage array sizes exceeding on-chip memory capacity. While frameworks like AutoDSE and NLP-DSE achieve satisfactory throughput—42.15 GF/s and 31.80 GF/s, respectively—they are unable to explore the full design space, which includes both code transformations and pragma insertion, within a reasonable timeframe. For

example, running AutoDSE for all possible loop permutations would take around 500 days, and for NLP-DSE, it would take 106 days. Other frameworks, which apply code transformations based solely on loop properties, such as prioritizing reduction loops in the outermost positions, explore a much more restricted design space, limiting their potential for further optimization. This underscores the challenge of navigating such a vast design space and highlights gaps in current approaches.

2.3 Overview of Sisyphus

We introduce Sisyphus, tailored for affine programs with loop bounds and conditionals as affine functions [6, 8]. It defines a unified search space for parameters like pragma insertion, loop ordering, and tiling, avoiding backtracking or disjoint optimizations. Sisyphus efficiently explores this design space, overcoming limitations of frameworks that either overly restrict or partially explore it. Crucially, it ensures only valid transformations. The process involves creating a kernel-specific space and exploring it.

The first step involves creating a template with fully distributed code that includes three levels of loops, each targeting specific optimizations. To achieve this, we strip-mined each original loop twice, ensuring that one of these loops is included at each level (assuming the permutation is valid). As a result, each original loop can be utilized across all three levels. For instance, in Listing 1, the loop k (L2) is used in level 0 (L5) and in level 2 (L11) in Listing 4. Specifically, we have an innermost level for fine-grained unrolling, a level dedicated to pipelining, and another level focused on on-chip data caching based on available resources. As demonstrated in Listings 4 and 5, depending on the resource constraints specified by the user, we obtain two different code versions. For instance, a reduced on-chip memory requirement necessitates the partial transfer of arrays A and B (lines 17, 18 in Listing 4), leading to the use of a smaller on-chip buffer size that meets the user's constraints.

Once this template is generated, it establishes a search space where the unknowns consist of the loop bounds, loop ordering, buffer sizes for on-chip arrays, and the locations in the code where data transfers take place. Although other DSE methods can explore this space, we chose to use a cost model formulated as a Nonlinear Programming (NLP) problem as in NLP-DSE [27]. This approach allows us to efficiently navigate the theoretical space within seconds or minutes, leveraging accurate modeling made possible by compile-time analysis for affine programs. It is important to emphasize that exhaustive exploration using a simple cost model or synthesizing each design with HLS [31] would be impractical due to the sheer size of the search space. In contrast, the NLP approach enables us to explore this space quickly and efficiently, allowing for a more comprehensive investigation than traditional methods.

Sisyphus outperforms NLP-DSE [26, 27] by exploring a larger optimization space, including tiling and loop permutations, unlocking performance unattainable by NLP-DSE, which requires separate runs for each transformation. Automating the entire process, Sisyphus reduces exploration time and delivers superior results. It generates Vitis HLS pragmas with broader parameter support, like array partitioning, enhancing accuracy and design space coverage. Its high model precision enables the generation of at most one or two optimized designs, avoiding NLP-DSE's extensive iterations.

For gemm, Sisyphus significantly enhances parallelism by splitting and permuting loops without relying on the reduction loop. In this configuration, the NLP identifies the setup shown in Listing 5. Both pipelined loops achieve an initiation interval (II) of 1, resulting in a throughput of 210 GFLOPs per seconds (GF/s). The reduction loop (L11) is partially unrolled, allowing the multiplications to be executed in parallel while performing four additions sequentially. This small unrolling, selected by the NLP, deepens the pipeline, improving the quality of results (QoR) even though the reduction must remain sequential. For the CNN, the NLP selects all parameters in the design space, including loop order (exploring 6! possibilities), on-chip array size, parallelism levels, etc. This analysis completes in 13 minutes, achieving a throughput of 341 GF/s.

Thanks to a well-defined space that considers only legal transformations and exploration guided by a cost model formulated as an NLP, our framework can identify the theoretical optimum across the entire space and generate a design within seconds or minutes.

3 Unified Space

We proceed to develop a methodology that integrates code transformation, tile size selection and pragma insertion into a unified optimization framework. This involves implementing maximal distribution for code transformation, followed by the design and implementation of a template capable of executing code transformation, tiles size selection and pragma insertion simultaneously. We illustrate our code transformation process step by step using the example from Listings 1 to 5.

Input Sisyphus takes an affine C/C++ file as input and extracts a polyhedral representation using PoCC [22], gathering information like loop order, bounds, and array sizes. This data is used to create the template and generate the NLP. The framework is most effective for fully distributable and permutable codes but still works with restricted solution space when transformations are not fully legal.

Maximal Distribution To initiate the process, we distribute the program to maximize the distance between statements, promoting parallelization by reducing dependency overlap. Typically, this yields one statement per loop body within perfectly nested loops. Using ISCC [32], we explore scheduling options and verify transformation legality by preserving dependencies. Listing 2 shows the fully distributed version of the code from Listing 1.

Strip-mining After maximal distribution, we apply strip-mining to each loop twice. This process transforms the original loop, such as the loop L0 in Listing 2, into three loops with trip counts $I0\_S0$, $I1\_S0$, and $I2\_S0$ (L0, L1, L2 in Listing 3). The product of these three loop trip counts equals the original trip count, i.e., $I0\_S0 \times I1\_S0 \times I2\_S0 = 200$. Since strip mining is inherently legal, there is no necessity to validate the legality of this transformation. Subsequently, we can arrange the loops in different permutations if it is legal. After applying three levels of strip-mining, the code from Listing 2 is transformed into the code seen in Listing 3.

Creation of different levels Next, we consider possible permutations of the strip-mined loops. If these permutations are legal, we establish three loop levels. We ensure legality by checking dependency preservation using ISCC [32].

Level 2: The innermost level facilitates fine-grained unrolling (complete unrolling), which increases parallelism by duplicating

statements and can utilize a tree reduction if a reduction loop is unrolled and the option is enabled. For example, this is illustrated by loops L9, L10, and L11 in Listing 5.

Level 1: The middle level facilitates pipelining, enhancing throughput by overlapping loop iterations. In cases where loop permutations allow us to achieve perfectly nested loops suitable for pipelining, we restrict pipelining to a single loop to ensure efficient synthesis. Flattening the loops at Level 1 into a single loop and then pipelining the resulting loop introduces significant design complexity and leads to excessively long synthesis times. Thus, if a loop at Level 1 is pipelined, it implies that the trip counts of the other loops at the same level are reduced to one. For instance, in Listing 5, the loop j (L8) is pipelined for the statement S1. This results in the loops k and i (L10 and L11 in Listing 6) having a trip count of 1, enabling them to be eliminated from the code.

Level 0: Finally, the outermost level manages coarse-grained unrolling, tiling, and/or sequential execution. It not only controls parallelism at a coarse level but also determines the on-chip buffer size through tiling. As described in the following paragraph, the loop order at this level is included in the design space. This level is illustrated by loops L5, L6, and L7 in Listing 4. The loop bounds determine the on-chip buffer sizes for arrays A and B, transferred below loop L5. Resource constraints also influence the sequential execution of loop L7. If the loop body contains a single statement and loop permutation is feasible, we disable coarse-grained unrolling. This achieves equivalent results since our template allows fine-grained unrolling of the innermost loop.

Loop Permutation Loop order selection is not required for levels 1 and 2, as only a single loop is pipelined, and the innermost level (level 2) is fully unrolled, making the loop order irrelevant. Conversely, at the outermost level (level 0), **all permutations are considered**, and the NLP selects the permutation. In Listings 4 and 5, we can see that the NLP selected two different permutations for the statement S1 (loops L5, L6, and L7).

Fusion We have opted not to incorporate fusion into our model for several reasons. Firstly, our objective is to maximize parallelization for computation-bound kernels. Fusion may reduce or minimize the dependency distances between statements, limiting potential parallelization opportunities. Additionally, since our focus is on computation-bound kernels, the primary bottleneck lies in computation. Therefore, we are willing to incur a minor cost, even if it involves transferring a tiled array multiple times.

Overal Template In the context of our example, these transformations yield the code presented in Listing 6, with each level corresponding to specific transformations. The iterator of the loop retains its original name, with an added number indicating the level. For example, loop L14 in Listing 6 refers to loop L2 in Listing 1 at level 2 of the strip mining. Notably, our framework is capable of preserving the original loop order, allowing it to retain the existing loop structure if the code is already optimized.

Design Space The challenge is to determine loop trip counts (e.g., $J0\_S0$) and level 0 loop order while meeting DSP and on-chip memory constraints. Hence, the trip counts corresponding to fully unrolled loops must be constrained to avoid over-utilization of DSP resources. The on-chip buffers size must not exceed the on-chip memory capacity. Consequently, for a large problem size, the outermost level will enable control over the buffer sizes.

```
1  int perm_S0[2] // Permutation order (0 for i0, 1 for j0)
2  int perm_S1[3] //Array to store permutation order (i0,j0,k0)
3  int lb_S0[2] = {I0_S0, J0_S0}; // loop bounds for S0
4  int lb_S1[3] = {I0_S1, J0_S1, K0_S1}; // loop bounds for S1
5  // possibilities to cache on-chip A, B and C
6  /********* Level 0 of S0 *********/
7  for (L0_S0=0; L0_S0 < lb_S0[perm_S0[0]]; L0_S0++) // L0
8  // possibilities to cache on-chip C
9   for (L1_S0=0; L1_S0 < lb_S0[perm_S0[1]]; L1_S0++) // L1
10  // possibilities to cache on-chip C
11 /********* Level 1 of S0 *********/
12   for (j1=0; j1 < J1_S0; j1++) // L2
13    for (i1=0; i1 < I1_S0; i1++) // L3
14 #pragma HLS pipeline
15 /********* Level 2 of S0 *********/
16     for (i2=0; i2 < I2_S0; i2++) // L4
17 #pragma HLS unroll
18      for (j2=0; j2 < J2_S0; j2++) // L5
19 #pragma HLS unroll
20       C[i][j]*=beta; // S0
21 /********* Level 0 of S1 *********/
22 for (L0_S1=0; L0_S1 < lb_S1[perm_S1[0]]; L0_S1++) // L6
23 // possibilities to cache on-chip A, B and C
24  for (L1_S1=0; L1_S1 < lb_S1[perm_S1[1]]; L1_S1++) // L7
25  // possibilities to cache on-chip A, B and C
26   for (L2_S1=0; L2_S1 < lb_S1[perm_S1[2]]; L2_S1++) // L8
27   // possibilities to cache on-chip A, B and C
28 /********* Level 1 of S1 *********/
29    for (j1=0; j1 < J1_S1; j1++) // L9
30     for (k1=0; k1 < K1_S1; k1++) // L10
31      for (i1=0; i1 < I1_S1; i1++) // L11
32 #pragma HLS pipeline
33 /********* Level 2 of S1 *********/
34       for (i2=0; i2 < I2_S1; i2++) // L12
35 #pragma HLS unroll
36        for (j2=0; j2 < J2_S1; j2++) // L13
37 #pragma HLS unroll
38         for (k2=0; k2 < K2_S1; k2++) // L14
39 #pragma HLS unroll
40          i = i0 * I1_S1 * I2_S1 + i1 * I2_S1 + i2;
41          j = j0 * J1_S1 * J2_S1 + j1 * J2_S1 + j2;
42          k = k0 * K1_S1 * K2_S1 + k1 * K2_S1 + k2;
43          C[i][j]+=alpha*A[i][k]*B[k][j]; // S1
```

Listing 6: Template automatically generated for the gemm input code (Listing 1)

4 NLP

We automatically generate constraints and variables for a nonlinear program to explore the theoretical solution space outlined in Section 3, adapting the approach from [26, 27] to our context. Latency is estimated with optimistic DSP utilization, assuming perfect resource reuse, but users can adjust the DSP limit or choose a pessimistic scenario with no reuse. Users can also customize on-chip memory size, maximal array partitioning, and latency/resources per operation, ensuring adaptability to different platforms and compilers. To gather all the required information for the NLP, we utilize PoCC [22] to extract the intermediate representation.

4.1 Variables

Table 2 defines the sets, variables, and constants utilized in our NLP formulation.

4.2 Constraints

We describe each constraint's meaning and implications with examples from the paper's listings.

Trip Count Equation 1 constrains the trip count of each loop, ensuring that the product of the trip counts equals the original trip

count. The three loops after strip-mining, e.g., L6, L7, and L8 in Listing 3, must be equivalent to the original loop L2 in our example from Listing 2. Therefore, we require $I0\_S0 \times I1\_S0 \times I2\_S0 = 200$.

$$\forall l \in \mathcal{L}, \prod_{v \in \mathcal{V}} TC_{l,v} = TC_l \tag{1}$$

Set	Description
$\mathcal{L}, \mathcal{A}, \mathcal{S}$	the set of loops, arrays and statements
$\mathcal{S}_1$	the set of statements alone in a loop body
O_s	the list of operations of the statements s
$\mathcal{L}_s$	the set of nested loops which iterate the statement s
$\mathcal{L}_s^{red}$	the set of reduction loops iterating the statement s
C_{a_d}	the set of loops iterating the array a at dimension d
$\mathcal{V}$	Strip mining level: 0 for coarse-grained/sequential, 1 for pipeline, 2 for fine-grained unrolled
$AP_{a,d}$	Array Partition for the array a in dimension d
Constant	**Description**
TC_l	Trip Count of the loop l before strip-mining
II_l	II of the loop l
IL_{par}, IL_{red}	Iteration Latency of the operations without (par) and with (red) dependencies of the statement s
$DSP_{s_{op}}$	Number of DSP used for the statement s for the operation op (This number accounts for the number of times the statement is replicated due to unrolling)
$DSP_{available}$	Number of DSP available for the FPGA used
max_{part}	Maximum array partitioning defined by the user
$ft\_arr_a\_loop_l$	Footprint of the array a if transferred to on-chip after the loop l
$reuse_{opt}$	Boolean for optimistic reuse
Variable	**Description**
$tc_{l,level}$	TC of the loop l for the level of strip-mining
$loop_l\_UF$	Coarse-grained unroll factor of the loop l at level 0
$loop_l\_pip$	Boolean to know if the loop l is pipelined at level 1
$cache_l\_arr_a$	Boolean to know if the array a is transferred on-chip after the loop l at level 0
$perm_s$	ID of the permutation of the statement s
$burst_a$	The maximum bitwidth at which the array a can be transferred from off-chip to on-chip

Table 2: Overview of the sets, constants and variables employed in formulating the NLP

Pipeline Constraints 2–5 facilitate the selection of loop pipelining. Constraint 2 ensures each loop has a flag indicating if it is pipelined. For example, in Listings 4 and 5, $S1$ shows loop $j1$ (L8) is pipelined, while its original loop is j (L3 in Listing 1), making $loop_j\_pip = 1$. As per Section 3, only one loop at level 1 can be pipelined, requiring the sum of pipeline flags must be less than or equal to 1 (Eq. 4). Non-pipelined loops must have a trip count of 1 (Eq. 3). For instance, in Listing 6, only loop L9 is pipelined at level 1 of $S1$, so $loop_j\_pip = 1$, while $loop_i\_pip = 0$ and $loop_k\_pip = 0$. Loops L10 and L11 have trip counts of 1 and can be removed.

Eq. 5 calculates the initiation interval (II) based on loop properties and reduction operation latency. In Listings 4 and 5, pipelined loops are non-reduction loops with no dependencies, giving $II = 1$. However, if loop L10 in Listing 6 were pipelined, it would involve a reduction requiring $II \geq IL_+$, as the write of iteration k must complete before the read of $k + 1$.

$$\forall l \in \mathcal{L}, loop_l\_pip \in \{0, 1\} \tag{2}$$

$$\forall l \in \mathcal{L}, (1 - loop_l\_pip) \times TC_{l,1} == 1 \tag{3}$$

$$\forall s \in \mathcal{S}, \sum_{l \in \mathcal{L}_s} loop_l\_pip \leq 1 \tag{4}$$

$$\forall s \in \mathcal{S}, II_s = \sum_{l \in \mathcal{L}_s} loop_l\_pip \times II_l \tag{5}$$

Coarse-grained unrolling Coarse-grained unrolling applies only to non-reduction loops (Eq. 6). For instance, loop L3 in Listing 2 cannot be unrolled as it is a reduction loop. The unroll factor (UF) must divide the loop trip count and not exceed it (Eq. 7 and 8). However, as noted in Section 3, coarse-grained unrolling is skipped if the loop has only one statement (Eq. 9), as in Listing 1.

$$\forall s \in \mathcal{S}_1, \forall l \in \mathcal{L}_s^{red}, loop_l\_UF = 1 \tag{6}$$

$$\forall l \in \mathcal{L}, loop_l\_UF \% TC_{l,0} == 0 \tag{7}$$

$$\forall l \in \mathcal{L}, loop_l\_UF \leq TC_{l,0} \tag{8}$$

$$\forall s \in \mathcal{S}_1, \forall l \in \mathcal{L}_s, loop_l\_UF = 1 \tag{9}$$

On-chip memory Constraints 10 and 11 ensure each array, represented by a boolean (Eq. 10), is cached at a single location per loop body, with only one boolean true per loop (Eq. 11). In Listing 6, array A can be cached before loops (line 5) or after L6, L7, or L8 (lines 23, 25, 27). In Listing 5, A is cached on-chip, matching the original array's size (line 2). In Listing 4, A is cached on-chip at line 18 with a smaller footprint. The on-chip footprint depends on the transfer location. For Listing 4, A is transferred below loop L5 (trip count 48), so its footprint is $200 \times \frac{240}{48}$.

$$\forall l \in \mathcal{L}, \forall a \in \mathcal{A}, cache_l\_arr_a \in \{0, 1\} \tag{10}$$

$$\forall a \in \mathcal{A}, \forall s \in \mathcal{S}, \sum_{l \in \mathcal{L}_s}, cache_l\_arr_a = 1 \tag{11}$$

$$\sum_{a \in \mathcal{A}} \sum_{l \in \mathcal{L}} cache_l\_arr_a \times ft\_array_a\_loop_l \leq Mem \tag{12}$$

Array partitioning Unrolling is applied only if all required data can be accessed in parallel, ensured by pragma *array_partitioning*. For each array a and dimension d, the cyclic partitioning factor must exceed the unroll factor (Eq. 13) to ensure data is on separate banks. If the array is reused, partitioning must align with each unroll factor iterating through the dimension (Eq. 14). $AP_{a,d}$ represents the cyclic partitioning factor used in the code, with the total partitioning per array constrained by the user-defined limit (Eq. 15). For example, in Listing 5, C is unrolled 200 times in the first dimension (lines 11, 12) and 4 times in the second (lines 13, 14) for the first loop body, and 200 times in the first dimension (lines 25, 26) for the second. This gives $AP_{C,1} = 200$, $AP_{C,2} = 4$, and $200 \times 4 = 800 \leq max_{part}$, with $max_{part} = 1,024$.

$$\forall a \in \mathcal{A}, \forall d \in \mathbb{N}, \forall l \in C_{a_d}, AP_{a,d} \geq TC_{l,2} \tag{13}$$

$$\forall a \in \mathcal{A}, \forall d \in \mathbb{N}, \forall l \in C_{a_d}, AP_{a,d} \% TC_{l,2} == 0 \tag{14}$$

$$\forall a \in \mathcal{A}, \prod_{d \in \mathbb{N}} AP_{a,d} \leq max_{part} \tag{15}$$

DSP utilization To manage DSP utilization, we use a user-defined flag, $reuse_{opt}$. If $reuse_{opt} = 1$, optimistic reuse is assumed: identical operations reuse DSPs across loop bodies, with total usage given by the maximum DSPs per operation (Eq. 16). However, if the operations differ, as discussed in [26, 27], no reuse is assumed. If $reuse_{opt} = 0$, no reuse is assumed, and DSPs are summed for each statement (Eq. 17). In both cases, total usage must not exceed available DSPs (Eq. 18 and 19). If a multiplication ($*$) uses 3 DSPs and an addition (+) uses 2 DSPs, in Listing 5, S0 needs $200 \times 4 \times 3$ DSPs for multiplications, while S1 requires $200 \times 4 \times 3 + 200 \times 4 \times 3 + 200 \times 4 \times 2$ DSPs for two multiplications and one addition. With optimistic reuse, DSP usage for multiplication is $\max(200 \times 4 \times 3, 2 \times 200 \times 4 \times 3) + 200 \times 4 \times 2$, while pessimistic reuse results in $200 \times 4 \times 3 + 2 \times (200 \times 4 \times 3) + 200 \times 4 \times 2$.

$$DSPs\_used_{opt} = \sum_{op \in \{+,-,*,/\}} \max_{s \in \mathcal{S}} (DSP_{s_{op}} / II_s) \tag{16}$$

$$DSPs\_used_{pes} = \sum_{op \in \{+,-,*,/\}} \sum_{s \in \mathcal{S}} (DSP_{s_{op}} / II_s) \tag{17}$$

$$reuse_{opt} \times DSPs\_used_{opt} \leq DSP_{available} \tag{18}$$

$$(1 - reuse_{opt}) \times DSPs\_used_{pes} \leq DSP_{available} \tag{19}$$

4.3 Objective Function

We use the objective function similar to the approach described in [26, 27], tailoring it to meet the specific needs of our problem. Here, Lat_2^s corresponds to the fine-grained unrolled level (level 2) for the statement s, Lat_1^s denotes the pipeline tile that incorporates the fine-grained unrolled level (level 1), and recursively, Lat_0 encompasses Lat_1, enabling coarse-grained unrolling and facilitating on-chip data caching (level 0).

The memory latency Lat_{mem}^s for a statement s corresponds to the time required to transfer the array from off-chip to on-chip and back for the arrays needed by a statement s. We assume the memory transfer is pipelined, allowing the transfer of one data element (with a maximum bitwidth of 512 bits) per cycle. Therefore, the latency at a given level is calculated as the array footprint divided by the burst size (bitwidth used for transferring the array), multiplied by the trip count of the loop that iterates over the function responsible for the transfer, based on the selected permutation of level 0. Additionally, if an array is fully transferred on-chip for a statement s and reused by a subsequent statement s', no further transfers are needed. The burst size for an array must divide its last dimension. For instance, if we have an array $float\ A[512][20]$, the maximum burst size for this array is 128 bits. This is because 128 is the largest power of 2 that can evenly divide the size of the last dimension (20×32 bits).

$$\begin{cases} Lat_2^s = IL_{par} + IL_{seq} \times \prod_{l \in \mathcal{L}^{red}} \times \log_2(TC_{l,2}) \\ Lat_1^s = Lat_2^s + II \times (TC_{l,1} - 1) \\ Lat_0^s = \prod_{l \in \mathcal{L}} \frac{TC_{l,0}}{loop_l\_UF} \times Lat_1^s \\ L_{mem}^s = \sum_{l \in \mathcal{L}} \max_{a \in \mathcal{A}} (cache_l\_arr_a \times ft\_array_a\_loop_l \\ \qquad\qquad \times TCs(perm_s)/burst_a) \\ Lat_s = Lat_0^s + L_{mem}^s \end{cases}$$

Finally, the objective function $obj\_func = \sum_{s \in \mathcal{S}} Lat_s$ is defined as the sum of the latencies of each loop body.

5 Post-Optimization and Customization

The NLP determines the trip counts, array partitioning, and on-chip buffer sizes. Data transfer functions between off-chip and on-chip memory are automatically generated with the maximum burst size. Further optimizations are also applied to improve QoR.

Optimization for Non-Constant Trip Count To optimize loops with non-constant trip counts, we implement a code transformation that preserves the NLP's estimation while simplifying compilation for the HLS compiler. We achieve this by replacing loops with non-constant trip counts with the maximal trip count computed using PoCC [22]. Next, we replace all statements iterated by this loop with calls to a function. For example, instead of using $C[i][i] += B[i][i]$, we replace it with $C[i][j] = f(C[i][j], B[i][i])$, where the function f ensures compliance with the constraints of the non-constant trip count loop. This function returns the computed value if the constraints are satisfied; otherwise, it simply returns the original output value. This technique helps to reduce compilation time and achieve designs with a good QoR.

Grouping Data Transfers In the NLP, we already account for parallel transfers of arrays when they are transferred at the same level in the code (e.g., within the same loop). However, we do not consider parallel transfers between arrays belonging to different loop bodies. In our example, we first manage the transfer of arrays required for initializing the matrix, transferring the arrays for multiplication only after the initialization is complete. However, FPGA DRAM banks allow parallel transfers of different arrays. Consequently, we can optimize load and store operations during post-processing to improve memory transfer overlap.

Code Transformation for HLS To help the HLS compiler pipeline loops with the correct initiation interval (II), we simplify reductions at level 2. A variable accumulates the reduction within the pipeline, which is then added to the output. Additionally, we use the *loop_flatten off* pragma on all reduction loops above the pipeline. Although this adjustment could theoretically improve the QoR, in practice, flattening these loops can complicate the code and potentially increase the initiation interval (II) of the pipeline.

Customization Our framework is designed to be explainable and user-friendly, requiring the user to provide only the affine C/C++ code of the kernel to be optimized and the available resources. To fully benefit developers, our NLP formulation allows the template to be completely customized to the user's needs. Each variable in the design space can be defined by the user if they have specific constraints. For example, in the gemm template (Listing 6), the user can set the permutations for level 0 while allowing the NLP to explore other parameters. Additionally, they can specify the on-chip size of matrix A by directly defining a variable in the NLP, allowing for exploration within the defined constraints.

6 Evaluation

The goal of this evaluation is to demonstrate how effectively our approach applies code transformations and inserts hardware directives while respecting user-defined resource constraints. We compared our method against four other frameworks: AutoDSE [31], NLP-DSE [27], HARP [30], and ScaleHLS [37]. Since generating a bitstream without significant manual intervention is challenging for some frameworks, we first demonstrate our ability to achieve a

high-quality design by utilizing all available resources—ensuring fair comparison with the other frameworks—without considering place and route. We generate HLS reports using the *vitis-flow*, which accounts for memory transfers between off-chip and on-chip, necessitating the inclusion of memory transfer steps in ScaleHLS's code. Thus, we show that our designs are synthesizable by targeting the resources of a single SLR, which eliminates the need for manual partitioning. Due to the time required for bitstream generation, we restrict our evaluation to a subset of the kernels.

6.1 Setup

We conducted experiments using kernels from Polybench/C 4.2.1 [23], a CNN kernel, and matrix multiplication tasks similar to those in the BERT transformer model. All computations used single-precision floating-point data for direct comparison with frameworks like AutoDSE, NLP-DSE, and ScaleHLS, using medium-sized Polybench/C datasets. We also compare our approach with HARP [30], which enhances HLS design exploration using a Graph Neural Network (GNN) model to predict HLS tool behavior. HARP explores this space for one hour and synthesizes the top 10 designs. We use the medium-sized Polybench kernels found in their training set with double-precision floating-point data types, aiming to deploy HARP under optimal conditions. As detailed in [27], HARP's effectiveness is not universally applicable without the necessary fine-tuning or training, particularly when confronted with diverse problem sizes.

We selected these problem sizes to highlight how our method performs in transforming code and inserting pragmas in cases where full unrolling is not feasible. We also demonstrate the effectiveness of our framework in tiling large tasks such as CNN and bert_3072_100, ensuring that only a small portion (a tile) of each array fits on-chip, in line with memory constraints. For the CNN kernel, the problem size and loop dimensions are the same as described in Section 2. For bert_n_m matrix multiplication, the dimensions are I = n, J = m, K = n (reduction).

We used the AMD/Xilinx Vitis HLS compiler [34] to demonstrate our method's effectiveness, generating reports with Vitis 2023.2. When enabling tree reduction, we selected the "funsafe math optimizations" option to allow commutative and associative reduction operators for logarithmic time reductions. For our hardware setup, we targeted the Xilinx Alveo U200 device operating at 250 MHz. We utilized the commercial solver Gurobi 11.0.0 [10] with AMPL.

6.2 Experimental Evaluation

To ensure a fair comparison, we modified kernels with non-constant trip counts for all benchmarks, as detailed in Section 5, except for ScaleHLS, which already handles such loops. We generated AutoDSE's design space using the *ds_generator* command, applying all trip count factors for unroll and tile sizes. Since AutoDSE lacks array partitioning, its design space is slightly larger. AutoDSE was run with a 1,000-minute exploration timeout and a 180-minute HLS synthesis timeout. For NLP-DSE, we followed the original paper's parameters [27]. HARP was run with the authors' settings [30] using Vitis 2021.1 and tree reduction enabled. ScaleHLS assumes that all kernel data are already located in on-chip memory. To provide a fair comparison, we modified their code to handle off-chip to on-chip memory transfers. We optimized these transfers to 512 bits per cycle, with all arrays transferred in parallel. Each array was assigned to a different DRAM bank. *However, due to the extensive manual modifications required, we did not extend this comparison to all kernels.* For Sisyphus evaluation, we set the Gurobi solver timeout to 14,400 seconds per design. By default, we assume optimistic resource reuse between non-parallel statements. If DSP utilization exceeds limits, we rerun the NLP with pessimistic assumptions (no resource reuse) and regenerate the design. The maximum partitioning constraint, max_{part}, was set to 1024, per AMD/Xilinx guidelines.

Each synthesis starts with a C-simulation to verify correctness. We generate Vitis reports using the *vitis-flow*, which counts memory transfers for all frameworks and considers the resources of the entire FPGA. For bitstream generation, we consider only one SLR to eliminate any manual intervention. We consider 60% of the resources of a single SLR to quickly identify a synthesizable design, minimizing the number of times AutoDSE needs to be run.

6.3 Comparison of the Methods

6.3.1 AutoDSE, NLP-DSE and Scale-HLS with tree reduction. Table 3 provides a comparison among AutoDSE (A-DSE), NLP-DSE (N-DSE), Scale-HLS (S-HLS) and Sisyphus (ours) with tree reduction. The **T** column indicates the specific transformations applied, with D representing distribution, T for data tiling, and P for permutation. The column **Perf. Impr. of ours framework vs.** displays the performance improvement of our framework compared to the three different frameworks.

		GF/s	Perf. Impr. of ours framework vs.		
Kernel	**T**	**Ours**	**A-DSE**	**N-DSE**	**S-HLS**
2mm	D,P	175.49	431.17x	1.42x	5.50x
3mm	D,P	140.83	80.75x	1.02x	3.86x
atax	D	1.96	0.99x	1.00x	1.23x
bicg	D,P	1.96	1.97x	1.97x	1.18x
doitgen	D,P	54.05	1.35x	2.71x	8.45x
gemm	D,P	203.39	1.84x	1.55x	5.81x
gemver	P	17.62	4.58x	1.74x	9.39x
gesummv	D	1.96	0.99x	1.00x	0.71x
mvt	P	13.24	1.70x	1.70x	1.26x
symm	D,P	240.50	8.28x	8.28x	2,704.51x
syr2k	D,P	254.69	5.52x	1.86x	174.47x
syrk	D,P	161.20	6.55x	2.32x	358.95x
trmm	D,P	148.53	8,345.10x	8,312.42x	2,376.91x
Average		108.88	683.91x	641.46x	434.78x
Geo Mean		40.13	9.17x	3.47x	16.15x

Table 3: Throughput (GF/s) Comparison with Tree Reduction Using Vitis HLS

Across all evaluated kernels, we consistently achieve performance that is comparable to or better than alternatives, except for the atax, and gesummv kernels, where we observe a slowdown of less than 5%. For 3mm, symm and trmm we had to apply the constraint indicating no reuse (Eq. 17), as the constraint with optimistic reuse (Eq. 16) resulted in kernels with resource over-utilization. For Gesummv, ScaleHLS is 1.41x faster. However, 91% of our design's execution time is spent on data transfers. ScaleHLS benefits from

512-bit data transfers, which we manually applied for testing. Our framework uses 64-bit chunks, the largest bitwidth that divides the array. With 512-bit transfers, our framework could achieve 14.50 GF/s. For other memory-bound kernels like Atax and Gesummv, the slight performance differences observed with other frameworks are largely due to how Merlin handles memory transfers.

Sisyphus averages a speedup of 683.91x, 641.46x and 434.78x over AutoDSE, NLP-DSE and Scale-HLS respectively, across the evaluated kernels. In terms of geometric mean, Sisyphus achieves a speedup of 9.17x, 3.47x and 16.15x.

Preserve the original schedule when necessary For atax, Sisyphus preserves the original schedule and inserts pragmas, effectively maintaining the existing schedule while efficiently incorporating pragmas, similar to two DSEs focused on pragma insertion.

Code transformation for memory-bound kernel For mvt and gemver, both memory-bound kernels, effective optimization entails loop permutations to prevent redundant array transfers and accommodate array partitioning within the constraints of the AMD/Xilinx compiler's 1024 limit. Sisyphus adeptly manages these loop permutations and pragma insertions, guaranteeing optimized performance for these kernels. For bicg, fully distributing and arranging the loops appropriately can enhance parallelization capabilities.

Allow to achieve a parallelism not achievable without loop transformation For doitgen, symm, syrk, syr2k, and trmm, the original schedule imposes constraints on the achievable level of parallelism. This limitation highlights the performance enhancements made possible by Sisyphus. We observe QoR improvements for these kernels because the code transformations enable a combination of loop unrolling that cannot be achieved even with maximal loop distribution, as each loop can be partially unrolled.

6.3.2 AutoDSE and NLP-DSE without tree reduction. Transformations without tree reduction, which may be needed to preserve floating-point data precision, must be different since the optimizations differ when tree reductions are not involved. Table 4 provides a comparison among AutoDSE (A-DSE), NLP-DSE (N-DSE) and Sisyphus (ours) without tree reduction.

		GF/s	Perf. Impr. Ours vs.	
Kernel	**T**	**Ours**	**A-DSE**	**N-DSE**
2mm	D,P	171.07	817.42x	1.44x
3mm	D,P	140.11	511.64x	1.04x
atax	D	1.86	0.98x	1.00x
bicg	D,P	1.86	1.92x	1.93x
doitgen	D,P	54.05	5.46x	2.71x
gemm	D,P	210.61	9.25x	4.01x
gemver	P	15.01	37.28x	8.73x
gesummv	D,P	1.81	0.97x	0.97x
mvt	P	9.75	1.42x	1.42x
symm	D,P	240.55	8.28x	7.32x
syr2k	D,P	429.34	17.61x	3.14x
syrk	D,P	333.03	13.54x	4.78x
trmm	D,P	130.90	7,362.68x	7,333.85x
Average		133.84	676.03x	567.10x
Geo Mean		41.62	18.50x	4.49x

Table 4: Throughput (GF/s) Comparison without Tree Reduction Using Vitis HLS

Sisyphus achieves an average speedup of 676.03x over AutoDSE and 567.10x over NLP-DSE across kernels without tree reduction, with geometric means of 18.50x and 4.49x, respectively. Performance improvement is higher without tree reduction, especially for gemm (Listing 5) and gemver, as optimizing reduction loops becomes harder, increasing the need for code transformations. The difference between syrk and syr2k with tree reduction occurs because the HLS compiler achieves II=2 instead of the expected II=1, while without tree reduction, II=1 is achieved.

6.3.3 AutoDSE and NLP-DSE for Bert and CNN. Table 5 compares AutoDSE (A-DSE), NLP-DSE (N-DSE), and Sisyphus (ours).

The **TR** column shows the tree reduction status. For bert_100_64, we observed a minor decrease in QoR, attributed to the NLP selecting a configuration where the pipeline was not applied as expected.

			GF/s	Perf. Impr. Ours vs.	
Kernel	**TR**	**T**	**Ours**	**A-DSE**	**N-DSE**
bert_100_64	1	D,P	81.22	0.99x	0.95x
bert_100_64	0	D,P	76.76	1.09x	1.09x
bert_100_768	1	D,P	218.08	1.12x	1.19x
bert_100_768	0	D,P	216.95	1.13x	1.13x
bert_100_3072	1	D,P	241.59	1.21x	1.21x
bert_100_3072	0	D,P	241.06	1.22x	1.22x
bert_3072_100	1	D,P,T	314.89	21.85x	24.29x
bert_3072_100	0	D,P,T	82.41	989.65x	1,156.02x
cnn	1	D,P,T	314.15	7.38x	8.23x
cnn	0	D,P,T	341.20	8.09x	10.73x
Average			212.83	103.37x	120.61x
Geo Mean			185.32	4.38x	4.69x

Table 5: Throughput (GF/s) Comparison Using Vitis HLS

Code transformation with tiling The selection of loop order and tile size is critical for determining which array sizes, particularly for large problem size like bert_3072_100 and CNN layers, should be cached on-chip and where they should be transferred. In these instances, the NLP aims to enhance data reuse and minimize unnecessary memory transfers while identifying the theoretical optimal balance with parallelism.

6.3.4 HARP. In Table 6 we compare the throughput and resource utilization (BRAM, DSP, LUT, FF) achieved with HARP [30].

	GF/s		Resource Utilization (%) BRAM,DSP,LUT,FF		Perf. Impr.
Kernel	**HARP**	**Ours**	**HARP**	**Sisyphus**	
atax	1.72	1.96	78,52,49,50	38,65,53,33	1.14x
bicg	0.92	1.96	75,25,35,36	38,65,48,31	2.13x
gemm	125.59	87.86	29,80,55,40	32,48,36,23	0.70x
gemver	1.66	9.42	29,28,35,19	34,8,31,11	5.69x
mvt	7.07	8.29	40,78,43,30	53,44,57,30	1.17x
Average	27.39	20.71			1.45x
Geo Mean	4.71	6.28			1.33x

Table 6: Comparison with HARP Using Vitis HLS

The enhanced performance seen in both bicg and gemver can be attributed to the same underlying factor observed in the comparison with AutoDSE and NLP-DSE. Regarding mvt, HARP produced

comparable performance to our own due to their use of the Merlin compiler, which facilitates loop permutations when partially unrolling consecutive loops. This allows them to find the same schedule that our framework identified. However, for gemm, the design discovered by HARP falls outside our specified parameter space, due to our restriction on maximum array partitioning.

6.4 Single SLR Onboard Evaluation

	GF/s	RU: BRAM,DSP,LUT(K),FF(K)		Perf. Impr.
Kernel	**Ours**	**Ours**	**AutoDSE**	
2mm	30.57	510,556,213,276	353.5,963,287,292	76.98x
3mm	29.89	611,984,230,300	470,1117,278,306	72.34x
atax	1.03	450,173,240,250	630.5,452,170,212	4.63x
bicg	1.02	451,173,238,265	867.5,196,168,217	1.80x

Table 7: Comparison of Onboard Evaluation for 1 SLR

The resource utilization (RU) is detailed in the thousands for both LUTs and FFs. URAM utilization is excluded as no kernels use it. And the last column show the performance improvement we achieve with Sisyphus over AutoDSE. For the kernels evaluated on board, we can see that our generated design is both faster and consumes fewer resources compared to AutoDSE.

6.5 Latency Estimation and Scalability

Implementing the post-optimization strategy from Section 5 results in a loss of the lower bound discussed in [26, 27], mainly because we exclude communication overlap modeling between loop bodies in the NLP to avoid significantly increasing search time. We also do not incorporate the *loop_flatten* pragma, even if the compiler may automatically apply it for loops without reductions, as we deactivate it for reduction loops. Although this leads to a lower bound loss, we prefer not to depend on an optimization that may not be consistently implemented. As noted in Section 5, our post-optimization focuses primarily on memory transfer overlap, so our final latency values do not directly reflect the model's initial predictions. However, our model achieves (when predicting the latency of the final design generated) an average prediction error of 8% across evaluated designs, with a 3% error in geometric mean. For certain designs, like syrk and syr2k, where the compiler achieves an II=2 rather than the expected II=1, prediction errors are higher (49% and 32%, respectively) due to an optimistic estimation of II.

The average solving time of the NLP was 455 seconds, while the geometric mean stood at 6.7 seconds. Only the kernel 3mm with tree reduction timeout at 4h. 32/36 NLP's problem are solve in less than 1 minutes. It is important to note that the solver provides a feasible solution early in the process, though it may not be theoretically optimal. As the search continues, the solution improves, and once the solver completes, the solution is theoretically optimal. Therefore, it is possible to set a timeout to obtain a feasible, but not optimal, solution more quickly.

7 Related Work

Various code transformations have been devised for CPUs [1], GPUs [33], FPGAs [3, 15, 17–19, 25, 42, 43] or for all [44]. While code transformations for CPUs and GPUs yield remarkable results tailored to their respective architectures, they may not be inherently suitable for FPGA targets aimed at achieving high parallelism. Regarding [42, 43], they employ Pluto on different scopes of the kernel for code transformation, yet its pragma insertion capabilities are limited, making it incomparable to our work. Conversely, [3, 15, 17–19, 25] pursue a distinct objective from ours. [25] aims to minimize communication between off-chip and on-chip, which results in better QoR than ours for memory-bound kernels. The works [3, 15, 17–19] concentrate on code transformations aimed at enhancing pipelining techniques. However, these objectives may not be suitable for computation-bound kernels requiring high levels of parallelization.

Ansor [44] achieves impressive results for CPU and GPU architectures; however, there are notable differences with our approach. Ansor requires multiple evaluations to perform its design-space exploration (DSE) and update its cost model, which accelerates the DSE process but is not scalable for FPGA, where a single evaluation can take minutes or even hours. Furthermore, Ansor's approach to code transformations and pragma insertions (which are more tailored for CPU/GPU) addresses these problems separately, potentially resulting in suboptimal designs.

The choice of tile sizes significantly impacts the QoR. In line with our methodology, [16, 20] employ a cost model to determine the tile size. However, while [20] focuses on minimizing communication overhead, our approach differs. On the other hand, [16] investigates tiles size selection for Convolutional Neural Networks (CNNs) with three-level CPU caching.

8 Limitations

Sisyphus is designed to optimize affine loop nests. For applications with non-affine code, we can partition the code into affine and non-affine sections. Sisyphus handles affine regions effectively, while tools like AutoDSE [31] or HARP [30] can optimize non-affine parts. The optimized sections are then integrated using a dataflow model, either with FIFOs for data exchange or by employing TAPA [9], which facilitates dataflow programming. This combined approach ensures that both affine and non-affine portions of the application are handled in an optimal manner, achieving a balance between different types of optimizations and seamlessly integrating them for improved overall performance.

9 Conclusion

In this paper, we introduced Sisyphus, a unified framework that streamlines hardware design through HLS. By integrating code transformation, pragma insertion, and tile size selection into a single optimization, Sisyphus addresses challenges in existing HLS workflows, which often rely on manual and disjointed methods.

Sisyphus defines a design space in the form of a template. Using NLP, it efficiently explores this space, quickly identifying transformations and pragma configurations to achieve designs with good QoR. Our evaluation demonstrates that Sisyphus outperforms AutoDSE, NLP-DSE, and ScaleHLS in terms of quality of result.

Acknowledgments - This work was supported by the NSF award #CCF-2211557. It is also supported by CDSC industrial partners and the AMD[1]/HACC Program.

[1]J. Cong has a financial interest in AMD.

References

[1] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 101–113. https://doi.org/10.1145/1375581.1375595

[2] Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhichen Zeng, Mengjia Dai, and Zhiru Zhang. 2024. Allo: A Programming Model for Composable Accelerator Design. *Proc. ACM Program. Lang.* 8, PLDI, Article 171 (jun 2024). https://doi.org/10.1145/3656401

[3] Young-kyu Choi and Jason Cong. 2018. HLS-Based Optimization and Design Space Exploration for Applications with Variable Loop Bounds. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1145/3240765.3240815

[4] Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. 2022. FPGA HLS Today: Successes, Challenges, and Opportunities. *ACM Trans. Reconfigurable Technol. Syst.* 15, 4, Article 51 (Aug. 2022), 42 pages. https://doi.org/10.1145/3530775

[5] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. 2011. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (2011), 473–491. https://doi.org/10.1109/TCAD.2011.2110592

[6] Paul Feautrier. 1988. Parametric integer programming. *RAIRO - Operations Research - Recherche Opérationnelle* 22, 3 (1988), 243–268. http://eudml.org/doc/104942

[7] Lorenzo Ferretti, Giovanni Ansaloni, and Laura Pozzi. 2018. Lattice-Traversing Design Space Exploration for High Level Synthesis. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*. 210–217. https://doi.org/10.1109/ICCD.2018.00040

[8] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. 2006. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Int. J. Parallel Program.* 34, 3 (jun 2006), 261–317. https://doi.org/10.1007/s10766-006-0012-3

[9] Licheng Guo, Yuze Chi, Jason Lau, Linghao Song, Xingyu Tian, Moazin Khatti, Weikang Qiao, Jie Wang, Ecenur Ustun, Zhenman Fang, Zhiru Zhang, and Jason Cong. 2023. TAPA: A Scalable Task-parallel Dataflow Programming Framework for Modern FPGAs with Co-optimization of HLS and Physical Design. *ACM Trans. Reconfigurable Technol. Syst.* 16, 4, Article 63 (dec 2023), 31 pages. https://doi.org/10.1145/3609335

[10] Gurobi Optimization, LLC. 2024. Gurobi Optimizer Reference Manual. https://www.gurobi.com

[11] Sitao Huang, Kun Wu, Hyunmin Jeong, Chengyue Wang, Deming Chen, and Wen-Mei Hwu. 2021. PyLog: An Algorithm-Centric Python-Based FPGA Programming and Synthesis Flow. *IEEE Trans. Comput.* 70, 12 (2021), 2015–2028. https://doi.org/10.1109/TC.2021.3123465

[12] Intel. 2024. Intel. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html

[13] 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* (Seaside, CA, USA) *(FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 242–251. https://doi.org/10.1145/3289602.3293910

[14] Jiajie Li, Yuze Chi, and Jason Cong. 2020. HeteroHalide: From Image Processing DSL to Efficient FPGA Acceleration. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '20)*. Association for Computing Machinery, New York, NY, USA, 51–57. https://doi.org/10.1145/3373087.3375320

[15] Peng Li, Louis-Noël Pouchet, and Jason Cong. 2014. Throughput optimization for high-level synthesis using resource constraints. In *Int. Workshop on Polyhedral Compilation Techniques (IMPACT'14)*.

[16] Rui Li, Yufan Xu, Aravind Sukumaran-Rajam, Atanas Rountev, and P. Sadayappan. 2021. Analytical characterization and design space exploration for optimization of CNNs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 928–942. https://doi.org/10.1145/3445814.3446759

[17] Junyi Liu, Samuel Bayliss, and George A. Constantinides. 2015. Offline Synthesis of Online Dependence Testing: Parametric Loop Pipelining for HLS. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. 159–162. https://doi.org/10.1109/FCCM.2015.31

[18] Junyi Liu, John Wickerson, Samuel Bayliss, and George A Constantinides. 2017. Polyhedral-based dynamic loop pipelining for high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 9 (2017), 1802–1815.

[19] Junyi Liu, John Wickerson, and George A Constantinides. 2016. Loop splitting for efficient pipelining in high-level synthesis. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 72–79.

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

[21] Microchip. 2023. SmartHLS Compiler Software. https://www.microchip.com/en-us/products/fpgas-and-plds/fpga-and-soc-design-tools/smarthls-compiler

[22] PoCC [n. d.]. *PoCC, the Polyhedral Compiler Collection 1.3*. http://pocc.sourceforge.net

[23] PolyBench [n. d.]. *PolyBench/C 4.2.1*. http://polybench.sourceforge.net

[24] Louis-Noël Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. 2013. Polyhedral-Based Data Reuse Optimization for Configurable Computing. In *21st ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'13)*. ACM Press, Monterey, California.

[25] Louis-Noel Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. 2013. Polyhedral-Based Data Reuse Optimization for Configurable Computing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '13)*. Association for Computing Machinery, New York, NY, USA, 29–38. https://doi.org/10.1145/2435264.2435273

[26] Stéphane Pouget, Louis-Noël Pouchet, and Jason Cong. 2024. Automatic Hardware Pragma Insertion in High-Level Synthesis: A Non-Linear Programming Approach. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey,CA,USA) *(FPGA '24)*. Association for Computing Machinery, New York, NY, USA, 184. https://doi.org/10.1145/3626202.3637593

[27] Stéphane Pouget, Louis-Noël Pouchet, and Jason Cong. 2025. Automatic Hardware Pragma Insertion in High-Level Synthesis: A Non-Linear Programming Approach. *ACM Trans. Des. Autom. Electron. Syst.* (2025).

[28] Siemens. 2023. Catapult High-Level Synthesis. https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/

[29] Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong. 2022. Automated Accelerator Optimization Aided by Graph Neural Networks. In *2022 59th ACM/IEEE Design Automation Conference (DAC)*.

[30] Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong. 2023. Robust GNN-Based Representation Learning for HLS. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. 1–9. https://doi.org/10.1109/ICCAD57390.2023.10323853

[31] 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 (feb 2022), 27 pages. https://doi.org/10.1145/3494534

[32] Sven Verdoolaege. 2011. Counting affine calculator and applications. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11), Chamonix, France*.

[33] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4, Article 54 (jan 2013), 23 pages. https://doi.org/10.1145/2400682.2400713

[34] AMD Xilinx. 2023.2. Vitis. https://www.xilinx.com/products/design-tools/vitis.html

[35] AMD Xilinx. 2024. Merlin. https://github.com/Xilinx/merlin-compiler

[36] Hanchen Ye, Hyegang Jun, and Deming Chen. 2024. HIDA: A Hierarchical Dataflow Compiler for High-Level Synthesis. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (La Jolla,CA,USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 215–230. https://doi.org/10.1145/3617232.3624850

[37] Hanchen Ye, HyeGang Jun, Hyunmin Jeong, Stephen Neuendorffer, and Deming Chen. 2022. ScaleHLS: A Scalable High-Level Synthesis Framework with Multi-Level Transformations and Optimizations: Invited. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 1355–1358. https://doi.org/10.1145/3489517.3530631

[38] 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 *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC.2018.8465827

[39] Weichuang Zhang, Jieru Zhao, Guan Shen, Quan Chen, Chen Chen, and Minyi Guo. 2024. An Optimizing Framework on MLIR for Efficient FPGA-based Accelerator Generation. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

[40] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. 2008. *AutoPilot: A Platform-Based ESL Synthesis System*. Springer Netherlands, Dordrecht, 99–112. https://doi.org/10.1007/978-1-4020-8588-8_6

[41] 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 *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 430–437. https://doi.org/10.1109/ICCAD.

2017.8203809
[42] Ruizhe Zhao and Jianyi Cheng. 2021. Phism: Polyhedral High-Level Synthesis in MLIR. *arXiv preprint arXiv:2103.15103* (2021).
[43] Ruizhe Zhao, Jianyi Cheng, Wayne Luk, and George A Constantinides. 2022. POLSCA: Polyhedral High-Level Synthesis with Compiler Transformations. *arXiv* (2022).
[44] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 863–879. https://www.usenix.org/conference/osdi20/presentation/zheng