Automatic Hardware Pragma Insertion in High-Level Synthesis: A Non-Linear Programming Approach

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 enables the rapid prototyping of hardware accelerators, by combining a high-level description of the functional behavior of a kernel with a set of micro-architecture optimizations as inputs. Such pragmas may describe the pipelining and replication of units, or even higher level transformations for HLS such as automatic data caching using the AMD/Xilinx Merlin compiler. Selecting the best combination of pragmas, even within a restricted set, remains particularly challenging and the typical state-of-practice uses design-space exploration to navigate this space. But due to the highly irregular performance distribution of pragma configurations, typical DSE approaches are either extremely time consuming, or operating on a severely restricted search space.

In this work we propose a framework to automatically insert HLS pragmas in regular loop-based programs, supporting pipelining, unit replication (coarse- and fine-grain), and data caching. We develop a simple analytical performance and resource model as a function of the input program properties and pragmas inserted. We prove this model provides a lower bound on the actual performance for any possible configuration. We then encode this model as a Non-Linear Program, by making the pragma configuration as unknowns of this model, which is computed optimally by solving this NLP. This approach can also be used during DSE, to quickly prune points with a (possibly partial) pragma configuration, employing this latency lower bound property. We extensively evaluate our end-to-end fully implemented system, showing it can effectively manipulate spaces of billions of designs in seconds to minutes for the kernels evaluated.

KEYWORDS

HLS, FPGA, Non-Linear Programming, Program Optimization

1 INTRODUCTION

High-level synthesis (HLS) [5, 36] compilers [13, 18, 24, 32] and source-to-source compiler for HLS [12, 14, 15, 31, 33, 34] can reduce development time while delivering a good performance for the designs. However, achieving a satisfactory Quality of Results (QoR) often requires design-space exploration (DSE). This is because the design space, including which pragmas to insert and where, can not only contain millions of points, but typically does not present characteristics suitable for fast analytical exploration, such as convexity and regularity. Although the existing DSE methods [25, 26, 30] can find designs with a good QoR, it comes at a high computation cost:

- For example hundreds, of designs may be concretely instantiated using HLS to compute its estimated QoR during exploration [26].

Our main objective is to provide a system to automatically insert a set of hardware pragmas for HLS that delivers a good QoR but significantly reduces the search time needed to obtain the final design.
is a lower bound on the final QoR. In Section 5, we introduce a non-linear formulation based on this model to automatically find pragma configurations by NLP optimization. Section 6 presents our lightweight DSE approach. Finally, Sections 7 and 9 are devoted to evaluating our method validating the effectiveness of our approach and presenting related work, before concluding.

2 BACKGROUND AND MOTIVATION

The main objective of this work is to automatically optimize FPGA designs using HLS [5, 36]. HLS has made FPGA usage more accessible, and many projects are looking to further democratize this field by automating optimizations [10, 12, 14, 26, 33] such as the AMD/Xilinx Merlin Compiler ([3, 4, 31]).

In particular, Merlin was developed to improve the performance and reduce the development time of HLS-based designs. To achieve this goal, it automatically generates data transfers between off-chip and on-chip memory and inserts important Vitis pragmas, such as for array partitioning. Merlin includes hardware directives pragmas such as ACCEL parallel <factor=x>, which creates x parallel instances, and Merlin restructures the code accordingly if the loop nest has more than x iterations. This pragma can be used for fine-grained and coarse-grained parallelization. There are also the pragmas ACCEL pipeline <II=y> for pipelining, and ACCEL tile <factor=z> for strip-mining a loop by z, enabling Merlin to insert other pragmas such as data caching in a loop with smaller trip count, matching the on-chip resources available and reducing off-chip communications. Merlin’s ACCEL cache <array=a> directive transfers all required elements of array a from off-chip to on-chip to perform computations within the specified sub-region.

2.1 DSE for Pragma Insertion

Tools such as AutoDSE [26] and GNN-DSE [25] rely on Merlin and perform DSE to find which pragmas to insert and where.

Although Merlin further raises the level of abstraction and ease to implement complex transformations such as data caching and various types of hardware replication and hence reduces the space of configurations, the space of pragmas and their parameter values is still large: as shown later in Sec. 7, this space quickly reaches billions of feasible designs even for kernels containing only a handful of loops and statements.

Furthermore, to avoid accuracy limitations of analytical performance models, DSE approaches exploring such space typically rely on partially generating the design, at least via HLS, and utilizing the QoR report as the performance estimate for a design. HLS time for highly optimized designs combining various Merlin pragmas (e.g., parallelism and caching) can quickly reach several hours per design, making the search process particularly time-consuming. One may restrict the space of pragmas considered, and especially their parameter range, to reduce search time [26]. However this comes at the expense of the potentially improved performance that can be reached by considering the complete parameter space.

Yet such DSE remains the state-of-practice approach to deliver high-quality designs, we illustrate its performance merits with three sample benchmarks from linear algebra: GEMM, the dense generalized matrix-multiply, 2mm, which computes the matrix product \( D = A \times B + C \), and Gramschmidt, which computes the Gram-Schmidt process for QR decomposition. Table 1 below displays the performance (in GigaFlop/s, GF/s) of the original program from PolyBench/C using the Medium dataset size and Large for Gramschmidt, without any pragmas, and optimized by Merlin and the final performance achieved by running AutoDSE to insert Merlin pragmas, given a time budget of 20 hours and a timeout of 3 hours per HLS run. DSE time is reported in minutes.

<table>
<thead>
<tr>
<th></th>
<th>2mm</th>
<th>Gemm</th>
<th>Gramsch.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GF/s</td>
<td>Time</td>
<td>GF/s</td>
</tr>
<tr>
<td>Original</td>
<td>0.10</td>
<td>N/A</td>
<td>0.07</td>
</tr>
<tr>
<td>AutoDSE</td>
<td>0.41</td>
<td>1,870</td>
<td>68.91</td>
</tr>
<tr>
<td>NLP-DSE</td>
<td>117.48</td>
<td>70</td>
<td>105.18</td>
</tr>
</tbody>
</table>

Table 1: Result of AutoDSE

This framework utilizes the HLS compilers as a black box used to select configurations that minimize the objective function. The tool is agnostic of the input program shape and if it detects that Merlin did not apply the pragmas as expected it allows the DSE to prune the design after Merlin has generated the HLS-C code.

By analyzing the space explored by the DSE for these three examples, valuable hints can be observed to improve the method. For 2mm, the fastest design found by AutoDSE is mainly optimized for a single loop body. When AutoDSE tries to optimize the second loop body, it favors the unroll factors to the power of two for the innermost loop and goes directly to the outermost loop. This does not improve performance or create configurations which are pruned. The fact that it does not try the other unroll factors for the innermost loop before optimizing the other pragmas creates a loss of performance. For Gemm Medium and Gramschmidt Large, the DSE makes it possible to find designs with a good QoR. However the DSE wastes much time exploring too large unroll factors, which generates ponderously long synthesis times without giving any result as the HLS timeout is reached. The time spent increasing the unroll factor for certain pragmas without result does not allow the unroll factor of other pragmas to be increased, which results in missed performance.

2.2 Overview of NLP-DSE

Our primary objective is to efficiently achieve high-quality designs within a limited compilation time budget. To accomplish this we develop a DSE based on a cost model which can find the theoretical optimal configuration in a few seconds/minutes, thanks to its implementation as a Non-Linear Programming (NLP) problem which is then solved using BARON [27]. Note, however, the in-practice configuration that achieves the best performance may be different: our NLP is unable to model the complexity of the downstream toolchain and issues of resource usage/sharing, actual implementation of the pragmas by Merlin and Vitis, and simple performance bugs in these optimizing compilers may imply a theoretically best design is not implemented as such, making a design space exploration approach still required to achieve a good performance. We have therefore set up a DSE which allows the framework to explore different spaces of parallelism as well as different configuration spaces by constraining the level of parallelism. Our model is presented in Sec. 3, and we
prove it is a performance lower bound in Sec. 4, an important feature to be able to prune designs during the search without the risk of losing performance. The associated NLP formulation is given in Sec. 5.

To make our approach feasible, and in particular maintain a sufficient level of accuracy in the analytical models, we target regular, loop-based computations, that is affine programs [7]. Such programs have a static control-flow, that is every operation executed, and a branch taken at run-time, can be exactly modeled at the compile-time using polyhedral structures [9]. This restriction improves accuracy of loop-based analytical model by providing them with exact information about the loop trip counts and dependencies.

It is important to note that our DSE process, presented in Sec. 6, fundamentally differs from AutoDSE. We first evaluate configurations that have the lowest theoretical latency, and hence typically a high level of parallelism. However AutoDSE starts from a configuration with no pragmas, which is typically feasible in terms of resource usage, and progressively increases the pragmas until resources are exceeded. Instead our NLP-DSE begins with a design featuring the maximum achievable parallelism, and then gradually reduces the level of parallelism. We illustrate this with Table 1 with the final performance and time-to-solution achieved by NLP-DSE.

As presented in Sec. 7, compared to AutoDSE, our framework locates designs with similar or higher throughput for 46 out of 47 benchmarks evaluated, with a performance improvement of 5.69x and 17.24x on average in terms of the DSE time and design throughput respectively. Only 1 design having a 7% lower throughput. In addition, Vitis will auto-pipeline with a target II of 1 the innermost loop which are not fully unrolled for each nested loop. This vector is built by syntactic analysis on the program, where the factor and the unrolled pragma applied to that loop. Similarly to Vitis, Merlin will auto-pipeline. Merlin also applies a program flatten, some other optimizations when the user gives a compilation option such as tree reduction.

Only a loop with a constant TC, i.e., $T_{C_{max}} = T_{C_{min}}$ can be unrolled. The unroll pragma options allow to specify the unrolling factor, $uf$. When the factor is not specified, it implies that the factor is equal to the TC of the loop. When a loop is pipelined, all innermost loops are automatically fully unrolled. Hence, we also propagate unrolling information, e.g., to mark a full loop nest for full unrolling if an outer loop is marked with #pragma ACCEL parallel pragma) and set to $T_{C_{max}}$ if parallel is defined without a factor $uf$ specified. tile is the TC of the innermost loop after strip mining. $T_{C_{min}}$ is the minimal trip count of loop i, for any of its execution in the program. We also compute the maximal trip count over all executions. These values are computed using polyhedral analysis on the loops [21]. The pragma cache transfers above the loop i the data needed for the computation of this loop nest for the array a.

This vector is built by syntactic analysis on the program, where the default value $PV_i$ is used for a loop without any pragma. Once all loops have been annotated by their $PV$ properties, subsequent treatment can be implemented to mirror the optimizations implemented by the back-end tool.

**Modeling Vitis optimizations**

Finally, Merlin will also add automatic optimizations. It will explicitly strip-mine a loop when it is partially unrolled with the innermost loop having a TC equal to the factor and the unrolled pragma applied to that loop. Similarly to Vitis, Merlin will auto-pipeline. Merlin also applies a program transformation for certain pragmas. When there are two perfectly
nested and partially unrolled loops, Merlin swaps the two loops strip-mined (if legal), unrolled innermost and flatten and pipeline the two outermost loops. Further, Merlin will automatically transfer the data from off-chip to on-chip and cache on-chip with packing, with a maximum packing of 512 bits for our FPGA while computing if the footprint of the data fit on-chip by static analysis. The pragma tile allows Merlin to strip mine a loop and give the compiler the opportunity to transfer less more data while respecting resource constraints. For our model we suppose an optimistic data transfer i.e., all memory transfers are done with a packing of 512 bits and each data are transferred once (perfect data reuse).

Consequently, the set of possible \( P_l \) vectors are adjusted by analyzing the input code, and modifying their initial value, possibly further constraining the set of possible \( P_l \) based on which program transformation will be performed, as described above. Overall, the \( P_l \) vectors, along with the summarized AST, contain sufficient information to capture several source-to-source transformations performed by the Merlin compiler for coarse- and fine-grain parallelization, and reason on the likeliness of the optimization to succeed at HLS time (e.g., capturing loops with non-constant trip count).

4 THEORETICAL LATENCY AND RESOURCE MODELING

We now outline key elements of our analytical performance model, and the associated proofs this model computes a lower bound on latency under resource constraints. Full proofs cover several pages, and are available in the Appendix. We limit below to summarizing the hypothesis and overall proof approach.

4.1 Analytical Model Template

We take as input the summary AST of the program in constructor form, and the set of \( P_l \) vectors for each loop, as defined in Sec. 3.1. Returning to our example in Sec. 3.1, this summary AST, capturing the loop structure and all statements, is \( \text{Loop}(\text{Loop}_j(S_1), \text{Loop}_j(S_2, S_3)) \).

Let \( C_i^{P_l}(X_1, ..., X_n) \) be an operator that computes the latency of a region of a code represented by \( X \), where \( X \) is composed of loops and/or statements. This function is computed as follows: \( C_i^{P_l}(X) = \left[ H_i * (T_{C}\text{min} / u_f - p_i) \right] \times X \) where \( \circ \) is + if the loop \( i \) is pipelined \( (p_i = 1) \), \( \times \) otherwise. Let \( C_i^{P_l}(X_1, ..., X_n) \) the operators which composed the region, defined as follows: \( C_i^{P_l}(X_1, ..., X_n) = \max_{i \in 1}^{n} C_i^{P_l}(X_i) \) if there is no dependency (WaR, RaW, WaW) between the statements of the regions and \( \sum_{i=1}^{n} X_i \) between the statements which have at least one dependency. Finally, \( S_l^{P_l}(X_1) \) denotes a region of straight-line code (e.g., an inner-loop body). Intuitively, \( S_l \) will represent a lower bound on the latency of a code block such that, by the composition across all loops as per the template formula, the result remains a lower bound on the full program latency.

We build the analytical formula template by simple substitution of the operators in the summary AST by the following operators:

1. For every \( \text{Loop}_i(X) \), replace it by \( C_i^{P_l}(X) \);
2. For every list \( \bar{I} \) containing the \( \bar{I} \) of the step 1, replace it by \( C_i^{P_l}(\bar{I}) \); and
3. replace statement lists inside loop 1 \( S_k \) by \( S_l^{P_l}(S_k) \). That is for our example, we simply rewrite: \( \text{Loop}(\text{Loop}_j(S_1), \text{Loop}_j(S_2, S_3)) \) into the following formula:

\[
\bar{I}_{l}^{P_l}(C_l^{P_l}(I_1^{P_l}(S_1^{P_l}(S_1)), I_2^{P_l}(S_2^{P_l}(S_2, S_3))))
\]

4.2 Restrictions and Applicability

Our objective is to formulate a lower bound on the latency of a program, after HLS. We therefore include several restrictions:

- The input program is a pure polyhedral program [11], and its analysis (loop trip counts for every loop, all data dependencies [8]) is exact.
- No HLS optimization shall change the number of operations in the computation: strength reduction, common sub-expression elimination, etc., shall either first be performed in the input program before analysis, or not be performed by the HLS toolchain.
- The program does not contain "useless" operations. A typical case is when a value is overwritten before being utilized, e.g., \( x = 42 + 51; x = 0; \)
- We only model DSP and BRAM resources for the considered kernel, ignoring all other resources.
- We assume resources (DSP) sharing across different operations executing at the same cycle is not possible.

4.3 Latency Lower Bound

An important term is \( S_L \), a latency lower bound for a region of straight-line code. To maintain a lower bound on latency by composition, we operate on a representation of (parts of) the program which is both schedule-independent and storage-independent: the operation graph, or CDAG [6]. Indeed, a lower bound on this representation is necessarily valid under any schedule and storage eventually implemented and can be used to prove I/O lower bounds on programs [6] which is the directed acyclic graph with one node per operation in the code region, connecting all producer and consumer operations to build the operation graph. Then, we can easily compute the length of its critical path, which represents the minimal set of operations to execute serially.

Then, we can build a lower bound on the number of cycles a region \( L \) may require to execute, under fixed resources, by simply taking the maximum between the weighted critical path and the work to execute normalized by the resources available.

**Theorem 4.1 (Latency Lower Bound under Operation Resource Constraints).** Given \( O \) the set of operation type and \( R_a \) a count of available resources of type \( o \in O \), \( O \) the operation graph, \( c_p \) the critical path of \( O \), and \( L(a) \) the latency function for operation \( a \), with \( L(a) \geq 1 \). \( N_o^L \) denotes the number of operations of type \( o \) in \( L \). We define \( L_{a} = \sum_{o \in O} L(o) \) the critical path of \( L \) weighted by latency of its operations. The minimal latency of a region \( L \) is bounded by

\[
\text{Lat}_{L_{a}} \geq \max_{o \in O} \left( N_o^L \times L(a)/R_a \right)
\]

This theorem provides the building block to our analysis: if reasoning on a straight-line code region, without any loop, then building the operation graph for this region and reasoning on its critical path is sufficient to provide a latency lower bound.
}

L3: for \((j = 0; j < N; j++)\)
1
L1: for \((i = 0; i < N; i++)\)
S0: \( s[i] = 0 \);

Automatic Hardware Pragma Insertion in High-Level Synthesis: A Non-Linear Programming Approach

max
to the completion of the preceding one. The initiation interval (II)

ple iterations of the loop, so that the next iteration can start prior

⌊

as

𝐿𝑎𝑡

the trip count of

𝐿𝑈𝐹

been unrolled by a factor denoted as

𝐿𝑎𝑡

critical paths, expressed as

max

(∗)

Consider the scenario of loop L0 within Listing 1, which has
been unrolled by a factor denoted as \( UF \leq TC_{L0} \) where \( TC_{L0} \)

replication of the loop body will be instantiated. If \( TC_i \) \( mod \) \( UF \) \# 0 then an epilogue code to execute

Unrolling can be viewed as a two-step transformation: first strip-
mine the loop by the unroll factor, then consider the inner loop
obtained to be fully-unrolled. The latency of the resulting sub-
program is determined by how the outer-loop generated will be
implemented. We assume without additional explicit information
this unrolled loop will execute in a non-pipelined, non-parallel
fashion. Note this bound requires to build the operation graph for
the whole loop body. This is straightforward for inner loops and/or
fully unrolled loop nests, but impractical if the loop body contains
other loops. We therefore define a weaker, but more practical, bound
that enables composition:

THEOREM 4.2 (MINIMAL LATENCY OF A PARTIALLY UNROLLED LOOP
WITH FACTOR UF AND COMPLEX LOOP BODIES). Given a loop L with
trip count \( TC_i \) and loop body L, unroll factor \( UF \leq TC \) and \( O \) the set of
operation type. Given available resources \( R_o \) and latencies \( L(o) \geq 1 \)
with \( o \in O \). Then the minimal latency of L if executed in a non-
pipelined fashion is bounded by:

\[
Lat^{LS}_{R_o} \geq \left[ \frac{TC_i}{UF} \right] \cdot Lat^{L0}_{R_o}
\]

The intuition of the proof of Theorem 4.2 is that there cannot
exist a shorter critical path in the operation graph of the unrolled
loop nest than in the non-unrolled loop body, that is there is no
useless operation. This is therefore essential to require the absence
of useless operations in the program, as otherwise a shorter critical
path may be exposed after loop unrolling, making this theorem
incorrect.

Consider the scenario of loop L0 within Listing 1, which has
been unrolled by a factor denoted as \( UF \leq TC_{L0} \) where \( TC_{L0} \) is
the trip count of L0. The latency for one iteration of S0 is denoted
as \( Lat^{S0}_{R_o} > 0 \). In the absence of pipelining, the lower bound of
the latency of this sub-loop body is:

\[
Lat^{L0}_{R_o} = \left[ \frac{TC_{L0}}{UF} \right] \cdot Lat^{S0}_{R_o}.
\]

Loop pipelining Loop pipelining amounts to overlapping multiple
iterations of the loop, so that the next iteration can start prior
to the completion of the preceding one. The initiation interval (II)
measures in cycles the delay between the start of two consecutive
iterations. It is easy to prove our formula template accurately
integrates the latency of pipelined loops with the I operator. We
compute the minimal II in function of the dependencies of the
operations of the operations during the statements of the NLP
generation. Let \( Rec_{MII} \) and \( Res_{MII} \) be the recurrence constraints and
the resource constraints of the pipelined loop, respectively. We have \( II \geq \max(Rec_{MII}, Res_{MII}) \).

Rec_{MII} = \max \left( \frac{delay(c_i)}{distance(c_i)} \right)

\( delay(c_i) \) the total latency in dependency cycle \( c_i \) and
distance(c_i) the total distance in dependency cycle \( c_i \). We suppose that \( Res_{MII} = 1 \), as we don’t know
how the resource will be used by the compiler. Hence, if the loop
is a reduction loop then the \( II \geq \frac{H_{reduction}}{1} \) with \( H_{reduction} \) the
iteration latency of the operation of reduction. For a kernel like
\( for(j = 0; j < N; j++)y[j] = y[j-2] + 3; \) the \( II \geq \frac{H_{q}}{1} \).

Program latency lower bound

We now focus on having the program latency lower bound, with
resource constraints. This bound takes into account the limitations
imposed by available resources, which can significantly affect the
achievable performance. We assume here that the resources con-
sumed are only consumed by the computing units and resource
use by the computational unit of one operation can not be reused
by the computational unit of another operation executing at the
same time. We also assume that the compilers have implemented
the pragma configuration given as input.

For DSPs we suppose we have a perfect reuse, i.e., that the com-
putation units for the same operation can be reused as soon as the
computation unit is not in use. Under-estimating the resources used
is fundamental to proving the latency lower bound, as otherwise
another design that consumes less resources than predicted may be
feasible, itself possibly leading to a better latency. The lower bound
of a kernel is therefore the lower bound of the configuration such
that the minimal resource consumption is less than or equal to the
resource available.

THEOREM 4.3 (LOWER BOUND OF THE KERNEL UNDER RESOURCE
CONSTRAINT). Given a loop body L, the set of statements \( S \), O the
set of operation type, \( N^o_s \) the number of operations \( o \in O \) for the
statements \( s \), DSP \( o \) the number of resources (DSPs) used for the
operation \( o \in O \), MCU \( o \) the maximal computational unit the
statement \( s \) can use in parallel and the configuration of pragma which
is constituted of the vectors \( PV_i \) for each loop. The configuration
of pragma, which has the minimal lower bound and which respect
\( \sum_{o \in S} N^o_s \times DSP_o \times MCU^o_o \leq R_o \), is the lower bound
of the kernel.

The intuition of the proof of the Theorem 4.3 is that \( p_{min}^{used} \)
is the minimal resource that the kernel can use for the configuration
of pragma. This implies that if \( p_{min}^{used} > R_o \) the configuration
over-utilizes resources. Hence, the lowest lower bound which does
not over-utilize the resource is the lower bound of the kernel.

5 NON-LINEAR FORMULATION FOR PRAGMA
INSERTION

We now present the complete set of constraints and variables em-
ployed to encode the latency and resource model as a non-linear
program. This section presents a modeling of section 4 in the prac-
tical case.
Let $\mathcal{L}$ be the set of loops, $\mathcal{A}$ the set of arrays, $\mathcal{S}$ the set of statements and $\mathcal{O}_t$ the operations of the statements $s$. In order to have an accurate model we distinguish for each statement the operation which can be done in parallel, i.e., does not have any loop-carried dependence, $\mathcal{O}_{\text{sp}}$, and the reduction operations, i.e., associative/commutative operators to reduce one or more values into a single value, leading to loop-carried dependencies, $\mathcal{O}_{\text{red}}$.

Let $\mathcal{P}$ be the set of different possible pipeline configurations. Let $\forall p \in \mathcal{P}$ define $L^p$ the set of loops loop piped and $\forall l \in L^p$, $L^p_{\text{under}}$ the set of loops under a loop piped and $L^p_{\text{above}}$ the set of loops above the loop piped. Let $\forall s \in \mathcal{S}$ define the set of nested loops which iterate the statement $s$, $L^p_s$. $\forall a \in \mathcal{A}$ and for $d$ a dimension of the array $a$, let $\text{C}_{un}$ be the set of loops which iterate the array $a$ at the dimension $d$. $\forall l \in \mathcal{L}$, let designate $d_l$ the maximum dependency distance of the loop $l$. And let $H_k$ be the II of the loop piped for the statement $s$.

The II for each loop, the dependencies, the properties of the loops, the TC, the iteration latency of the parallel operations and the reduction operations and the number of DSPs per operation per statements are computed at compile time with PolyOpt-HLS [21] and used as constants in the NLP problem.

### 5.1 Variables

Variables in the formulation correspond to $PV_l$ attributes. We consider the possibilities of pipelining (Eq. 3), unrolling (Eq. 1), and tiling (Eq. 2) for each loop. Additionally, we include the possibility of caching an array that is iterated over by the loop (Eq. 4).

### 5.2 Modeling Compiler Transformations

Now that we have defined our design space, we need to constrain the space by removing infeasible cases and those that do not comply with the rules of the compilers.

**Pipeline Rules** Vitis HLS unrolls all loops under the pipelined loop. This implies that all loop $l$ under the pipelined loop must have $\text{loop}_l^{\text{UF}} = \text{TC}_l$ (Eq. 15). Considering this constraint, it is important to note that only one pipelined loop can exist per loop (Eq. 5). If multiple pipelined loops were present, the loops beneath the first pipelined loop would be unrolled instead.

**Memory Transfer Rules** Merlin automates the process of transferring data on-chip and applying array partitioning. The tool caches data on-chip and packs it in chunks of up to 512 bits, enabling efficient transfer speed. When the data is already present on-chip it can be reused provided that resource constraints are satisfied. The compiler caches on-chip the data only above the loop pipelined (Eq. 14).

**Dependencies** Loop-carried dependencies are managed using constraints (Eq 8). If a loop has a dependency distance of $n$, this means that if we unroll with an unroll factor $uf > n$ this is equivalent to unrolling the loop with a factor $uf = n$ because the statements corresponding to the iteration $[uf + 1, \ldots, uf + n]$ will be executed only after the first $n$ statements are executed due to the dependency. Loop-independent data dependencies are managed at the objective function, as elaborated in Section 5.4.

**Supplementary Rules** In addition, we add the constraints for the maximum unrolling (Eq. 10), the divisibility of the problem size of the unroll factors (Eq. 6), the tile size (Eq. 7) and the maximum array partitioning (Eq. 13). For this last, this corresponds to adding an upper bound to the product of the UF of all the loops which iterate the same array on different dimensions.

During our DSE, we can force the solution to be fine-grained. In this case we add a constraint where the loop above the loop pipelined has a UF of 1, i.e., for all loop $l$ above the pipelined loop $\text{loop}_l^{\text{UF}} = 1$ (Eq. 9). We also constrain the resources, modeling their sharing optimistically. We consider the number of DSPs (Eq. 11) and on-chip memory (Eq. 12) used. As the consumption of DSPs can be difficult to estimate due to resource sharing we utilize an optimistic estimate, which considers a perfect reuse/sharing: as soon as a computation unit is free, its resource can be reused.

### 5.3 Constraints

\begin{align}
\forall l \in \mathcal{L}, 1 \leq \text{loop}_l^{\text{UF}} &\leq \text{TC}_l \quad (1) \\
\forall l \in \mathcal{L}, 1 \leq \text{loop}_l^{\text{tile}} &\leq \text{TC}_l \quad (2) \\
\forall l \in \mathcal{L}, \text{loop}_l^{\text{pip}} \in \{0, 1\} \quad (3) \\
\forall l \in \mathcal{L}, \forall a \in \mathcal{A}, \text{loop}_a^{\text{UF}} \text{cache array}_a \in \{0, 1\} \quad (4) \\
\forall l \in \mathcal{L}, \sum_{s \in \mathcal{S}} \text{loop}_l^{\text{pip}} \leq 1 \quad (5) \\
\forall l \in \mathcal{L}, \text{loop}_l^{\text{UF}} \% \text{TC}_l \quad (6) \\
\forall l \in \mathcal{L}, \text{loop}_l^{\text{tile}} \% \text{TC}_l \quad (7) \\
\forall l \in \mathcal{L}, \forall l' \in \mathcal{L}_{\text{above}}, \text{loop}_l^{\text{UF}} \times \text{loop}_{l'}^{\text{UF}} \leq 1 \quad (8) \\
\forall l \in \mathcal{S}, \prod_{l \in \mathcal{L}} \text{loop}_l^{\text{UF}} \leq \text{MAX\_PARTITIONING} \quad (9) \\
D\text{SPs used}_{\text{opt}} &\leq \sum_{o \in \{+, -, \times, /\}} \max(D\text{SPs used}_o, H_k) \quad (11) \\
\forall a \in \mathcal{A}, \forall l, l' \in \mathcal{L}^{d, d'} \in \mathcal{H}^2 \text{ with } d \neq d', \forall a \in \mathcal{A}, \forall l, l' \in \mathcal{L}^{d, d'} \quad (12) \\
\forall a \in \mathcal{A}, \forall l, l' \in \mathcal{L}^{d, d'} \text{ with } d \neq d', \forall a \in \mathcal{A}, \forall l, l' \in \mathcal{L}^{d, d'} \quad (13) \\
\forall p \in \mathcal{P}, \forall l, l' \in \mathcal{L}^{p}, \forall a \in \mathcal{A}, \\text{loop}_l^{\text{pip}} \text{cache array}_a &= 0 \quad (14) \\
\forall p \in \mathcal{P}, \forall l, l' \in \mathcal{L}^{p}, \forall a \in \mathcal{A}, \\text{loop}_l^{\text{pip}} \times \text{loop}_{l'}^{\text{pip}} \text{cache array}_a &= 0 \quad (15)
\end{align}

### 5.4 Objective function

Lastly, we need to define the objective function ($\text{obj\_func}$) that supports fine-grained and coarse-grained parallelism. Fine-grained parallelism involves duplicating a specific statement(s), while coarse-grained parallelism duplicates modules, including statements and loops. However, it may not always be feasible to achieve parallelism based on the characteristics of the loops and the level of parallelism required. Therefore, we distinguish between parallel and reduction loops. A parallel loop can be coarse and fine-grained unrolled, whereas a reduction loop can only be fine-grained unrolled with a tree reduction process that operates in logarithmic time.
As the pragmas cache are part of the space we compute the communication latency with these pragmas. If more than one array is transferred above the same loop we take the maximum as Merlin transferred them in parallel. To ensure these properties, we formulate the objective function for each pipeline configuration. The objective function uses the combined latencies of communication and computation. When using Merlin, communication and computation do not overlap, but communication tasks can overlap when they occur consecutively in the code at the same level. Consequently, for each loop, where two arrays are transferred consecutively within the loop, we calculate the sum of the maximum latencies for these transferred arrays \((L_{mem})\).

In every loop nest, there will invariably be a pipeline loop due to either user-inserted or compiler-inserted instructions (AMD/Xilinx Merlin and Vitis automatically insert the pragma pipeline if it is not done by the user or the previous compiler). Therefore, the objective function takes the following form: \(TC_{ap} \times (IL + II \times (\frac{TC}{TF} - 1))\), where \(TC_{ap}\) includes the loops situated above the pipeline. Parallel loops above the pipeline can be coarse-grained parallelized. The iteration latency within the unrolled loop body is divided into either reduction operations or non-reduction operations, as reduction operations require logarithmic time for the reduction process. The variable IL encompasses the latencies of the statements found within the pipelined loop body. Independent statements can be executed in parallel and statements with dependencies are summed, as detailed in Section 4.1.

\[
TC_{ap} = \prod_{l \in L_{par}} TC_{loop_{UF}} \prod_{l \in L_{red}} TC_{loop} \\
IL = IL_{par} + IL_{seq} \times \prod_{l \in L_{red}} TC \times IL_{mem} \\
oj \text{func} = TC_{ap} \times (IL + II \times (\frac{TC}{loop_{UF}} - 1)) + IL_{mem}
\]

**5.5 Example**

1. Loop0: for \(i = 0; \ i < 2100; \ i++) \ S0: y[i] = 0; \\
2. Loop2: for \(i = 0; \ i < 1988; \ i++) \ S1: z[i] = 0; \\
3. Loop3: for \(j = 0; \ j < 2100; \ j++) \ S2: x[i] = a[i][j] \times b[i][j]; \\
4. Loop3: for \(j = 0; \ j < 2100; \ j++) \ S3: y[j] = a[i][j] \times b[i][j];

Listing 2: Atax Large

We now employ the Atax kernel as an illustration. \(S0 \) and \(S3\) do not have inter-iteration dependencies within their respective loops, namely Loop0 and Loop3. Therefore, it is possible to pipeline Loop0 and Loop3 with an \(II \geq 1\). In other words, for all iterations within these loops, there are no dependencies on previous iterations within the same loop. Loop1 and Loop2 are reduction loops, and the reduction operations is an addition which has an IL of \(IL_{+}\) cycles. So the \(II = IL_{+}\). If loop1 is pipelined, there is a dependency between \(S1\), \(S2\) and \(S3\). Between \(S1\) and \(S2\) the distance is 1, so we just add \(IL_{1} + IL_{2}\). Between \(S2\) and \(S3\) the distance is \(log(N)\) so the final equation will be \(IL_{1} + IL_{2} + log(N)\) for the loop body cycle. If statements can be run at the same time (i.e., there is no dependency) it is a max instead of an addition. If we encounter code like for \(j = 0; \ j < N; \ j++) y[j] = y[j-2] + 3\), a straightforward approach to handling this type of dependency is to impose a constraint such as \(loop_{UF} \leq 2\), which is represented by equation 8. In this case, due to dependencies an \(UF > 2\) is similar to \(UF = 2\).

**6 DESIGN SPACE EXPLORATION**

We now present our Design Space Exploration (DSE) approach. Our approach focuses on identifying designs with the most promising theoretical latency within the available design space. However, it may result in suboptimal designs if the selected pragmas are not applied during compilation. To address this potential issue and ensure high QoR, we conduct an additional exploration within a restricted subspace. Our DSE explores two additional parameters: the type of parallelism and the maximum array partitioning factor. Array partitioning is a technique commonly used in FPGA contexts to divide arrays or matrices into smaller sub-arrays, which can be stored in independent memory blocks known as Block RAMs (BRAMs). AMD/Xilinx HLS has a limit of 1,024 partitions per array. The array partitioning is calculated by taking the product of loops that iterate on the same arrays on different dimensions (cf. Section 5). So constraining the maximal array partitioning also constrains the maximal UF. This NLP based DSE technique is presented in the Algorithm 1. The DSE starts without constraint on parallelism and array partitioning. Then we alternate constraints on parallelism while decreasing the maximum unrolling factor and array partitioning.

**Data:**
- \(kernel\) // withoutPragma
- \(timeout_HLS, timeout_NLP\)

**Result:**
- \(kernel\) // with MerlinPragma
- \(nlp_{file} \leftarrow generate_nlp_file(kernel), min\_lat \leftarrow \infty;\)
- for \(max\_array\_partitioning \in \{\infty, 2048, 1024, 512, 256, 128, 64, 32, 16, 8, 1\}\)
  - for \(parallelism \in \{\text{coarse} + \text{fine}, \text{fine}\}\)
    - \(current\_nlp\_file \leftarrow \text{change}\_max\_array\_partitioning(copy(nlp_{file}));\)
    - \(current\_kernel \leftarrow \text{change}\_max\_array\_partitioning(copy(kernel));\)
    - \(pragma\_configuration, lower\_bound \leftarrow \text{SOLVE}(current\_nlp\_file, timeout\_NLP);\)
    - if lower\_bound < min\_lat then
      - \(current\_kernel \leftarrow \text{introduce pragma copy(kernel), pragma\_configuration};\)
      - \(hls\_lat, valid \leftarrow \text{MERLIN(kernel, timeout\_HLS)};\)
      - if valid then // no over-utilization
        - \(min\_lat \leftarrow \text{min}(min\_lat, hls\_lat);\)
    - end
  - end
- Algorithm 1: NLP-DSE

In order to reduce the maximum unroll factor and array partitioning, we modify the parameters specified in the NLP file. And we automatically add constraints (Eq 9) to restrict parallelism to fine-grained levels, as described in Section 5. The choice to restrict
the maximum array partitioning to the power of 2 is to improve the speed of the DSE. Adding possibilities would permit exploring a larger space and potentially finding a design with a faster latency at the cost of a longer DSE.

7 EVALUATION

We now present our experimental results using a set of polyhedral computation kernels.

7.1 Setup

We use kernels from Polybench/C 4.2.1 [20]. In addition we add a kernel of Convolution Neural Network (CNN). A single-precision floating point is utilized as the default data type in computations. Computations operate on medium and large datasets from Polybench/C [20] in order to have kernels with large footprint and have a large enough space to explore. The problem size and loop order of CNN are J1=256, P,Q=5, H,W=224. A description of each benchmark can be found in Table 2. The ludcmp, deriche and nusinnov kernels are not present as PolyOpt-HLS [22] does not handle negative loop stride. Cholesky and correlation contains a sqrt() operation which we do not support currently. Finally, we removed FDTD-2D because it exposed a bug in Merlin, and this generated a program where data dependencies are not fully preserved.

We evaluate designs with AMD/Xilinx Merlin [31]. The synthesis is carried out with AMD/Xilinx Vitis 2021.1. We choose the option "-funsafe-math-optimizations" to enable commutative/associative reduction operators and implementation of reductions in logarithmic time. We change the default on-chip memory size of Merlin by the size of the device we use. As the target hardware platform, we run the Xilinx Alveo U200 device where the target frequency is 250 MHz.

We analyze the kernels and automatically generate each NLP problem with a version of PolyOpt-HLS [21]. We modified and extended for our work. Employing the AMPL description language to solve the NLP problems, we ran the commercial BARON solver [23, 27] version 21.1.13. For our experiments, we utilize 2 Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz and 252GB DDR4 memory.

7.2 Experimental Evaluation

We compare our method with AutoDSE [26], described in Section 2, and we automatically generate the space of AutoDSE with the command ds_generator. We replace the UF and tile size by all the UF and tile size which divide the TC in order to have the same space. AutoDSE doesn’t impose any constraints on parallelism or the maximum array partitioning. It employs an incremental exploration approach, enabling it to make compiler-specific pragma selections.

Table 2 displays the space size of each design. The DSE is done in 4 parts with two threads for each (default parameter), with a timeout for the generation of the HLS report of 180 minutes, and a timeout of the DSE of 600 minutes (not always respected cf. Table 2). For our method we take the same parameters, and we add a timeout for BARON of 30 minutes.

7.3 Comparison with AutoDSE

Table 2 shows the comparison with AutoDSE. NL, ND, S, and Space S are respectively the number of loops, the number of polyhedral dependencies (WaR, WaW, RaW), problem size (L for Large and M for Medium) and space size. For each method we compute the throughput (GF/s) in GFLOPs per second, the total time of the DSE (T) in minutes, the number of designs explored (DE) and the number of designs timeout (DT). In addition, for AutoDSE we add the number of design that are early rejected/prune (ER) as AutoDSE prunes the design when AMD/Xilinx Merlin can not apply one of the pragmas, due to its analysis limitations.

To illustrate the performance achievable without a complete DSE, the first synthesizable design found with NLP-DSE (FS) is displayed. Indeed, due to our under-estimation of resources, the theoretically best design produced by NLP solving may not be synthesizable. We report the improvements in DSE time (T) and throughput (GF/s).

The performance of the kernel evaluated show significant improvements in both time and throughput. The time of the DSE is 5.69x faster on average (3.70x for geo-mean) and the throughput is 17.24x higher on average (2.38x for geo-mean) for the kernel evaluated. For almost all (46/47) kernels and problem sizes the method identifies a design with a throughput similar to (+/- 2%), or better than, AutoDSE. We have a slight slowdown for Doitgen Large because NLP-DSE explores the design found by the NLP with a maximum array partitioning of 2048 which timeouts, and then 1024 which is the best design found. However AutoDSE finds a design with a maximum array partitioning of 1280. By changing the maximum array partitioning to 1280 we find the same configuration as AutoDSE. Thus it is possible to obtain designs with a better performance but at the cost of a longer search. For all kernels and problem sizes, except Durbin, NLP-DSE is faster than AutoDSE. AutoDSE prunes all configurations of Durbin which explains the speed of AutoDSE for this kernel.

We can observe a difference of the performance for the same kernel in function of the problem size. If we take the example of 2mm, the difference has many factors. First as the footprint of the kernel becomes more important, it begins overusing the BRAMs. A large parallelism requires a bigger array partitioning which considerably increases the number of BRAMs and uses more BRAMs than available. Additionally, for large problems with high levels of parallelism, there are multiple instances of timeouts observed. Furthermore, the compilers applied the pragmas more efficiently for smaller problem sizes. We observe twice as many kernels where the pragmas are not applied as expected for the large problem size.

For Atax Large (Lst. 2), AutoDSE explores 166 designs of which 106 are early rejected and 30 timeout. AutoDSE starts by partially unrolling Loops 2 and 3 and will then attempt to do a coarse-grained parallelization on Loop 1 with all divisors, which is impossible due to dependencies. Although AutoDSE manages to prune/early reject the designs because Merlin cannot apply the pragmas, it still requires several minutes of compilation by Merlin for each unroll factor. In parallel, AutoDSE tries to pipeline the outermost loops (and therefore unroll the innermost loops) which creates numerous timeouts. Although the first two designs timeout due to too high level of parallelism, NLP-DSE allows us to find a configuration with the innermost loops unrolled with a UF=700. This allows us to find a design with a 3.46x higher throughput in 11.34x less time.

Our method experiences some timeouts for designs with high levels of parallelism. However, thanks to our DSE approach, we quickly identify optimized designs where each loop body has a
## Table 2: Comparison with AutoDSE

similar level of parallelism. For 20/47 cases, the first synthesizable design is equal to the best design of the DSE. This is because compilers can be conservative and not apply pragmas as expected. In this case, another configuration is applied than what was identified by the NLP, which explains the difference in performance.
7.4  Accuracy
The tightness of the lower bound estimation relies on the correct application of pragma directives such as pipeline and parallel. It also assumes that Merlin can efficiently transfer memory from off-chip to on-chip using 512-bit chunks. Finally, it assumes that Merlin optimally handles the transfer of memory from off-chip to on-chip. Figures 1a and 1b compare the measured HLS latency for every synthesizable design explored during our DSE with its predicted latency per solving the NLP. The Y-axis represents log(latency) and the X-axis the rank of the design sorted by HLS latency. For Fig 1b we exclude designs when we detect that the pragma parallel and pipeline are not applied as defined by Merlin. We observe that about half of the designs have at least one pragma not applied, leading logically to a larger difference between measured and predicted latency. Generally speaking, for parallelization pragmas, Merlin is more restrictive for coarse-grained parallelization, in many cases these pragmas are not applied. We also observe certain cases where the partitioning is not done correctly which does not allow a pipeline with II=1 when it is theoretically possible. For Figure 1b we observe a better overall accuracy, albeit imperfect. These differences are due in large part to how Merlin eventually implemented memory transfers, which we model optimistically.

7.5  Scalability
To mitigate prolonged solving times for specific kernels and problem sizes, we’ve implemented a 30-minute timeout constraint for the BARON solver. While this timeout doesn’t guarantee achieving optimality, it ensures that the solver provides the best solution it has found within the time limit. In Table 3, we present statistics regarding the number of problem timeouts (ND T/O) and problem non-timeouts (ND NT/O), along with the average time in seconds (Avg Time) for all problems and exclusively for those that did not time out. We can note that the 20 NLP problems for CNN finish in few seconds with an average of 3.71 seconds. We notice that 12 kernels exhibit at least one NLP problem that times out. To investigate scalability further, we conducted restarts for NLP problems that timed out at 30 minutes, extending the timeout to 30 hours. For 30 out of 126 problems (23.8%), we found an optimal theoretical solution within an average time of 3.13 hours. We observe that problems timing out after 30 hours often involve trip counts with numerous divisors, significantly expanding the space for the unroll factor. Consequently, non-linear conditions involving more than three unknown variables of unroll factors (UFs) become extremely time-consuming to resolve. By relaxing these constraints, we are able to find a solution in seconds but this can result in infeasible designs due to over-utilization of resources as these constraints are removed. For 23.8% of problems not timing out at 30 hours, we examined the disparity in objective function values provided by the solver when it times out at 30 minutes (representing the best solution found so far) versus when it discovers the optimal solution. For 25 out of 30 problems, the estimated latency is exactly the same. However, for the remaining 5 problems, the differences in the estimated latencies range from a mere 0.04% up to 2,426%.

8  RELATED WORK
NLP-DSE makes it possible to automatically introduce pragmas in order to obtain a design with a good QoR. Many previous works using different DSE methods have the same objective. Model-free DSEs [10, 26, 34] evaluate each candidate by generating the HLS report. The synthesis time or report generation can last several hours, which considerably reduces the explored design space. Additionally, some DSE, such as [26], may lead to local minima. In order to avoid the synthesis time limitation, model-based DSEs and AI-driven DSEs have been developed. They estimate the performance of each design using a cost model [1, 29, 37–40], Graph Neural Networks (GNN) [25, 30] or decision trees [17, 19, 35]. These DSEs estimate the QoR of the design in a few milliseconds. However, the evaluation of a large number of designs can still take several hours and the accuracy may not match the HLS report. Hence, evaluating only the top-n results with HLS may lead to sub-optimal solutions. Other works allow one-shot optimization with code transformations, e.g., [33] but the space of hardware directives and code transformations is limited. Some focus on a predefined micro-architectures, e.g., [28] but are limited in their application. NLP-DSE is a hybrid approach: our cost model in the form of NLP makes it possible to consider very large design spaces in a few minutes. This makes it potentially even faster than existing cost models. Yet we rely on HLS to ensure accurate performance.

Recent advances in optimization solvers such as BARON [23, 27] have allowed NLP-based approaches to become a promising alternative to approximate ILP-based methods, as they can encode more complex and realistic performance models. Previous works [2, 21, 41] encode a cost model as Linear Programming (LP) problem.
However, it requires an approximation, such as approximating the volume of communication inter loops [2, 21], or simplifying the space by exposing direct parallelization in the problem [41].

The selection of tile sizes remains fundamental for the final QoR. Similar to our approach, [16] uses a cost model to select the tile size. Although their space is much more complete than ours as we are restricted to Merlin’s transformations, their method does not allow the evaluation of the whole space.

9 CONCLUSION
Our work targets the automatic selection of pragma configurations for HLS with a framework that automatically inserts Merlin pragmas into loop-based programs. Our framework is guided by an analytical performance and resource model, which serves as a lower bound estimation for the achievable performance across all possible configurations.
We present in this appendix more details about key elements of our analytical performance model, and the associated proofs this model computes a lower bound on latency under resource constraints. Note we plan to publish this appendix on ArXiv if our paper is accepted.

.1 Analytical Model Template

Let \( I_i^{PV}(X) \) the operators defined as follows:

\[
I_i^{PV}(X) = [I_i \bullet (TC_{-i}^{arg} [if i – is pipeline;]) \odot X
\]

where \( \odot \) is + if is pipeline; = 1, * otherwise. We also have:

\[
c_i^{PV}(X_1, X_2, \ldots, X_n) = \max(X_1, X_2, \ldots, X_n) \text{ if } in parallel \text{, and}
\]

\[
c_i^{PV}(X_1, X_2, \ldots, X_n) = \sum_{k=1}^{n} X_k \text{ if } in sequence; \text{ with } in parallel = 1 = 1 – in sequence; \text{ if } (X_1, X_2, \ldots, X_n) \text{ do not have dependencies ( WaR, RaW, WaW) else 0. Finally, } S_i^{PV} (\overline{S_i}) \text{ denotes a region of straight-line code (e.g., an inner-loop body).}
\]

Intuitively, SL will represent a lower bound on the latency of a code block such that by composition across all loops as per the template formula, the result remains a lower bound on the full program latency.

Given the summary AST of the program in constructor form, and the set of \( PV_i \) vectors for each loop, we build the analytical formula template as follows: (1) replace the \( Loop_i(X) \) operator by \( I_i^{PV}(X) \operatorname{operator} \); (2) replace the \( \overline{X} \) list by \( C_i^{PV} (\overline{X}) \); and (3) replace statement lists inside loop \( 1 \overline{S}_i \) by \( S_i^{PV} (\overline{S}_i, \overline{X}) \). That is for our example, we simply rewrite: \( \text{loop}_j (\text{loop}_j(S1, S2, S3)) \) into \( I_j^{PV} (C_i^{PV} (I_j^{PV} (S_i^{PV} (S1)), I_j^{PV} (S_i^{PV} (S2, S3)))) \). 

.2 A Formal Model for Latency

Our objective is to formulate a lower bound on the latency of a program after HLS. We therefore have put several restrictions: we assume the input program is a polyhedral program, that is the control-flow is statically analyzable; all loops can be recognized and their trip count computed; and all array / memory accesses can be exactly modeled at compile-time. No conditional can occur in the program. While our approach may generalize beyond this class, we limit here to these strict assumptions.

To maintain a lower bound on latency by composition, we operate on a representation of (parts of the) program which is both schedule-independent and storage-independent: indeed, a lower bound on this representation is necessarily valid under any schedule and storage eventually implemented. We however require HLS to not change the count and type of operations. Furthermore, for lower bounding purposes, we assume unless stated otherwise \( vi, in parallel \) = 1. We will discuss in the next section a more realistic but compiler-dependent approach to set \( in parallel \), based on dependence analysis.

We assume programs are made of affine loops, that are loops with statically computable control-flow, with loop bounds made only of intersection of affine expressions of surrounding loop iterators and program constants. We now assume loop bodies (i.e., statements surrounded by loops) have been translated to a list of statements, with at most a single operation (e.g., +, -, /, *) per statement. Operations are n-ary, that is they take \( n \geq 0 \) input scalar values as operand, and produce 0 or 1 output scalar value. A memory location can be loaded from (resp. stored to) an address stored in a scalar variable. This is often referred to as straight-line code. This normalization of the loop body facilitates the computation of live-in/live-out data for the code block, and the extraction of the computation graph. Note the region can be represented in Static Single Assignment form, to ensure different storage location for every assignment, facilitating the construction of the operation graph. In addition we suppose that the input program don’t have useless operation i.e., there is no dead-code elimination as in the Listing 3.

Listing 3: Example with dead code elimination

```c
int example(int y, int z){
  int x;
  x = 12 + y; // dead-code elimination
  x = y + z;
  return x;
}
```

The restriction can be summarized as:

- The input program is a pure polyhedral program [11], and its analysis (loop trip counts for every loop, all data dependencies [8]) is exact.
- No HLS optimization shall change the number of operations in the computation: strength reduction, common sub-expression elimination, etc. shall either first be performed in the input program before analysis, or not be performed by the HLS toolchain.
- The program does not contain "useless" operations. A typical case is when a value is overwritten before being used.
- We only model DSP and BRAM resources for the considered kernel, ignoring all other resources.
- We assume resource (DSP) sharing across different operations executing at the same cycle is not possible.

**Definition 1 (Straight-line code).** An n-ary operation takes n scalar operands \( \overline{a} \) as input, and produces a single scalar \( o \) as output. A statement contains a single n-ary operation, or a load from (resp. store to) a memory location to (resp. from) a scalar. A straight-line code region \( L \) is a list of consecutive statements, with a single entry and single exit.

**Definition 2 (Live-in set).** The live-in set \( V_L^I \) of region \( L \) is the set of scalar values, variables and memory locations that read before being written, under any possible valid execution of \( L \).

**Definition 3 (Live-out set).** The live-out set \( V_L^O \) of region \( L \) is the set of variables and memory locations that written to during any possible valid execution of \( L \).

**Definition 4 (Operation Graph).** Given a straight-line code region \( R \) made of a list \( L \) of statements \( S \in L \), the operation graph \( OG \) is the directed graph \(< \{ N, V_I, \text{root}, V_O \}, E > \) such that \( \forall S_i \in L, N_{S_i} \in N; \)
and for every operation with output \( o \) and inputs \( i \) in \( L \) \( \forall S_i, V_i \in V \), \( e_{i,o} \in E, V_S : (O_S^i, I_S^i) \in L, S_j : (O_S^i, I_S^j) \in L \) with \( S_j \neq S_i \), then we have \( E_{S,S_j} \in E \) if \( o \in \bigcap I_{S_j} \neq \emptyset \). For every input (resp. output) in \( S_i \) which is not matched with an output (resp. input) of another \( S_j \) in \( L \), create a node \( V_{\text{real}} \in V_l \) (resp. \( V_O \)) for this input (resp. output) value. If \( \dim(\hat{I}_{S_i}) = 0 \) then an edge \( e_{\text{root},S_i} \) is added to \( E \).

From this representation, we can easily define key properties to subsequently estimate the latency and area of this code region, such as its span, or critical path.

\textbf{Definition 5 (Operation Graph critical path).} Given \( OG^L \) an operation graph for region \( L \). Its critical path \( OC^L \) is the longest of all the shortest paths between every pairs \( (v_i, v_o) \in \langle V_l, root \rangle, V_O \rangle \). Its length is noted \( #OG^L \).

\textbf{2.1 Estimating Latency.} We can build a simple a lower bound on the latency of an operation graph:

\textbf{Theorem 6 (Lower bound on latency of an Operation Graph).} Given infinite resources, and assuming no operation nor memory movement can take less than one cycle to complete, the latency \( \text{LAT}^L \geq #OG^L \) is a lower bound on the minimal feasible latency to execute \( L \).

\textbf{Proof Th. 6.} Every operation \( S_i \) is associated at least one edge with a source in \( < \{ V_l, \text{root} \} \), so there is a path between one of these nodes and every operation by construction in Def. 4. For an operation to produce a useful output, there must be a path from its output to a node in \( V_O \), otherwise the memory elimination can be removed by dead code elimination. Therefore the shortest path \( sp \) between a pair of node \( (v_i, v_o) \in \langle V_l, \text{root} \rangle, V_O \rangle \) is the shortest sequence of operations in dependence that must be executed to produce \( v_o \). As any operation takes at least one cycle to complete per Def. 6, then this path must take at least \( sp \) cycles to complete. As we take the longest of shortest paths between all possible pairs \( (v_i, v_o) \) then \( #OG^L \) is the length of the longest shortest path to produce any output \( v_o \) from some input, via a sequence of producer-consumer operations. Therefore it must take at least \( #OG^L \) cycles to execute this path.

We can then build a tighter lower bound on the number of cycles a region \( L \) may take to execute, under fixed resources, by simply taking the maximum between the weighted span and the work to execute normalized by the resources available.

\textbf{Theorem 7 (Latency Lower Bound under Operation Resource Constraints).} Given \( R_{op} \) a count of available resources of type \( op \), for each operation type, and \( LO(op) \) the latency function for operation \( op \), with \( LO(op) \geq 1 \). \( #L(op) \) denotes the number of operations of type \( op \) in \( L \). We define \( LO(#OG^L_{\text{cp}}) = \sum_{o \in \text{op}} LO(n) \) the critical path weighted by latency of its operations. The minimal latency of a region \( L \) is bounded by

\[ \text{Lat}^L_{R_{op}} \geq \max(LO(#OG^L_{\text{cp}}), \max_{o \in \text{op}}(\lceil #L(o) \times LO(o) / R_{op} \rceil)) \]

\textbf{Proof Th. 7.} Suppose \( \forall o \in \text{op}, R_{op} \geq #L(o) \). Then there is equal or more resources available than work to execute, this is equivalent to the infinite resource hypothesis of Th. 6, the minimal latency is \( LO(#OG^L_{\text{cp}}) \).

Suppose \( \exists o \in \text{op}, R_{op} < #L(o) \). Then there exists at least one unit in \( R_{op} \) that is executing \( #L(o) / R_{op} \) operations. As every operation \( op \) takes at \( L(op) \geq 1 \) cycle to complete, this unit will execute in at least \( #L(o) \times L_{op} / R_{op} \) cycles. If \( #L(o) \times L_{op} / R_{op} \geq LO(#OG^L_{\text{cp}}) \), the computation cannot execute in less than \( #L(o) \times L_{op} / R_{op} \) cycles. \( \Box \)

\textbf{2.2 Loop Unrolling: full unroll.} We start by reasoning on the bound for latency of a loop nest which has been fully unrolled, e.g. as a result of \#pragma ACCEL parallel or \#pragma HLS unroll. Full unrolling amounts to fully unroll all TC iterations of a loop, replacing the loop by \( TC \) replications of its original loop body, where the loop iterator has been updated with the value it takes, for each replication.

It follows a simple corollary:

\textbf{Corollary 8 (Equivalence between fully unrolled and straight-line code).} Given a loop nest \( L \), if full unrolling is applied to \( L \) then the code obtained after full unrolling is a straight-line code as per Def. 1.

\textbf{Proof Co. 8.} By construction the process of fully unrolling all the loops creates a straight-line code region without loop control, which therefore fits Def. 1. \( \Box \)

Consequently, we can bound the latency of a fully unrolled loop nest:

\textbf{Theorem 9 (minimal latency of a fully unrolled loop nest).} Given a loop nest \( L \), which is first rewritten by fully unrolling all loops to create a straight-line code region \( L \). Given available resources \( R_{op} \) and latencies \( L(op) \geq 1 \). Then its minimal latency is bounded by:

\[ \text{Lat}^L_{R_{op}} \geq \max(LO(#OG^L_{\text{cp}}), \max_{o \in \text{op}}(\lceil #L(o) \times L_{op} / R_{op} \rceil)) \]

\textbf{Proof Th. 9.} By Corollary 8. \( \Box \)

\textbf{2.3 Loop Unrolling: partial unroll.} Loop unrolling is a HLS optimization that aims to execute multiple iterations of a loop in parallel. Intuitively, for an unroll factor \( UF \geq 1 \), \( UF \) replications of the loop body will be instantiated. If \( TC_1 \mod UF \neq 0 \) then an epilogue code to execute the remaining \( TC_1 \mod UF \) iterations is needed.

Partial unrolling can be viewed as a two-step transformation: first strip-mine the loop by the unroll factor, then consider the inner loop obtained is fully-unrolled. The latency of the resulting sub-program is determined by how the outer-loop generated will be implemented. We assume without additional explicit information this outer loop will execute in a non-pipelined, non-parallel fashion, to provide the following bound:

\textbf{Theorem 10 (Minimal latency of a partially unrolled loop with factor UF).} Given a loop \( L \) with trip count \( TC_1 \) and loop body \( L \), and unroll factor \( UF \geq TC \). Given available resources \( R_{op} \) and latencies \( L(op) \geq 1 \). Given \( L \) the loop body obtained by replicating \( UF \) times the original loop body \( L \). Then the minimal latency of \( L \) if executed in a non-pipelined fashion is bounded by:

\[ \text{Lat}^L_{R_{op}} \geq \lfloor TC_1 / UF \rfloor \times \text{Lat}^L_{R_{op}} \]
Given a loop with factor $UF$ and complex loop bodies.

\[ \text{Lat}_{\text{unroll}} \leq \log (\text{Trip Count}) \]

The sequential execution of the loops without pragma repeat this process $[TC/UF]$ times.

**Proof Th. 10.** By construction and Theorem 7, $\text{Lat}_{\text{unroll}}^{\prime}$ is a lower bound on the latency of $L^\prime$, that is the sub-program made of $UF$ iterations of the loop. \[ \frac{TC}{UF} \leq TC_1/UF \] is a lower bound on the number of iterations of the loop. As we assume a non-pipelined execution for the resulting outer loop, every iteration shall start after the completion of the preceding one, that is its iteration latency, itself bounded by $\text{Lat}_{\text{unroll}}^L$.

Note this bound requires to build the operation graph for the whole loop body. This is straightforward for inner loops and/or fully unrolled loop nests, but impractical if the loop body contains other loops. We therefore define a weaker, but more practical, bound:

**Theorem 11 (Minimal latency of a partially unrolled loop with factor UF and complex loop bodies).** Given a loop $l$ with trip count $TC_1$ and loop body $L$, and unroll factor $UF \leq TC$. Given available resources $R_{op}$ and latencies $L(\text{op}) \geq 1$. Then the minimal latency of $l$ if executed in a non-pipelined fashion is bounded by:

\[ \text{Lat}_{\text{unroll}}^L \geq \frac{TC}{UF} + \text{Lat}_{\text{unroll}}^L \]

**Proof Th. 11.** Given $OG_l$ and $OG_j$ two CDAGs, for a pair of distinct iterations $i, j$ of loop $l$.

If $V_i \cap V_j = \emptyset$, then the graph $OG^{i,j}$ made of the two iterations $i, j$ cannot have a smaller critical path length than $OG_l$ and $OG_j$; there is no edge crossing $OG_l$ and $OG_j$ in $OG^{i,j}$ since outputs are distincts, therefore $\text{Lat}(OG^{i,j}) \geq \max(\text{Lat}(OG_l), \text{Lat}(OG_j))$.

If $V_i \cap V_j \neq \emptyset$. Then iterations $i$ and $j$ produce at least one output in common. As there is no useless operation, the graph $OG^{i,j}$ made of the two iterations $i, j$ can not be smaller than $OG_l$ or $OG_j$ and hence $\text{Lat}(OG^{i,j}) \geq \max(\text{Lat}(OG_l), \text{Lat}(OG_j))$. □

Vitis allows to do a reduction with a tree reduction in logarithmic time with the option “unsafe-math”.

**Theorem 12 (Minimal latency of a partially unrolled loop with factor UF and reduction loop with tree reduction).** Given a reduction loop $l$ with trip count $TC_1$ and loop body $L$, and unroll factor $UF \leq TC$. Given available resources $R_{op}$ and latencies $L(\text{op}) \geq 1$. Then the minimal latency of $l$, if executed in a non-pipelined fashion and the reduction tree is legal is bounded by:

\[ \text{Lat}_{\text{unroll}}^{L_S} \geq \frac{TC}{UF} \times \text{Lat}_{\text{unroll}}^L \times \log_2 (UF) \]

**Proof Th. 12.** By definition a reduction loop is a loop with a dependency distance of 1. Hence, at each iteration the same memory cell is read and write. Because of the dependency distance of 1, only one element can be add to the same memory cell. However each data can be add independently two by two and the result of this independent addition can also be add two by two until we obtained one value. Hence the reduction can be done in $\log_2 (UF)$ iterations with a tree reduction. As the depth of the tree is $\log_2 (UF)$ and each node at the same depth can be execute independently in $\text{Lat}_{\text{unroll}}^L$ cycles the straight line code has a latency greater or equal to $\text{Lat}_{\text{unroll}}^L \times \log_2 (UF)$.

.24 Loop pipelining. Loop pipelining amounts to overlapping multiple iterations of the loop, so that the next iteration can start prior to the completion of the preceding one. The initiation interval (II) measures in cycles the delay between the start of two consecutive iterations.

It follows a bound on the minimal latency of a pipelined loop:

**Theorem 13 (Minimal latency of a pipelined loop with known II).** Given a loop $l$ with trip count $TC_1$ and loop body $L$. Given available resources $R_{op}$ and latencies $L(\text{op}) \geq 1$. Then the minimal latency of $l$ if executed in a pipelined fashion is bounded by:

\[ \text{Lat}_{\text{unroll}}^L \geq \text{Lat}_{\text{unroll}}^L + II \times (TC_1 - 1) \]

**Proof Th. 13.** $\text{Lat}_{\text{unroll}}^L$ is the minimal latency to complete one iteration of $l$ by Theorem 7. The initiation interval measures the number of elapsed cycles before the next iteration can start, it takes therefore at least $TC_1 + II - 1$ to start $TC_1 - 1$ iterations, irrespective of their completion time. Therefore the latency of the loop is at least the latency of one iteration to complete, and for all iterations to be started.

.2.5 Non-Parallel, Non-Pipelined Loops. We conclude by a trivial case: if the loop is not optimized by any directive (including any automatically inserted by the compilers), that is is not parallelized nor pipelined, then every next iteration of the loop starts only after the end of the prior iteration.

**Definition 14 (Lower bound on latency of a non-parallel, non-pipelined loop under resources constraints).** Given a loop $l$ with trip count $TC_1$ which is neither pipelined nor parallelized, that is, iteration $i + 1$ starts after the full completion of iteration $i$, for all iterations. Given $\text{Lat}_{\text{unroll}}^L$ the minimal latency of its loop body. Then

\[ \text{Lat}_{\text{unroll}}^L \geq TC_1 \times \text{Lat}_{\text{unroll}}^L \]

.2.6 Program latency lower bound. We now focus on having the program latency lower bound, with resource constraints. This bound takes into account the limitations imposed by available resources, which can significantly affect the achievable performance. We assume here that the resources consumed are only consumed by the computing units and resource use by the computational unit of one operation can not be reused by the computational unit of another operation executing at the same time. We also assume that the compilers have implemented the pragma configuration given as input.

For DSPs we suppose we have a perfect reuse i.e., that the computation units for the same operation can be reused as soon as the computation unit is not in use. Under-estimating the resources used is fundamental to proving the latency lower bound, as otherwise another design that consumes less resource than predicted may be feasible, itself possibly leading to a better latency.

**Theorem 15.** Given a loop body $L$, the set of statements $S$, $#\text{op}$ the number of operation $\text{op}$ for the statement $s$, $\text{DSP}_{op}$ the number of resource (DSPs) used for the operation $\text{op}$, $\text{MCU}_{op}$ the maximal computational unit the statement $s$ can use in parallel and the configuration of pragma which is constitute of the vectors $\vec{P}_i$ for each loop.
The minimal number of resource (DSPs) consumed, \( P_{\text{used}}^{\min} \), by \( L \) for the pragma configuration is the sum, for each operation, of the maximum number of DSPs used in parallel by a statement. This corresponds to:

\[
P_{\text{used}}^{\min} = \sum_{\text{op} \in S} \max_{v \in S}(\#I_{\text{op}}^v \times DSP_{\text{op}} \times MCU_{\text{op}}^v)
\]

**Proof Th .15.** Considering perfect resource reuse, where all unused computational units can be reused, and assuming that the compilers have implemented the pragma configuration provided as input. For each statement \( s \), the maximum number of computational units used in parallel is determined. This means that each statement \( s \) requires at least \( \#I_{\text{op}}^v \times DSP_{\text{op}} \times MCU_{\text{op}}^v \) DSPs. By considering the maximum across all statements, we can guarantee that at least one statement will require \( \max_{v \in S}(\#I_{\text{op}}^v \times DSP_{\text{op}} \times MCU_{\text{op}}^v) \) DSPs. Since there is no possibility of resource reuse between different operations, the summation of the resource consumed for each operation remains the minimum consumption of resources. In other words, the sum of the individual resource consumption for each operation represents the minimum amount of resources required.

The lower bound of a kernel is therefore the lower bound of the configuration such that the minimal resource consumption is less than or equal to the resource available.

**Theorem .16 (Computation Lower bound of the kernel under resource constraint).** Given a loop body \( L \), the set of statements \( S \), \( \#I_{\text{op}}^v \), the number of operation \( \text{op} \) for the statements \( s \), \( DSP_{\text{op}} \) the maximal number of resources (DSPs) used for the operation \( \text{op} \), \( MCU_{\text{op}}^v \) the maximal computational unit the statement \( s \) can use in parallel and the configuration of pragma which is constituent of the vectors \( PV_i \) for each loop.

The configuration of pragma, which has the minimal lower bound and which respect \( P_{\text{used}}^{\min} = \sum_{\text{op} \in S} \max_{v \in S}(\#I_{\text{op}}^v \times DSP_{\text{op}} \times MCU_{\text{op}}^v) \leq R_{\text{op}} \), is the lower bound of the kernel.

**Proof Th .16.** According to Theorem .15, \( P_{\text{used}}^{\min} \) is the minimal number of DSP a configuration can use. Hence, all configuration which does not respect the constraint \( P_{\text{used}}^{\min} \leq R_{\text{op}} \) have a over-utilization of the resource and hence are invalid.

Hence, the minimal lower bound of each configuration of pragma is the lower bound of the kernel.

2.7 Memory transfer. AMD/Xilinx Merlin manages automatically the memory transfer. The memory transfer and computation are not overlap (no dataflow) hence the latency is the sum of the latency of computation and communication. We assume that for each array the contents of the array are in the same DRAM bank.

**Theorem .17 (Lower bound of the memory latency for an array).** Given a loop body \( L \), the set of array \( A \), an array \( a \in A \), and \( LAT_{a}^{\text{mem}} \) the latency to transfer the array \( a \) from off-chip to on-chip (inputs) and from on-chip to off-chip (outputs). \( \forall a \in A, LAT_{a}^{\text{mem}} \geq (1_{a \in V_O^L} + 1_{a \in V_I^L}) \times footprint_a / \text{max}_\text{burst}_\text{size} \). With \( 1_{\text{cond}} = 1 \) if \( \text{cond} = \text{true} \) else 0.

**Proof Th .17.** In order to transfer all the element of the array \( a \) we can use packing with a maximum packing allowed by the target device of max_burst_size, which mean in practice the real burst size will be equal or less than max_burst_size. As all the elements of the array \( a \) are in the same bank the transfer is sequential. And as we as suppose all operation including memory transfer are done in at least one cycle the minimum latency is footprint_a / max_burst_size to transfer time one the array \( a \). As an array can be input, output or both we need to add the transfer from off-chip to on-chip for inputs i.e. \( a \in V_O^L \) and from on-chip to off-chip for the outputs i.e. \( a \in V_I^L \).

**Theorem .18 (Lower bound of the memory latency).** Given a loop body \( L \), the set of array \( A \), the number of cycle to transfer the array \( a \) is bounded by \( \max_{a \in A}(1_{a \in V_O^L} + 1_{a \in V_I^L}) \times \text{footprint}_a / \text{max}_\text{burst}_\text{size} \).

**Proof Th .18.** According to Th .17 the lower bound to transfer one array \( a \) is \( (1_{a \in V_O^L} + 1_{a \in V_I^L}) \times \text{footprint}_a / \text{max}_\text{burst}_\text{size} \). As the array can be on different DRAM banks the transfer from off-chip to on-chip can be done in parallel but at least one array have a latency greater or equal to \( (1_{a \in V_O^L} + 1_{a \in V_I^L}) \times \text{footprint}_a / \text{max}_\text{burst}_\text{size} \) and hence the memory latency is equal to \( \max_{a \in A}(1_{a \in V_O^L} + 1_{a \in V_I^L}) \times \text{footprint}_a / \text{burst}_\text{size} \).

2.8 Summary. We have the lower bound of the computation and communication. As the computation and communication are not overlap the lower bound of a kernel is the sum of lower bound of the computation for this kernel and of the lower bound for the communication for this kernel. This can be summarize with the theorem .19.

**Theorem .19 (Lower bound of the kernel).** Given a loop body \( L \), the set of array \( A \), \( \#I_{\text{op}}^v \), the number of operation \( \text{op} \) for the statements \( s \), \( DSP_{\text{op}} \) the maximal number of resources (DSPs) used for the operation \( \text{op} \), \( MCU_{\text{op}}^v \) the maximal computational unit the statement \( s \) can use in parallel and the configuration of pragma which is constituent of the vectors \( PV_i \) for each loop.

The configuration of pragma, which has a sum of the minimal lower bound of computation and communication and which respect \( P_{\text{used}}^{\min} = \sum_{\text{op} \in S} \max_{v \in S}(\#I_{\text{op}}^v \times DSP_{\text{op}} \times MCU_{\text{op}}^v) \leq R_{\text{op}} \), is the lower bound of the kernel.