Enhancing High-Level Synthesis with Automated Pragma Insertion and Code Transformation Framework

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

Louis-Noël Pouchet
pouchet@colostate.edu
Colorado State University

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

Abstract

High-level synthesis, source-to-source compilers, and various Design Space Exploration techniques for pragma insertion have significantly improved the Quality of Results of generated designs. These tools offer benefits such as reduced development time and enhanced performance. However, achieving high-quality results often requires additional manual code transformations and tiling selections, which are typically performed separately or as pre-processing steps. Although DSE techniques enable code transformation upfront, the vastness of the search space often limits the exploration of all possible code transformations, making it challenging to determine which transformations are necessary. Additionally, ensuring correctness remains challenging, especially for complex transformations and optimizations.

To tackle this obstacle, we first propose a comprehensive framework leveraging HLS compilers. Our system streamlines code transformation, pragma insertion, and tiles size selection for on-chip data caching through a unified optimization problem, aiming to enhance parallelization, particularly beneficial for computation-bound kernels. Them employing a novel Non-Linear Programming (NLP) approach, we simultaneously ascertain transformations, pragmas, and tile sizes, focusing on regular loop-based kernels. Our evaluation demonstrates that our framework adeptly identifies the appropriate transformations, including scenarios where no transformation is necessary, and inserts pragmas to achieve a favorable Quality of Results.

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

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, 6–9, 18, 26, 34, 35, 37–39]. 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) [5, 23, 24, 27, 28, 28, 29, 40] or source-to-source compilers [8, 37] to guide the synthesis process. These pragmas aid in optimizing the generated hardware code. Despite the improvements in productivity, both code transformation and ensuring the correctness of the transformations remains challenging. The HLS design space encompasses various code transformations and pragma placements. However, exploring this joint space can be daunting due to its vastness, with millions of potential points, and traditional analytical methods struggle to efficiently navigate it. Key issues include the lack of convexity and regularity in this space. Moreover, separating code transformation and pragma insertion into distinct optimization problems complicates the search and may necessitate initiating the search for one optimization problem upon any changes occurring in the other.

Our primary research objective is to develop a system capable of autonomously conducting search-friendly 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 problem. Even though several exploration methods could be used, we chose to employ a Non-Linear Programming (NLP) approach. This method facilitates rapid exploration of the entire theoretical space. We develop an analytical model that integrates considerations of latency and resource utilization, building upon previous research [23, 24]. Our focus lies specifically on regular loop-based kernels [20], 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 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 template that streamlines 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.
- We develop a novel NLP approach tailored to exploring this joint space, particularly focusing on regular loop-based kernels. The NLP allows for the exploration of a space whose resources are set by the user.
- Our framework acts as a comprehensive, fully automated system, offering end-to-end functionality. With it, we conduct thorough evaluations and attain QoR designs that are comparable or superior to those achieved by AutoDSE, NLP-DSE and HARP, *all without the necessity of Design Space Exploration*. Furthermore, Sisyphus accurately identifies the appropriate transformation, even in cases where no transformation is required.

The paper is organized into the following sections. Section 2 lays out the motivation behind our approach and the proposed solution. Following that, Section 3 presents the range of code transformation and pragma insertion we consider. In Section 4, we introduce a non-linear formulation based on this model to automatically discover schedule and pragma configurations through NLP optimization. Moving forward, Section 5 elaborates on our code generation process. Finally, sections 6, 7 and 8 are dedicated to evaluating our method, presenting related work, and drawing conclusions.

2 Background and Motivation

2.1 Space to explore

The exploration space for optimizing an HLS design encompasses pragma insertion, code transformation, and data caching, all while adhering to resource constraints. While various objective functions can be considered, our focus in this work is on minimizing latency by maximizing parallelism. This objective function is especially potent for computation-bound kernels, addressing the bottleneck caused by insufficient parallelization. These different elements of the space (pragma insertion, code transformation, and data caching) are inherently interconnected. Inserting unrolled pragmas influences array partitioning and thus the utilization of BRAM. The schedule and pragmas influence the possibility of caching data on-chip, and the choice of schedule impacts how we want to unroll based on loop properties (e.g., reduction loop), and so on. Thus, the choice of one element affects the others, necessitating backtracking if the spaces are separated. This complexity complicates the exploration of the space and also requires constant verification of the legality of transformations.

2.2 Limitation of the current DSE

Several frameworks [8, 37, 42] enable code transformations and pragma insertion, but their spaces are limited for both. Transformations are restricted to loop property-based permutations or the use of Pluto [1], which tiles the program but may restrict parallelization for FPGAs and does not consider FPGA-specific optimizations such as pipelining or array partitioning.

On the other hand, numerous Design Space Explorations (DSEs) [24, 27–29] explore pragma insertion for a fixed loop order. Although the user may initially set the schedule, it can be difficult, even for experts, to predict which transformations are necessary prior to pragma insertion. Furthermore, the multitude of code transformations renders the exploration of each pragma impractical in terms of scalability.

We will now examine examples that highlight the limitations of current methods. For this, we use two different HLS DSE methods, AutoDSE [29] and NLP-DSE [23, 24]. A comparison with HARP [28] will be presented in Section 6. AutoDSE trents the compiler Merlin [35] as a black box, adjusting pragmas based on 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/Xilinx 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.

In the upcoming examples, we utilize the HLS compiler Vitis 2023.2, excluding the unsafe-math option. We examine two scenarios: the kernel gemm from Polybench [21] as shown in Listing 1, and CNN.

```
for (i = 0; i < 200; i++) {
  for (j = 0; j < 220; j++)
    C[i][j] *= beta;
  for (k = 0; k < 240; k++)
    for (j = 0; j < 220; j++)
      C[i][j] += alpha * A[i][k] * B[k][j];}
```

Listing 1. Code original of gemm

The gemm kernel's original loop order, as illustrated in Listing 1, situates a non-reduction loop innermost for the second statement. This arrangement facilitates AutoDSE and NLP-DSE in unrolling this loop and pipelining the reduction loop. However, partially unrolling the reduction loop in this scenario diminishes performance due to the increased initiation interval (II) of the pipeline resulting from unrolling the reduction loop. As a consequence, the designs yielding the best QoR for these two methods achieve a throughput of approximately 20 GF/s. Listing 1 highlights the design found by AutoDSE. The pragmas with $factor = 1$ or pipeline off are not utilized in the final design. However, they illustrate the potential space explored by AutoDSE with various factors. In such instances, a transformation becomes imperative to augment parallelization.

```
#pragma ACCEL parallel factor=1
#pragma ACCEL pipeline off
#pragma ACCEL TILE FACTOR=5
for (i = 0; i < 200; i++) {
#pragma ACCEL tile factor=1
#pragma ACCEL pipeline off
#pragma ACCEL parallel FACTOR=4
  for (j = 0; j < 220; j++)
    C[i][j] *= beta;
#pragma ACCEL tile factor=1
#pragma ACCEL parallel factor=1
#pragma ACCEL pipeline flatten
  for (k = 0; k < 240; k++)
#pragma ACCEL pipeline off
#pragma ACCEL tile factor=1
    for (j = 0; j < 220; j++) // fully unrolled
      C[i][j] += alpha * A[i][k] * B[k][j];}
```

Listing 2. Code of gemm found by AutoDSE

When dealing with Convolutional Neural Networks (CNNs), there's significant room for code optimization, especially considering the need for tiling. Each loop split introduces various permutations, leading to a vast range of options, totaling 2.23×10^{13} potential combinations of loop orders and tile sizes. Despite achieving satisfactory QoR — 42.15 GF/s for AutoDSE and 31.80 GF/s for NLP-DSE — both methods fall short of fully exploiting the available level of parallelism, hinting at further optimization potential. However, exhaustively exploring each code transformation proves practically infeasible due to the sheer volume of possibilities. While the Merlin compiler can leverage strip mining to enhance on-chip data caching potential, its capabilities in this regard are constrained by the inability to permute the loops.

2.3 Overview of Sisyphus

To address these limitations, we propose Sisyphus, which focuses on linear code to offer a space capable of exploring only legal code transformations, pragma insertions, and tile size selections as a single optimization problem. Leveraging a Non-Linear Programming

(NLP) approach, Sisyphus can efficiently explore this space, benefiting from accurate modeling enabled by compile-time analysis feasible for linear code. The space comprises three levels of tiling, each corresponding to a specific optimization: fine-grained unrolling, pipelining, and coarse-grained unrolling, with loops executed sequentially and data transferred between off-chip and on-chip.

For gemm, Sisyphus excels in boosting parallelism by splitting and permuting loops, all without depending on the reduction loop. In this particular setup, the NLP discovers the configuration showcased in Listing 3.

```
// possibilities to cache on-chip A, B and C
/********* Level 0 *********/
for (int i0 = 0; i0 < 1; i0++)
// possibilities to cache on-chip C
  for (int j0 = 0; j0 < 1; j0++)
// possibilities to cache on-chip C
/********* Level 1 *********/
    for (int i1 = 0; i1 < 1; i1++)
     for (int j1 = 0; j1 < 55; j1++)
#pragma HLS pipeline
/********* Level 2 *********/
      for (int i2 = 0; i2 < 200; i2++)
#pragma HLS unroll
        for (int j2 = 0; j2 < 4; j2++)
#pragma HLS unroll
          C[i0*200+i2][j0*220+j1*4+j2] *=beta ;
/********* Level 0 *********/
for (int i0 = 0; i0 < 1; i0++)
// possibilities to cache on-chip A and C
  for (int j0 = 0; j0 < 1; j0++)
// possibilities to cache on-chip B and C
    for (int k0 = 0; k0 < 60; k0++)
// possibilities to cache on-chip A and B
/********* Level 1 *********/
      for (int i1 = 0; i1 < 1; i1++)
       for (int k1 = 0; k1 < 1; k1++)
        for (int j1 = 0; j1 < 220; j1++)
#pragma HLS pipeline
/********* Level 2 *********/
          for (int i2 = 0; i2 < 200; i2++)
#pragma HLS unroll
            for (int j2 = 0; j2 < 1; j2++)
#pragma HLS unroll
              for (int k2 = 0; k2 < 4; k2++)
#pragma HLS unroll
                i = i0*200+i2; j = j0*220+j1*1+j2;
                k = k0 * 4 + k2;
                C[i][j] +=alpha * A[i][k] * B[k][j];
```

Listing 3. Code of gemm found by Sisyphus

Both pipelined loops feature an II of 1, enabling a throughput of 210 GF/s. Partially unrolling the reduction loop allows for parallel execution of the two multiplications.

Concerning CNN, the NLP swiftly identifies a solution within 4.5 minutes. This solution optimizes tile configuration, caches a data tile of the array on-chip, partially unrolls one reduction loop and two non-reduction loops, and pipelines a non-reduction loop. As a result, the design achieves a throughput of 339 GF/s. The design discovered by Sisyphus is showcased in Listing 4.

```
for (int i0 = 0; i0 < 16; i0++) {
  load_weight(weight,vweight,i0);  // size 16*256*5*5
  load_output(output, voutput, i0); // size 16*224*224
  for (int j0 = 0; j0 < 256; j0++) {
    load_input(input, vinput, j0); // size 228*228
    for (int q0 = 0; q0 < 5; q0++)
      for (int w0 = 0; w0 < 14; w0++)
        for (int h1 = 0; h1 < 224; h1++)
#pragma HLS pipeline
          for (int i2 = 0; i2 < 16; i2++)  // unrolled
            for (int w2 = 0; w2 < 16; w2++) // unrolled
              for (int p2 = 0; p2 < 5; p2++){ // unrolled
                i = ...
                output[i2][h][w] +=weight[i2][j][p][q]
                                   * input[j2][h+p][w+q];}}
  store_output(voutput, output, i0);}
```

Listing 4. Code of CNN found by Sisyphus

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 provided in Listing 1.

Maximum Distribution To kickstart the process, we begin by thoroughly distributing the program, with the goal of maximizing the distance between each statement. This distribution strategy creates ample opportunities for parallelization by maximizing the separation between dependencies. Typically, this results in one statement per loop body with perfectly nested loops. To achieve this, we employ ISCC [31]. It allows us to explore various schedule distributions and validate the legality of transformations by ensuring dependency constraints are preserved. After distribution, the example (Listing 1) yields two loop bodies. The first statement iterates through loops i and j, while the second statement iterates through loops i, k, and j.

Strip-mining Following maximal distribution, we perform strip-mining on each loop twice. This process replaces the original loop, with a trip count TC_I0, with three new loops, each having trip counts TC_I0_0, TC_I0_1 and TC_I0_2 with $TC\_I0\_0 \times TC\_I0\_1 \times TC\_I0\_2 = TC\_I0$. 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. Following the implementation of three levels of strip-mining, the first loop body of the code from Listing 1 undergoes transformation into the code depicted in Listing 5.

```
for (i0 = 0; i0 < TC_I0_0; i0++)
 for (i1 = 0; i1 < TC_I0_1; i1++)
  for (i2 = 0; i2 < TC_I0_2; i2++)
   for (j0 = 0; j0 < TC_J0_0; j0++)
    for (j1 = 0; j1 < TC_J0_1; j1++)
     for (j2 = 0; j2 < TC_J0_2; j2++)
      i = i0 * TC_I0_1 * TC_I0_2 + i1 * TC_I0_2 + i2;
      ...
      C[i][j] *= beta;
```

Listing 5. Code of the first loop body of gemm with three level strip mining

Following the explanation provided in the next paragraph, strip mining allows us to create opportunities for fine-grained unrolling, pipelining, sequential execution, coarse-grained unrolling and tiling if loop permutation is legal. This expands the range of possibilities, allowing each loop to integrate diverse hardware directives and be reorganized across different levels of the code.

Loop Permutation 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 [31].

The innermost level facilitates fine-grained unrolling (complete unrolling), which increases parallelism by duplicating statements and can utilize tree reduction if a reduction loop is unrolled and the option is enabled. The middle level facilitates pipelining, enhancing throughput by overlapping loop iterations. Finally, the outermost level coordinates coarse-grained unrolling, tiling, and/or sequential execution, optimizing to increase the parallelism, and controlling the size of the on-chip buffer through tiling. If the loop body consists of only one statement and loop permutation is feasible, we disable coarse-grained unrolling. This will achieve equivalent results because our template permits unrolling the same loop innermost for fine-grained unrolling. Furthermore, coarse-grained unrolling entails duplicating each array, including read-only arrays, to ensure full overlapping of the modules, even when there are no dependencies between the concurrently executed modules.

In scenarios where permutations enable us to achieve perfectly nested loops capable of being pipelined, only one loop can usually be pipelined. Trying to pipeline all loops simultaneously, akin to pragma *loop_pipelined*, significantly complicates the design and extends synthesis times unreasonably. Thus, if a loop in the middle level is pipelined, it implies that the trip counts of the other loops (in the middle level) are reduced to one, ensuring manageable design complexity and synthesis times. As explained earlier, loop order selection is not necessary at the pipeline level. In the fine-grained level, arranging loops is straightforward; placing the reduction loop innermost enables efficient tree reduction. Conversely, at the coarse-grained or tiled level, prioritizing memory transfer latency is key. Therefore, we opt for a loop order that maintains the output stationary, minimizing costs associated with loading on-chip and storing off-chip during tile iterations, especially when the output is not fully transferred to the chip.

In the context of our example, these transformations yield the code presented in Listing 3, with each level corresponding to specific transformations. It is notable that our framework maintains the original loop order. In Listing 3, if the loop on line 12 ($i2$) iterates once and the loop on line 3 ($i0$) iterates 200 times, we keep the original loop order, then proceed to pipeline and partially unroll loop j.

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.

Space Therefore, the challenge involves determining the trip counts of each loop (e.g., TC_I0_0). Additionally, the design must adhere to resource constraints, including the usage of DSP and on-chip memory. Hence, the trip counts corresponding to fully unrolled loops must be constrained to avoid over-utilization of DSP resources. The on-chip buffers must not exceed the on-chip memory capacity. Consequently, for a large problem size, the outermost level will enable control over the buffer sizes. Thus, the space we consider encompasses the original schedule on which we can insert pragmas, but it also includes all the transformations mentioned previously for which the pragmas are inserted.

4 NLP

Presented here is a comprehensive set of constraints and variables employed in a non-linear program aimed at discovering the theoretical solution space outlined in Section 3. We employ the methodology proposed by [23, 24], adapted to our specific context. Similar to their approach, we establish an estimation of the latency by assuming optimistic DSP utilization, assuming perfect resource reuse. However, we afford the user the flexibility to adjust the DSP limit or opt for a pessimistic DSP utilization scenario, where no reuse between statements is considered. Moreover, users have the ability to adjust the size of on-chip memory, define the maximum number of array partitions, and customize the latency and resources allocated for each operation. This flexibility ensures adaptability across various platforms and compilers.

To gather all the required information for the NLP, we utilize PoCC [19] to extract the intermediate representation.

4.1 Variables

Tables 1 and 2 delineate the sets, variables, and constants utilized in our NLP formulation.

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 which iterate the statement s
$\mathcal{C}_{a_d}$	the set of loops which iterates the array a at the dimension d
$\mathcal{V}$	Level of the strip mining, 0 for coarse-grained/sequential, 1 for pipeline and 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
$DSP_{available}$	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

Table 1. Overview of the set and constant employed in formulating the NLP

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 of the strip-mining
$loop_l\_pip$	Boolean to know if the loop l is pipelined at level 1 of the strip-mining
$cache_l\_arr_a$	Boolean to know if the array a is transferred on-chip after the loop l at level 0 of the strip-mining

Table 2. Overview of the variable employed in formulating the NLP

4.2 Constraints

Now, we explore the precise meaning and implications of each constraint.

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

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

Coarse-grained unrolling We enforce coarse-grained unrolling solely for non-reduction loops (Eq. 4), where the unroll factor (UF) must divide and be less than or equal to the trip count of the current loop (Eq. 5 and 2). As elaborated in Section 3, coarse-grained unrolling is disregarded if there is only one statement within the loop (Eq. 3).

$$\forall l \in \mathcal{L}, loop_l\_UF \leq TC_{l,0} \tag{2}$$

$$\forall s \in \mathcal{S}_1, \forall l \in \mathcal{L}_s, loop_l\_UF = 1 \tag{3}$$

$$\forall s \in \mathcal{S}_1, \forall l \in \mathcal{L}_s^{red}, loop_l\_UF = 1 \tag{4}$$

$$\forall l \in \mathcal{L}, loop_l\_UF \% TC_{l,0} == 0 \tag{5}$$

Pipeline Constraints 6, 7, 8, and 9 facilitate the selection of loop pipelining, allowing only one pipeline per nested loop and computing the initiation interval (II) in accordance with the chosen pipelined loop. The II is determined based on the loop's properties and the iteration latency of the reduction operation, specifically when the loop being pipelined is a reduction loop, following the approach outlined in [23, 24].

$$\forall l \in \mathcal{L}, loop_l\_pip \in \{0, 1\} \tag{6}$$

$$\forall l \in \mathcal{L}, (1 - loop_l\_pip) \times TC_{l,1} == 1 \tag{7}$$

$$\forall s \in \mathcal{S}, \sum_{l \in \mathcal{L}_s} loop_l\_pip \leq 1 \tag{8}$$

$$\forall s \in \mathcal{S}, II_s = \sum_{l \in \mathcal{L}_s} loop_l\_pip \times II_l \tag{9}$$

On-chip memory Constraints 10 and 11 ensure that each array is cached on-chip at a single location within the code, and that the on-chip tiled data fits within the available on-chip memory.

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

$$\sum_{a \in \mathcal{A}} \sum_{l \in \mathcal{L}} cache_l\_arr_a \times ft\_array_a\_loop_l \leq Mem \tag{11}$$

Array partitioning Equations 12, 13 and 14 guarantee that the maximum array partitioning is not exceeded, and it ensures that the partitioning of the array is greater than or equal to the maximum unroll factor applied to that dimension. Thanks to Equation 13, which ensures that each unroll factor ($TC_{l,2}$) divides $AP_{a,d}$, $AP_{a,d}$ directly determines the array partitioning applied in the generated code.

$$\forall a \in \mathcal{A}, \prod_{d \in \mathbb{N}} AP_{a,d} \leq max_{part} \tag{12}$$

$$\forall a \in \mathcal{A}, \forall d \in \mathbb{N}, \forall l \in C_{a_d}, AP_{a,d} \% TC_{l,2} == 0 \tag{13}$$

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

DSP utilization Lastly, Equations 15, 16, 17, and 18 compute the DSP utilization under optimistic (full reuse) and pessimistic (no reuse) DSP reuse assumptions.

$$DSPs\_used_{opt} = \sum_{op \in \{+,-,*,/\}} \max_{s \in \mathcal{S}} (DSP_{s_{op}} / II_s) \tag{15}$$

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

$$reuse_{opt} \times DSPs\_used_{opt} \leq DSP_{available} \tag{17}$$

$$(1 - reuse_{opt}) \times DSPs\_used_{pes} \leq DSP_{available} \tag{18}$$

4.3 Objective Function

We structure the objective function akin to the approach described in [23, 24], tailoring it to meet the specific needs of our problem. Here, Lat_2 corresponds to the fine-grained unrolled tile, Lat_1 denotes the pipeline tile that incorporates the fine-grained unrolled tile, and recursively, Lat_0 encompasses Lat_1, enabling coarse-grained unrolling and facilitating on-chip data caching.

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

5 Code Generation and Optimization

The NLP allows us to find the trip count of each loop, the array partitioning for each array, and the size of the on-chip buffers. By considering the loop order and the pragmas applied to each loop as described in Section 3, we can directly generate the code. We automatically generate the functions that load data from off-chip to on-chip with the maximum burst size possible and vice versa with the functions that store the data off-chip. Additionally, we incorporate further optimizations to enhance the QoR.

5.1 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 [19]. Subsequently, we create a function for all statements iterated by this loop, ensuring compliance with the constraints of the non-constant trip count loop. This function returns the computation if the constraints are met or returns just the value of the output otherwise. Introducing a condition solely above the statement results in excessive compilation times. Therefore, we employ these techniques to reduce compilation time and achieve designs with a good QoR.

5.2 Overlapping computation and communication

The constraints of the NLP consider the sum of the footprints of all arrays used during the design execution. Thus, we have the option, while already anticipating resource constraints, to overlap

the transfer of arrays between off-chip and on-chip (or vice versa) at the same level or between the transfer of the arrays needed for the next loop body with the computation of the current loop body.

To achieve this, we separate each loop body into different functions, as well as the functions that transfer input data between off-chip and on-chip. Additionally, we create new functions that execute these independent functions in parallel. To ensure parallel execution, within these newly created functions designed for overlap, we include only independent functions.

5.3 Code transformation for HLS

To simplify the understanding of the HLS compiler, particularly to ensure that the compiler can pipeline loops with the correct initiation interval (II), we simplify the reductions (regardless of whether we use tree reductions) in the fine-grained unrolled part. As specified in section 3, the fully unrolled loops are placed innermost. Therefore, before the reduction, we create a variable to accumulate the reduction, which we then add to the output.

Additionally, we incorporate the pragma *loop_flatten off* on all reduction loops above the pipeline. While theoretically, this adjustment could enhance the QoR, in practice, the compiler might choose not to pipeline at all if it cannot flatten the loop and pipeline correctly.

6 Evaluation

6.1 Setup

We employ kernels sourced from Polybench/C 4.2.1 [20], supplemented by a Convolutional Neural Network (CNN) kernel and matrix multiplication tasks akin to those in the BERT transformer model. Our computations utilize single-precision floating-point data types as the default, facilitating comparison with AutoDSE and NLP-DSE. We operate on medium-sized datasets from PolyBench/C [20]. We also compare our approach with HARP [28], which seeks to enhance High-Level Synthesis design exploration by integrating a Graph Neural Network model that predicts HLS tool behavior. They conduct exploration within this space utilizing the GNN model for one hour and then synthesize 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 [23, 24], HARP's effectiveness is not universally applicable without the necessary fine-tuning or training, particularly when confronted with diverse problem sizes.

We chose these problem sizes to demonstrate the efficacy of our approach in transforming codes and inserting pragmas in scenarios where fully unrolling is not feasible. Additionally, we aim to illustrate the effectiveness of our framework in tiling code for substantial tasks such as CNN and bert_3072_100. This approach ensures that only a portion (a tile) of each array fits on-chip, aligning with the available memory constraints. The problem size and original loop order of CNN are I,J=256, H,W=224 and P,Q=5. For bert_n_m matrix multiplication the schedule and problem size are I=n, J=m, K=n (reduction).

We utilize the AMD/Xilinx Vitis HLS compiler [34] to showcase the effectiveness of Sisyphus. Report generation is performed using AMD/Xilinx Vitis 2023.2. When enabling tree reduction, we utilize the "funsafe math optimizations" option to enable commutative/associative reduction operators, thereby facilitating reductions to be implemented in logarithmic time. In our comparison with HARP [28], we utilized the same parameters as the authors, employing Vitis 2021.1 and activating the option for tree reduction. As our designated hardware platform, we deploy the Xilinx Alveo U200 device with a target frequency of 250 MHz. We conduct an analysis of the kernels and automatically generate each NLP problem using PoCC [19]. Solving these NLP problems, we employ the AMPL description language and utilize the commercial BARON solver version 21.1.13 [25, 30]. For our experimental setup, we harness the computing power of 2 AMD EPYC 7V13 64-Core Processor.

6.2 Experimental Evaluation

The aim of this evaluation is to demonstrate our capability to perform the proper transformation while simultaneously inserting the correct hardware directives. To achieve this, we compare our approach with three frameworks, that conduct design space exploration to insert pragmas with a fixed schedule.

The input codes provided are the original codes from Polybench or as described in the subsection 6.1. However, we modify the kernels containing loops with non-constant trip counts using the method outlined in Section 5.3. This adjustment ensures a fair evaluation with the transformations we apply.

We compare our method with AutoDSE [29] and NLP-DSE [23, 24] described in Section 2, and HARP [28].

We generate the AutoDSE space automatically by running the command *ds_generator*. We replace the unroll factor and tile size with all factors of the trip count to ensure consistency within the space. However, since maximum array partitioning is not a parameter within AutoDSE's space, this space, as observed later with the kernel atax, is slightly larger. AutoDSE is executed using the bottleneck method with a DSE timeout set to 1,000 minutes and a timeout of 180 minutes for each HLS synthesis. NLP-DSE is executed with the parameters specified in the paper, including a 30-minute timeout for the BARON solver and the space provided as input is $\{\infty, 2048, 1024, 512, 256, 128, 64, 32, 16, 8, 1\}$. For HARP, we adhere to the parameters outlined by the authors mentioned earlier.

In evaluating Sisyphus, we set a solver timeout of 14,400 seconds (4 hours) for the Baron solver, which generates only one design. By default, we assume that the compiler can efficiently reuse resources between statements that are not executed in parallel. If the synthesis results in the design exceeding DSP utilization, we relaunch the NLP with a pessimistic DSP utilization assumption—meaning we consider no reuse between statements—and regenerate the design accordingly. For heat-3d, symm, syrk, syr2k, and bert_3072_100 we had to apply the constraint indicating no reuse (Eq. 16), as the constraint with optimistic reuse (Eq. 15) resulted in kernels with resource over-utilization. The maximum partitioning constraint, max_{part} (cf. Section 4), is set at 1024, as determined by our use of AMD/Xilinx compilers and FPGAs, adhering to their specified limit.

Each synthesis process begins with a C-simulation to verify the correctness of the generated code.

6.3 Comparison of the methods

6.3.1 AutoDSE and NLP-DSE. Table 3 provides a comparison among AutoDSE, NLP-DSE, and Sisyphus. The **TR** column denotes the status of the tree reduction option, while the **T** column indicates the specific transformations applied, with D representing distribution, T for data tiling, and P for permutation. Across all evaluated kernels, we consistently achieve comparable or superior

performance, with the exception of the atax kernel, where AutoDSE yields designs approximately 1% to 2% faster, and heat-3d, where NLP-DSE makes a 1% improvement. However, the designs obtained by AutoDSE fall outside our defined space due to the stringent constraint of a maximum array partitioning limit of 1024 imposed by AMD/Xilinx Vitis. The design found by NLP-DSE for heat-3d shares the same schedule and pragma as ours, but slight enhancements in QoR are attributed to how Merlin handles memory transfer.

			GF/s	Perf. Improvement	
Kernel	**TR**	**T**	**Sisyphus**	**AutoDSE**	**NLP-DSE**
atax	1	D	1.96	0.99x	1.00x
atax	0	D	1.86	0.98x	1.00x
heat-3d	-	-	131.33	35.09x	0.99x
jacobi-2d	-	-	128.83	13.06x	6.60x
2mm	1	D,P	164.34	403.55x	1.33x
2mm	0	D,P	162.24	775.26x	1.36x
bert_100_64	1	D,P	88.30	1.08x	1.03x
bert_100_64	0	D,P	76.76	1.09x	1.09x
bert_100_768	1	D,P	218.08	1.04x	1.10x
bert_100_768	0	D,P	216.95	1.13x	1.13x
bert_100_3072	1	D,P	220.29	1.10x	1.10x
bert_100_3072	0	D,P	219.97	1.11x	1.11x
gemm	1	D,P	198.26	1.79x	1.51x
gemm	0	D,P	210.63	9.25x	9.57x
bicg	1	D,P	1.96	1.97x	1.97x
bicg	0	D,P	1.86	1.92x	1.93x
gemver	1	P	17.85	4.64x	1.77x
gemver	0	P	12.45	30.91x	7.24x
mvt	1	P	13.71	1.76x	1.76x
mvt	0	P	10.00	1.45x	1.45x
bert_3072_100	1	D,P,T	42.93	2.98x	3.31x
bert_3072_100	0	D,P,T	44.67	536.42x	626.60x
cnn	1	D,P,T	340.20	8.00x	8.91x
cnn	0	D,P,T	339.34	8.05x	10.67x
doitgen	1	D,P	53.22	1.33x	2.66x
doitgen	0	D,P	54.05	5.46x	2.71x
symm	1	D,P	236.82	8.15x	7.20x
symm	0	D,P	236.56	8.14x	7.19x
syr2k	1	D,P	428.12	9.28x	3.12x
syr2k	0	D,P	428.39	17.57x	3.13x
syrk	1	D,P	306.07	12.44x	4.40x
syrk	0	D,P	308.71	12.55x	4.43x
Average			153.65	59.99x	22.85x
Geo Mean			69.09	5.97x	2.89x

Table 3. Comparison of the Throughput (GF/s) achieved with Sisyphus, AutoDSE and NLP-DSE

Sisyphus averages a speedup of 59.99x and 22.85x over AutoDSE and NLP-DSE, respectively, across the evaluated kernels. In terms of geometric mean, Sisyphus achieves a speedup of 5.97x and 2.89x.

Conserve the original schedule when needed In the case of atax and jacobi-2d, Sisyphus preserves the original schedule and solely inserts pragmas. Thus, it effectively maintains the existing schedule while efficiently incorporating pragmas, a process comparable to that of two DSEs dedicated to pragma insertion. Regarding heat-3d, the original design opts for partial unrolling of the three loops with unroll factors of 2, 2, and 38, respectively, enabling a level of parallelism unattainable without code transformation. However, upon evaluation, we found that this design was over-utilizing resources. Therefore, the NLP solution, using a pessimistic constraint for DSP reuse, resulted in a design with the original schedule.

Although we note improvements for jacobi-2d, it is noteworthy that the same pragma insertion was utilized in at least one DSE. However, despite this consistency, AMD/Xilinx Vitis 2023.2 failed to synthesize the design generated by the source-to-source compiler Merlin, which the two DSEs employ.

Code transformation to manage reduction loop In the cases of 2mm, bert, and gemm, we observe notable improvements resulting from code transformations, particularly in effectively managing the reduction loop. This enhancement is particularly evident with bert_3072_100, where the reduction loop exhibits a large trip count, as well as in gemm when tree-reduction is not used.

Code transformation with tiling The selection of tile size becomes crucial for determining which array sizes, particularly for bert_3072_100 and CNN with their large arrays, should be cached on-chip and where they should be transferred. Organizing the outermost loop as output stationary (cf. Section 5.3), prompts the NLP to perform partial transfers of the array iterated by these loops in these examples. Consequently, the NLP's choices in these scenarios aim to optimize data reuse and minimize redundant memory transfers. However, for CNN, its large size requires multiple transfers of the array *input* (cf. Listing 4).

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. Additionally, in the case of gemver without tree reduction, the NLP reverses all loop orders to pipeline the non-reduction loop, further enhancing the QoR. 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, and syr2k, the original schedule imposes constraints on the achievable level of parallelism. This limitation elucidates the performance enhancements facilitated by Sisyphus. In the case of syrk and syr2k, the primary advantageous transformation is the maximal loop distribution. Meanwhile, for doitgen and symm, we observe QoR improvements because code transformations enable a combination of loop unrolling not attainable even with maximal loop distribution.

6.3.2 HARP. In Table 4 we compare the throughput and resource utilization (BRAM, DSP, LUT, FF) achieved with HARP [28]. 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 source-to-source compiler, which facilitates loop permutations when partially unrolling consecutive loops. This allows for them to find the same schedule that our framework identified. However, for Gemm, the design discovered by HARP falls outside our specified parameter space, particularly because of our restriction on maximal array partitioning. Nonetheless, it is worth noting that it is the only design among the top 10 evaluated that demonstrates a performance enhancement over our framework.

	GF/s		Resource Utilization (%) BRAM,DSP,LUT,FF		Perf. Imp
Kernel	**HARP**	**Sisyphus**	**HARP**	**Sisyphus**	
atax	1.72	1.77	78,52,49,50	38,26,25,31	1.03x
bicg	0.92	1.77	75,25,35,36	38,26,24,26	1.92x
gemm	125.59	99.96	29,80,55,40	1,67,51,34	0.80x
gemver	1.66	7.38	29,28,35,19	43,27,39,47	4.45x
mvt	7.07	7.06	40,78,43,30	21,64,33,26	1.00x
Average	27.39	23.58			1.84x
Geo Mean	4.71	6.95			1.47x

Table 4. Comparison of GF/s and resources utilization with HARP

6.4 Latency estimation

Implementing the post-optimization strategy outlined in Section 5.2 leads to a loss of the lower bound discussed in [23, 24], primarily because we do not incorporate computation-communication overlap modeling into the NLP. This omission is deliberate, as including such modeling would significantly increase search time.

Similarly, we opt against integrating the *loop_flatten* pragma into our model, despite its potential automatic invocation by the compiler for loops without reductions. As detailed in Section 5, we intentionally deactivate it for loops with reductions. Although this optimization's automatic application results in the loss of the lower bound, we have made this choice owing to its rare implementation. Our aim is to avoid reliance on an optimization that may not be consistently available.

In Figure 1, we depict the estimated latency by the NLP of the evaluated designs and juxtapose them with the latencies reported by HLS. Our post-optimization efforts reveal that most latencies estimated by NLP exceed those reported by HLS, with the exception of CNN with and without tree reduction. In the case of CNN (the last two dots), the pragmas are implemented as anticipated, but the lower bound of the computed unrolled part is not sufficiently tight.

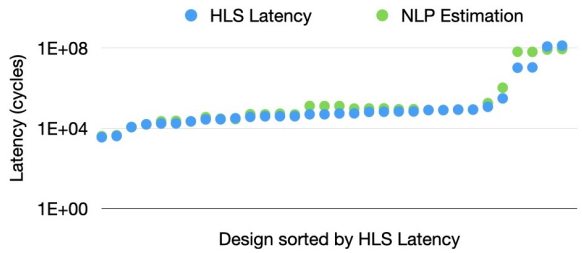

Figure 1. Comparison of the latency reported by HLS with the estimated latency obtained through NLP

6.5 Scalability of the NLP

Among the 42 NLP evaluations (including those with pessimistic resource reuse), only 3 timed out after 4 hours. These timeouts were specifically encountered during the bicg and atax kernels with tree reduction, and the gemver kernel without tree reduction. Throughout the search process, the NLP solver BARON maintains two bounds: a lower bound, which may be unattainable, and the lowest achievable bound discovered thus far. If the solver completes before reaching the timeout, these two bounds are identical. In the cases of atax and bicg, these two bounds were within 1% of each other, while for gemver, the difference was 20%. The average solving time was 33 minutes, while the geometric mean stood at 2.94 minutes. Out of 42 instances, 35 finished within an hour.

7 Related Work

Design Space Exploration (DSE) methodologies for pragma insertion, such as those mentioned in [5, 12, 17, 23, 24, 27, 28, 28, 29, 33, 38, 40, 43], yield designs with a satisfactory QoR. However, they often require extensive computational time, with AutoDSE taking up to a day to complete. While users have the option to perform code transformations before conducting DSE, the process typically involves treating each code transformation and pragma insertion as separate optimization problems. This decomposition of the problem can result in a loss of QoR because predicting which transformation is necessary to achieve an optimal design is challenging. The NLP-DSE method, as explained in [23, 24], employs a non-linear solver to integrate pragmas into the code. We build upon the methodology developed by the authors and tailor it to our specific optimization space to determine the pragmas and schedule for the kernel.

Various code transformations have been devised for CPUs [1], GPUs [32], and FPGAs [3, 10, 13–15, 22, 41, 42]. 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. Pluto [1], considered state-of-the-art, performs code transformations including tiling to minimize dependency distance between memory accesses, thereby facilitating data reuse. However, in our scenario, reducing dependency distance could potentially constrain parallelization efforts. Regarding [41, 42], 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, 10, 13–15, 22] pursue a distinct objective from ours. [22] aims to minimize communication between off-chip and on-chip, which results in better QoR than ours for memory-bound kernels. The works [3, 10, 13–15] 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.

The [2, 6, 8, 36, 37] compilers undertake code transformation and pragma insertion. However, these transformations are somewhat limited, mostly encompassing heuristic modifications based on loop properties. Moreover, the scope of pragma insertion is more limited compared to our proposed approach.

The choice of tile sizes significantly impacts the QoR. In line with our methodology, [11, 16] employ a cost model to determine the tile size. However, while [16] focuses on minimizing communication overhead, our approach differs. On the other hand, [11] investigates tiles size selection for Convolutional Neural Networks (CNNs) with three-level CPU caching.

8 Conclusion

This paper addresses challenges in optimizing HLS via code transformation and pragma insertion. While HLS and source-to-source compilers accelerate hardware design, achieving high-quality results often demands manual code intervention, prone to errors and time-consuming. To address this, Sisyphus is introduced, automating code transformation and pragma insertion for HLS into a single optimization problem. Using Non-Linear Programming (NLP), Sisyphus determines code transformations and pragma placements simultaneously, ensuring optimized performance within resource constraints.

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, 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

[5] 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

[6] 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

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

[8] 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

[9] 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

[10] 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)*.

[11] 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

[12] Hung-Yi Liu and Luca P. Carloni. 2013. On learning-based methods for design-space exploration with High-Level Synthesis. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–7.

[13] 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

[14] 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.

[15] 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.

[16] 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

[17] Anushree Mahapatra and Benjamin Carrion Schafer. 2014. Machine-learning based simulated annealer method for high level synthesis design space exploration. In *Proceedings of the 2014 Electronic System Level Synthesis Conference (ESLsyn)*. 1–6. https://doi.org/10.1109/ESLsyn.2014.6850383

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

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

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

[21] Polybench [n. d.]. PolyBench/C: the Polyhedral Benchmark suite. http://tinyurl.com/m7ztgex.

[22] 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

[23] 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

[24] Stéphane Pouget, Louis-Noël Pouchet, and Jason Cong. 2024. Automatic Hardware Pragma Insertion in High-Level Synthesis: A Non-Linear Programming Approach. *arXiv* (2024).

[25] N. V. Sahinidis. 2017. *BARON 21.1.13: Global Optimization of Mixed-Integer Nonlinear Programs*, User's Manual.

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

[27] 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)*.

[28] 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

[29] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. 2021. AutoDSE: Enabling Software Programmers Design Efficient FPGA Accelerators. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '21)*. Association for Computing Machinery, New York, NY, USA, 147. https://doi.org/10.1145/3431920.3439464

[30] M. Tawarmalani and N. V. Sahinidis. 2005. A polyhedral branch-and-cut approach to global optimization. *Mathematical Programming* 103 (2005), 225–249. Issue 2.

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

[32] 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

[33] Nan Wu, Yuan Xie, and Cong Hao. 2023. IronMan-Pro: Multiobjective Design Space Exploration in HLS via Reinforcement Learning and Graph Neural Network-Based Modeling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 3 (2023), 900–913. https://doi.org/10.1109/TCAD.2022.3185540

[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] 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

[40] 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

[41] Ruizhe Zhao and Jianyi Cheng. 2021. Phism: Polyhedral High-Level Synthesis in MLIR. *arXiv preprint arXiv:2103.15103* (2021).

[42] Ruizhe Zhao, Jianyi Cheng, Wayne Luk, and George A Constantinides. 2022. POLSCA: Polyhedral High-Level Synthesis with Compiler Transformations. *arXiv* (2022).

[43] 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 *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/2897937.2898040

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009