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

STÉPHANE POUGET, University of California, Los Angeles
LOUIS-NOËL POUCHET, Colorado State University
JASON CONG, University of California, Los Angeles

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 optimizations can be described by inserting pragmas for e.g. 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 an analytical performance and resource model as a function of the input program properties and pragmas inserted, using non-linear constraints and objectives. We prove this model provides a lower bound on the actual performance after HLS. We then encode this model as a Non-Linear Program, by making the pragma configuration unknowns of the system, 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, driven by lower bounds on achievable latency. 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.

Additional Key Words and Phrases: HLS, FPGA, Non-Linear Programming, Program Optimization

ACM Reference Format:
Stéphane Pouget, Louis-Noël Pouchet, and Jason Cong. 2024. Automatic Hardware Pragma Insertion in High-Level Synthesis: A Non-Linear Programming Approach. In . ACM, New York, NY, USA, 48 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

High-level synthesis (HLS) [8, 51] compilers [16, 25, 33, 44] and source-to-source compiler for HLS [15, 18, 20, 43, 46, 47] 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 [34, 38, 42] 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 [38].

Our main objective in this work is to provide a system to automatically insert a set of hardware pragmas for HLS, that delivers a good QoR and yet significantly reduces the search time needed to obtain the final design. To address this challenge, we propose NLP-DSE , a framework built on top of the AMD/Xilinx Merlin source-to-source compiler [43]. This framework automatically inserts pragmas for unrolling/parallelization, pipelining, tiling and data caching *for affine programs* [19], prior to HLS. These Merlin pragmas can also be inserted using a DSE approach, such as AutoDSE [38] or

HARP [35], which we use as reference for our evaluations. However, in contrast to AutoDSE [38] and HARP [35], we specifically restrict the class of programs we manipulate to affine programs that are regular loop-based computations. In turn, it enables us to develop a hybrid analytical approach to drive the search, combined with a lightweight DSE to reduce the number of designs actually explored. NLP-DSE preserves, and often even improves, the final QoR of designs produced.

To this end, we create a novel Non-Linear Programming approach to automatically insert pragmas in an existing program. We develop an analytical model combining latency and resources, targeting regular loop-based kernels [27], that is parameterized by the pragma configuration. We can then designate the pragma configuration as the unknowns of this model, solving it by NLP to obtain the set of pragmas that minimizes latency. An important design principle of our approach is to ensure that the latency computed *is a lower bound on the achievable latency for a given pragma configuration*. This enables efficient pruning during DSE: any design predicted to have a latency lower bound higher than the best latency obtained through exploration so far is necessarily slower and does not need exploration. To overcome the fact that optimizing compilers, and the overall HLS toolchain underneath, may not apply optimizations as expected (e.g., due to insufficient resources, or limitations of the compiler's implementation), we develop a lightweight NLP-based DSE approach, exploring parts of the design space with different types and amounts of hardware parallelism and array partitioning factors. We make the following contributions:

- We present an analytical performance and resource model specifically built for AMD/Xilinx Vitis and Merlin compilers, which is amenable to optimization via non-linear programming.
- We prove our model is a lower bound on the final latency of the design, under reasonable hypothesis. This enables fast pruning of the design space: it ensures designs which have a higher latency lower bound than the best design found so far can be safely pruned from the search.
- We develop an NLP-based DSE approach exploiting this model, targeting regular loop-based kernels, which can significantly outperform DSE-based search approaches such as AutoDSE, delivering equal or better QoR in a significantly less search time.
- We implement NLP-DSE it as an end-to-end, fully automated system and use it to conduct extensive evaluation on 47 benchmarks including kernels from linear algebra, image processing, physics simulation, graph analytics, datamining, etc. [27]. Our results show the ability of our approach to find in most cases a better QoR than AutoDSE, in significantly less time. Furthermore, in most instances, our approach outperforms HARP in terms of QoR within a comparable timeframe.

The paper is organized as follows. Section 2 motivates our approach and solution proposed. Section 3 presents our analytical performance and resource model. Section 4 delves into proving it 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 10 are devoted to evaluating our method validating the effectiveness of our approach and presenting related work, before concluding.

2 BACKGROUND AND MOTIVATION

2.1 Pragma-based Optimizations for HLS

This work targets the automatic optimization of FPGA designs using HLS [8, 51], in particular when using compilers to automatically generate HLS-friendly optimized programs such as the AMD/Xilinx Merlin Compiler [6, 7, 43]. Falcon

Computing Solutions developed this source-to-source automation tool for FPGAs, which was acquired by Xilinx in late 2020 and is now open source.

HLS has made FPGA usage more accessible, and many projects are looking to further democratize this field by automating optimizations [13, 15, 18, 38, 46]. Merlin was developed to improve the performance, reduce the development time of HLS-based designs and simplify the search space. 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. Moreover, Merlin does specific code transformation for the targeting hardware in adequacy with the hardware directive describe by the user. Additionally, it enables parallel coarse-grained processing elements by encapsulating inner loops as functions.

Merlin includes hardware directive 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. Merlin will insert the array partitioning corresponding to the parallelization and optimize memory coalescing accordingly for the memory transfer;
- `ACCEL pipeline <II=y>` for pipelining;
- `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;
- `ACCEL cache <array=a>` which transfers all required elements of array a from off-chip to on-chip to perform computations within the specified sub-region. If the user does not specify this pragma, it can be applied automatically by Merlin.

In this work we target the automatic generation of pragmas for the Merlin toolchain, to enable the seamless deployment of optimizations such as array partitioning, off-chip data transfers using bursts, coarse-grain and fine-grain replication, etc. These pragma-directed optimizations are implemented by Merlin on loop-based programs, combining source code transformations and the automatic insertion of Vitis pragmas to drive the HLS process.

We note our approach is not restricted to Merlin, nor a particular version of a toolchain: by adjusting the parameters of the performance model, such as operation latency, resource usage per operation(s), etc. one can easily target other toolchains than the one we evaluate here.

2.2 DSE for Pragma Insertion

Design-space exploration techniques typically trade-off coverage for speed [38, 56]. That is, it may impose restrictions on the input programs supported, on the pragmas/transformations considered, etc. in order to accelerate the search [28, 56]. To overcome the difficulty of providing accurate performance models for arbitrary programs, HLS may be invoked to obtain a QoR estimate, without imposing any restriction on the input programs and transformations used. However 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. General-purpose DSE approaches such as AutoDSE [38] are by design agnostic to the input program features and search space explored, preserving generality. But as we demonstrate in this paper, it also misses opportunities for search acceleration that can be provided by careful static analysis, leading to missed performance opportunities.

In this work, we target the *specialization* of the DSE process to *affine programs*, that are programs with a statically analyzable control-flow and dataflow. By restricting the class of programs supported to affine programs, we can deploy *exact* loop and data dependence analysis [12]. More importantly, as shown in Section 5, for this class of programs we can model accurately enough the behavior of a design in terms of latency and resource usage by using *non-linear programming*, significantly accelerating the DSE time for such programs by avoiding the need for actual HLS estimation in numerous cases.

We illustrate the performance merits and limitations of such general-purpose DSE [38] on three important loop-based linear algebra benchmarks. *GEMM*, the classical dense general matrix-multiply, and *2mm* shown in Listing 1 which computes the product of three matrices $D = alpha * A * B * C + beta * D$. Both are key computations in e.g., inference of transformers [9]. *Gramschmidt* computes QR decomposition using the Gram-Schmidt process. In later Section 7 we evaluate these benchmarks using various problem sizes, ranging from kBs to MBs of footprints for the matrices to demonstrate the robustness of our approach to varying and large problem sizes. Below we use matrices of about 300kB each.

```
    Loop0: for (i1 = 0; i1 < 180; i1++)
      Loop1: for (j1 = 0; j1 < 190; j1++) {
        S0: tmp[i1][j1] = 0.0;
        Loop2: for (k1 = 0; k1 < 210; ++k1)
          S1: tmp[i1][j1] += alpha * A[i1][k1] * B[k1][j1];
      }
    Loop3: for (i2 = 0; i2 < 180; i2++)
      Loop4: for (j2 = 0; j2 < 220; j2++) {
        S2: D[i2][j2] *= beta;
        Loop5: for (k2 = 0; k2 < 190; ++k2)
          S3: D[i2][j2] += tmp[i2][k2] * C[k2][j2];
      }
}
```

Listing 1. 2mm code: $D = \alpha \times A \times B \times C + \beta \times D$

The search spaces considered here quickly reach billions of feasible designs, even for kernels containing only a handful of loops and statements. Considering *2mm*, each loop can have a pragma tile and parallel, all with factors that are divisors of the loop trip count and a pragma pipeline. We obtain a space of 1.37×10^{10} **valid** designs. This represents **432 years** if assuming one design takes a single second to evaluate. Obviously, only a minimal fraction of these spaces is actually explored, making it essential to adequately select the order in which designs are explored.

Table 1 displays the performance (in GigaFlop/s, GF/s) of the original programs, from PolyBench/C [27], when fed to Merlin as-is. The best design found by AutoDSE, given a time budget of 20 hours per benchmark and a timeout of 3 hours per HLS run, is also reported. AutoDSE uses a bottleneck-driven search approach, which targets the improvement of the code section with the lowest throughput [38]. It unambiguously achieves particularly solid improvements over a naive design without pragmas. However, we show in Table 3 below that a carefully built DSE technique, exploiting the regularity of affine programs and leveraging non-linear programming, can provide order(s) of magnitude higher performance for these exact benchmarks, all while using significantly less search time.

	2mm (footprint: 773kB)		**Gemm** (footprint: 579kB)		**Gramsch.** (footprint: 15MB)	
	GF/s	Time (min)	GF/s	Time (min)	GF/s	Time (min)
Original	0.10	N/A	0.07	N/A	0.14	N/A
AutoDSE	0.41	1,870	68.91	1,345	0.95	819
Improvement	4.1x		984x		6.8x	

Table 1. Comparison of throughput (GF/s) between the AutoDSE framework and the source-to-source compiler Merlin without pragma insertion for the kernels 2mm, Gemm and Gramschmidt

2.3 Limitations of General-Purpose DSE

By analyzing the space explored by the DSE for these three examples, valuable hints can be observed which drive the design of NLP-DSE .

Exploration of the space. AutoDSE [38] utilizes the HLS compilers as a black box, in order to select the configurations that minimize the objective function. The tools are agnostic of the input program shape and if AutoDSE 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. These frameworks use incremental DSEs, i.e., having no information on the characteristics of the program, they explore the space by increasing the parallelism in order to respond to a problem, e.g., a bottleneck for Auto-DSE.

Table 2 shows the number of valid designs in each space and the number of synthesized, pruned, timeout designs for each kernel. As we can see with a timeout of 20 hours for the DSE and a timeout of 3 hours for each synthesis, the DSE only allows a tiny part of the space to be explored.

	2mm	**Gemm**	**Gramsch.**
Nb. valid designs (Space)	1.37×10^{10}	2.30×10^{6}	1.22×10^{8}
Nb. design Synthetized (AutoDSE)	15	25	15
Nb. design pruned (AutoDSE)	49	34	239
Nb. design timeout (AutoDSE)	37	27	11
Nb. Design explored (AutoDSE)	101	86	265

Table 2. Investigation into the design space and exploration extent concerning synthesized designs, pruned designs, and designs reaching timeout by the AutoDSE framework for the kernels 2mm, Gemm, and Gramschmidt.

Over Parallelization. AutoDSE is an incremental method: in order to speed up the search AutoDSE will seek to pipeline certain loops which leads to an unrolling of the innermost loops. Without knowledge of the code, trip counts and resources used this leads to over-use of parallelism, leading to timeouts and/or over-use of resources. For *2mm*, it attempts to pipeline the outermost loops, leading to the above issues.

Parallelism imbalance. Botlleneck analysis will make it possible to select which part of the code to optimize as a priority [38]. However, this priority does not take into account parallelism (i.e., hardware resources) that shall be deployed for other parts of the code. This creates code with regions that are extremely/fully parallelized, and others without any parallelism.

For 2mm, the fastest design found by AutoDSE mainly optimizes one 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 unrolls factors for the innermost loop before optimizing the other pragmas creates a loss of performance. For Gemm and Gramschmidt, the DSE finds 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.

2.4 Overview of NLP-DSE

NLP-DSE targets the (conservative) modeling of the performance and resources used by a design, such that arbitrary pragma configurations from Section 2.1 are applied on a regular, loop-based affine program. It deploys accurate static analysis to reason on the input program features, and a complex non-linear analytical performance model to drive the design space exploration. That is, *NLP-DSE is a method for automatic pragma insertion that is specialized to affine programs*. As demonstrated below, this specialization enables significantly better QoR and DSE time for affine programs than general-purpose DSE approaches such as AutoDSE.

To make our approach feasible and maintain sufficient accuracy in analytical models, we focus on programs with static control flow that can be exactly captured using *polyhedral structures* [12]. These affine, or polyhedral, programs can be analyzed to obtain the exact information about loop trip counts and dependencies, enabling more accurate performance predictors [56]. The Affine MLIR dialect specializes in modeling such programs [19, 46].

Although NLP-DSE it can be used to compute pragmas without any DSE, the inherent limit of analytical models persist with NLP-DSE : as the implementation details of the back-end toolchain for HLS and synthesis may not be accurately captured by a model, DSE remains needed for best performance.

NLP-DSE enables the exploration of different parallelism and configuration spaces by constraining the level of parallelism, as detailed in Section 6. 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 provided in Section 5.

The effectiveness of our framework is demonstrated in Table 3, comparing the performance and time-to-solution of NLP-DSE . We also display the result of the first synthesizable design produced, NLP-DSE -FS.

	2mm			**Gemm**			**Gramsch.**		
	GF/s	Time (mn)	DSP (%)	GF/s	Time (mn)	DSP (%)	GF/s	Time (mn)	DSP (%)
Original	0.10	N/A	0	0.07	N/A	0	0.14	N/A	0
AutoDSE	0.41	1,870	14	68.91	1,345	10	0.95	819	1
NLP-DSE -FS	13.19	N/A	24	105.18	N/A	26	2.34	N/A	2
NLP-DSE	117.48	70	39	105.18	185	26	2.34	420	2
Imp. vs. AutoDSE	286x	26x		1.5x	7.2x		2.4x	1.9x	

Table 3. Comparison of NLP-DSE , NLP-DSE -FS, which provides the result of the first synthesizable design, the source-to-source compiler Merlin without pragma insertion, and AutoDSE in terms of throughput (GF/s), DSE time (mn) and DSP utilization (%) for the kernels 2mm, Gemm, and Gramschmidt.

For *Gemm* and *Gramsch* the first design synthetizable has the best QoR of our DSE. For these two kernels, NLP-DSE implements better parallelism usage compared to AutoDSE. Specifically, our methodology successfully identified configurations with more balanced levels of parallelism. In contrast, AutoDSE failed to achieve the same level of parallelization within the fixed time for the DSE. On one hand, AutoDSE tends to explore first configurations with

low levels of parallelism. On the other hand, it concurrently explores design spaces with excessively high levels of parallelism, leading to timeouts and unmet resource constraints. This discrepancy highlights the merits of seeding the DSE with configurations optimized for maximum parallelism, and systematically adjusts this level based on hardware directives and compiler expectations, as implemented in NLP-DSE .

For *2mm*, the first design synthetizable allows us to have a better QoR vs. AutoDSE but our DSE shows its interest in finding a configuration 8.9 times faster than the first configuration found by the DSE. More detail can be found in Section 8.

It is noteworthy that our Design Space Exploration (DSE) approach, detailed in Section 6, deviates intentionally from AutoDSE. Unlike AutoDSE, which starts with a pragma-free configuration and gradually introduces pragmas, we begin with configurations characterized by the lowest theoretical latency, emphasizing high levels of parallelism. This deliberate departure from the conventional approach is further discussed in Section 6.

In Section 7, we present a comprehensive evaluation of our framework, demonstrating the improvements over AutoDSE that can be achieved by specializing to affine programs, in terms of design throughput and time-to-solutions across various benchmarks. The results indicate an average performance improvement of 5.69x and 17.24x in terms of DSE time and design throughput, respectively, with only a marginal decrease in throughput for 1 out of 47 benchmarks. Importantly, the time-to-solution of NLP-DSE it is consistently up to 30x faster than AutoDSE across all benchmarks.

3 MODELING PROGRAMS AND THEIR PRAGMAS

We now present our analytical performance model. We assume the input programs are polyhedral programs [12, 14], and therefore exact loop trip counts can be computed by static analysis, similarly for all data dependencies.

3.1 Program Representation

We represent programs using a summary of their Abstract syntax tree (AST), with sufficient information to estimate latency and resource consumption by analytical modeling. Intuitively, we can build a constructor-style description of the summary AST, and then directly instantiate the complete formula for estimating e.g., latency, based on loop properties. We first introduce this representation before proving how to compute a latency lower bound with it.

We employ the code below as a running example with the pragma above the loops as AMD/Xilinx Merlin. For presentation simplicity, we assume each loop iterator in the program region has been renamed to a unique name, so that we can uniquely identify loops in the code by their iterator name.

```
<some-pragmas-for-loop-i>
for (i = lbi; i <= ubi; i++) {
    <some-pragmas-for-loop-j1>
    for (j1 = lbj1(i); j1 <= ubj1(i); j1++)
        S1(i,j1);
    <some-pragmas-for-loop-j2>
    for(j2=lbj2(i);j2<=ubj2(i);j2++){
        S2(i,j2);
        S3(i,j2);
    }
}
```

The summary AST for this loop is simply built by creating one node per `for` loop and one per statement `Sx`, the body of a loop is made of loops and/or statements, and their nodes are children of said loop in the tree, listed in their

syntactic order. For example, above, utilizing a constructor notation, it gives: $Loop_i(Loop_{j1}(S1), Loop_{j2}(S2, S3))$. Then, a simple rewrites of this tree using loop properties and composition operators will lead to the proposed analytical model, as outlined below. We first describe the loop properties we associate to each loop. We consider combinations of the following pragmas, based on Merlin's optimizations, for the loop with iterator `i`:

- `#pragma ACCEL parallel <factor=ufi>`
- `#pragma ACCEL pipeline <II=IIi>`
- `#pragma ACCEL tile`
- `#pragma ACCEL cache <variable=a>`

We therefore associate to each loop `i` in the program a *property vector* that informs about the optimizations to be considered. We define $\vec{PV}_i$ as follows: $\vec{PV}_i :< ispipelined_i, II_i, uf_i, tile_i, TC_i^{min}, TC_i^{max} >$ where we have: $ispipelined_i = 1$ if the loop is pipelined, 0 otherwise; II_i is the initiation interval, set to 1 by default; uf_i is the parallelism/unroll factor, set to 1 by default (no `#pragma ACCEL parallel` pragma) and set to TC_i^{max} if `parallel` is defined without a factor uf_i specified. $tile_i$ is the TC of the innermost loop after strip mining. TC_i^{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 [29]. 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 $PV_i :< 0, 1, 1, 1, TC_i^{min}, TC_i^{max} >$ is used for a loop without any pragma. Once all loops have been annotated by their $\vec{PV}$ properties, subsequent treatment can be implemented to mirror the optimizations implemented by the back-end tool.

Modeling Vitis optimizations. AMD/Xilinx Vitis will apply several optimizations automatically, such as auto-pipeline and auto-loop-flatten, some other optimizations when the user gives a compilation option such as tree reduction.

Only a loop with a constant TC, i.e., $TC_{max} = TC_{min}$ can be unrolled. The unroll pragma options allow to specify the unrolling factor, uf_i. 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 pipeline`. The pipeline pragma options allow to specify the objective II the user wants to achieve. When II is not specified, it is automatically set to 1. In addition, Vitis will auto-pipeline with a target II of 1 the innermost loops which are not fully unrolled for each nested loop.

Within Vitis, users can enable optimizations like logarithmic time reduction through tree reduction. This optimization choice will be a global option within our model, applicable across the entire model rather than being limited to a specific loop.

Modeling Merlin 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 $\vec{PV_l}$ vectors are adjusted by analyzing the input code, and modifying their initial value, possibly further constraining the set of possible $\vec{PV_l}$ based on which program transformation will be performed, as described above. Overall, the $\vec{PV_l}$ 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 MODELING OF LATENCY AND RESOURCE LOWER BOUND

We present the foundational components of our analytical performance model and demonstrate how this model computes a lower bound on latency while adhering to resource constraints. The theorems and their accompanying proofs can be found in Appendix B.

4.1 Analytical Model Template

Consider the expression $I_i^{\vec{PV}_i}(X)$ for the loop i, where X denotes a subpart of the program, and the operators are defined as follows:

$$I_i^{\vec{PV}_i}(X) = \begin{cases} \left\lfloor II_i \cdot \left(\frac{TC_i^{avg}}{uf_i} - \text{ispip}_i \right) \right\rfloor + X & \text{if loop } i \text{ is pipelined} \\ & (\text{ispip}_i = 1) \\ \left\lfloor II_i \cdot \left(\frac{TC_i^{avg}}{uf_i} - \text{ispip}_i \right) \right\rfloor \cdot X & \text{otherwise } (\text{ispip}_i = 0) \end{cases}$$

For simplicity, we will represent the operation as $\odot$ to differentiate between the two different cases.

Let $C_i^{\vec{PV}_i}(X_1, X_2, \ldots, X_n)$ which compose the different sub part of a program under the loop i.

$$C_i^{\vec{PV}_i}(X_1, X_2, \ldots, X_n) = \begin{cases} max(X_1, X_2, \ldots, X_n) & \text{if } (X_1, \ldots, X_n) \text{ do not have dependencies (WaR, RaW, WaW)} \\ \sum_{k=1}^{n} X_k & \text{otherwise} \end{cases}$$

```
1 S0: x1 += y;
2 S1: x2 += x1 + z;
```

Listing 2. Example of code where two statements have dependencies

```
1 S0: x1 += y;
2 S1: x2 += z;
```

Listing 3. Example of code where two statements do not have dependency

In List 2, statements $S0$ and $S1$ exhibit dependencies, resulting in a program latency equal to the sum of their latencies, corresponding to $issequence = 1$. Conversely, the statements in List 3 lack dependencies, allowing $S0$ and $S1$ to be executed concurrently. Therefore, the latency is determined by the maximum of $S0$ and $S1$ latencies, representing the case $inparallel = 1$.

Finally, $SL_i^{\vec{PV}_i}(\vec{S}_k)$ 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 $\vec{PV}_i$ vectors for each loop, we build the analytical formula template as follows:

(1) replace the $Loop_i(\vec{X})$ operator by $I_i^{\vec{PV}_i}(\vec{X})$ operator;
(2) replace the $\vec{X}$ list by $C_i^{\vec{PV}_i}(\vec{X})$;
(3) replace statement lists inside loop i $\vec{S}_x$ by $SL_i^{\vec{PV}_i}(\vec{S}_x)$.

For our example, we can rewrite the expression:

$$Loop_i(Loop_{j1}(S1), Loop_{j2}(S2, S3))$$

as

$$I_i^{\vec{PV}_i}(C^{\vec{PV}_i}(I_{j1}^{\vec{PV}_{j1}}(SL_{j1}^{\vec{PV}_{j1}}(S1)), I_{j2}^{\vec{PV}_{j2}}(SL_{j2}^{\vec{PV}_{j2}}(S2, S3))))$$

Upon substituting the definitions and assuming that the statements are independent, we obtain:

$$\left\lfloor II_i \cdot \left(\frac{TC_i^{avg}}{uf_i} - \text{ispip}_i \right) \right\rfloor \odot \max\left(\left\lfloor II_{j1} \cdot \left(\frac{TC_{j1}^{avg}}{uf_{j1}} - \text{ispip}_{j1} \right) \right\rfloor \odot SL_{j1}^{\vec{PV}_{j1}}(S1), \left\lfloor II_{j2} \cdot \left(\frac{TC_{j2}^{avg}}{uf_{j2}} - \text{ispip}_{j2} \right) \right\rfloor \odot SL_{j2}^{\vec{PV}_{j2}}(S2, S3) \right)$$

4.2 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 $\forall i,\ inparallel_i = 1$. We will discuss in the next section a more realistic but compiler-dependent approach to set $inparallel_i$, 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 require the input program to not contain useless operations which may be removed by the HLS toolchain e.g. by dead code elimination, as illustrated in Listing 4.

```
int example(int y, int z){
    int x;
    S0: x = 12 + y; //  dead-code elimination
    S1: x = y + z;
    return x;
}
```

Listing 4. Example of Code Illustrating Dead Code Elimination: As Statement S1 writes to variable x, but the value of x assigned in Statement S0 is never used, the compiler will remove Statement S0.

The restriction can be summarized as:

- The input program is a pure polyhedral program [14], and its analysis (loop trip counts for every loop, all data dependences [12]) 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 also does not contain "useless" operations that may be removed by the compiler.
- We only model DSP and BRAM resources for the considered kernel, ignoring all other resources. We do not model LUT and FF resources, because from experience in the loop-based benchmarks we consider DSP and BRAM resources are the most constraining resources. Moreover, the estimation of LUTs and FFs is more tedious.
- We assume resource (DSP) sharing across different operations executing at the same cycle is not possible.

An important term is SL, 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 [11]. 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 [11] 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.

We can compute the directed acyclic graph made of all statements (i.e., all n-ary operations), connecting all producer and consumer operations, to build the operation graph:

Definition 4.1 (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 $< \{N, V_I, root, V_O\}, E >$ such that $\forall S_i \in L,\ N_{S_i} \in N$; and for every operation with output o and inputs $\vec{i}$ in L $\forall S_i, \forall i_k \in \vec{i}_{S_i}, e_{i_k,o} \in E$, $\forall S_i : (o_{S_i}, \vec{i}_{S_i}) \in L, S_j : (o_{S_j}, \vec{i}_{S_j}) \in L$ with $S_i \neq S_j$ then we have $E_{S_i,S_j} \in E$ iff $o_{Si} \cap \vec{i}_{S_j} \neq \emptyset$. 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 $V_{val} \in V_I$ (resp. V_O) for this input (resp. output) value. If $dim(\vec{i}_{S_i}) = 0$ then an edge e_{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.

Definition 4.2 (Operation Graph critical path). Given OG^L an operation graph for region L. Its critical path OG_{cp} is the longest of all the shortest paths between every pairs $(v_i, v_o) \in < \{V_I, root\}, V_O >$. Its length is noted $\#OG^L_{cp}$.

On the graph on the Figure 1 the critical paths from $A[i]$ to $c, \forall i \in [[0, 7]]$, have the same length.

4.2.1 Latency Lower Bound. We can build a simple a lower bound on the latency of an operation graph:

Theorem 4.3 (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 $LAT^L_{cp} \geq \#OG^L_{cp}$ is a lower bound on the minimal feasible latency to execute L.*

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.

Theorem 4.4 (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 $LO(\#OG^L_{cp}) = \sum_{n \in cp} LO(n)$ the critical path weighted by latency of its operations. The minimal latency of a region L is bounded by*

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

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.

```
#pragma ACCEL unroll factor=uf
L0: for (i = 0; i < N; i++)
    S0: s[i] = 0;
#pragma ACCEL pipeline
L1: for (i = 0; i < M; i++) {
        S1: q[i] = 0;
#pragma ACCEL unroll
    L2: for (j = 0; j < N; j++) {
        S2: s[j]+=r[i]*A[i][j];
        S3: q[i]+=A[i][j]*p[j];
    }
}
```

Listing 5. Bicg code: $s = r \times A; q = A \times p$

For example, in Lst. 5, if we consider the sub-loop body composed of loops L2 (fully unrolled) and the statements $S2$ and $S3$ as straight-line code regions, we can calculate the critical paths for $S2$ and $S3$ as follows: For $S2$, the critical path is given by: $cp_{S2} = \max(L(+) + L(*), N \times (DSP_+ + DSP_*)/R_o)$ with DSP_o the number of DSP for the operation o. For $S3$, the critical path is determined by: $cp_{S3} = \max(L(+) \times \log(N) + L(*), N \times (DSP_+ + DSP_*)/R_o)$, considering the possibility of a tree reduction. In this context, the critical path for the entire sub-loop body is the maximum of these two individual critical paths, expressed as $\max(cp_{S2}, cp_{S3})$.

We now need to integrate loops and enable the composition of latency bounds.

4.2.2 Loop Unrolling: partial unroll. Loop unrolling is an 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_l mod $UF_l \neq 0$ then an epilogue code to execute the remaining $TC_l \mod UF_l$ iterations is needed.

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 4.5 (Minimal latency of a partially unrolled loop with factor UF). *Given a loop l with trip count TC_l and loop body L, and unroll factor $UF \leq TC$. 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:*

$$Lat^{l,S}_{R_{op}} \geq \lfloor TC/UF \rfloor \times Lat^{L^{'}}_{R_{op}}$$

```
#pragma ACCEL parallel UF=4
L1: for (i = 0; i < 8; i++)  {
    S0: a[i] = b[i] * c[i];
}
```

Listing 6. Example before partial unrolling

```
// explicitly unroll with UF=4
L1: for (i = 0; i < 8; i+=4)  {
    S0: a[i] = b[i] * c[i];
    S1: a[i+1] = b[i+1] * c[i+1];
    S2: a[i+2] = b[i+2] * c[i+2];
    S3: a[i+3] = b[i+3] * c[i+3];
}
```

Listing 7. Example after partial unrolling

In Listing 6, the loop body denoted by L is depicted in the rectangle. The loop L1 is subjected to a partial unrolling with a factor of 4, as specified by the directive *#pragma ACCEL parallel UF=4* hence S0 have to be expanded and replicated four times. The explicit unrolling of the loop is illustrated in Listing 7, where S0, S1, S2 and S3 represent the loop body $L^{'}$ of the Theorem B.10. The latency of the program in Listing 6 and 7 (which are equivalent) is greater or equal to $\lfloor \frac{TC}{UF} \rfloor$, in this case 2, multiply by the latency of the loop body $L^{'}$. In this context, $L^{'}$ comprises statements S0, S1, S2, and S3. These four statements are independent of each other and can be executed concurrently. Therefore, the lower bound on latency for $L^{'}$ is equivalent to the latency of the multiplication operation.

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

$$Lat^{l,S}_{R_{op}} \geq \lfloor TC/UF \rfloor * Lat^{L}_{R_{op}}$$

Consider the scenario of loop $L0$ within Listing 5, 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: $\lfloor TC_{L0}/UF \rfloor \times Lat^{S0}_{R_o}$.

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

Theorem 4.7 (Minimal latency of a partially unrolled loop with factor UF for reduction loop with tree reduction). *Given a reduction loop l with trip count TC_l and loop body L, and unroll factor $UF \leq TC$. Given available resources R_{op} and latencies $L(op) \geq 1$. Then the minimal latency of l, if executed in a non-pipelined fashion and the tree reduction is legal is bounded by:*

$$Lat^{l,S}_{R_{op}} \geq \lfloor TC/UF \rfloor \times Lat^{L}_{R_{op}} \times \lfloor \log_2(UF) \rfloor$$

```
L1: for (i = 0; i < 8; i++)  {
    c += a[i];
}
```

Listing 8. Example of code demonstrating a reduction, where a tree reduction technique can be applied, as depicted in Figure 1.

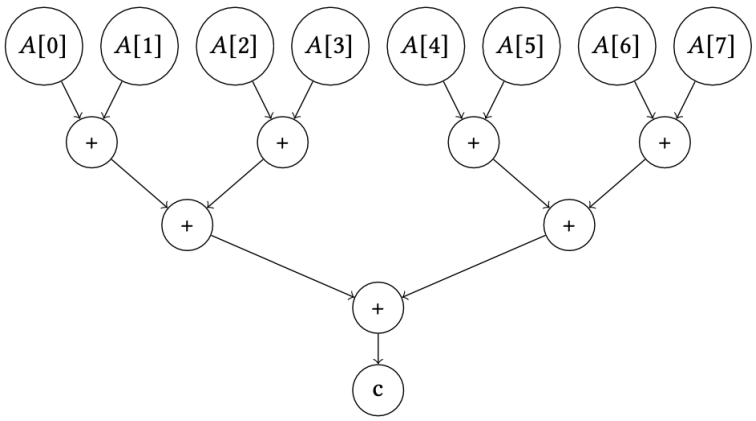


Fig. 1. Illustrating the concurrent execution depicted in Listing 12 through a tree reduction method, accomplished within logarithmic time.

Figure 1 represent the tree reduction of the Listing 12. Due to the reduction, we use a tree reduction in order to increase the parallelism of the reduction, and execute a loop of trip count $n = 8$ in $\log_2(n) = 3$ iterations.

4.2.3 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 pipelined loop and the iteration latency of the operations of the statements during the NLP generation. Let $RecMII$ and $ResMII$ be the recurrence constraints and the resource constraints of the pipelined loop, respectively. We have $II \geq \max(ResMII, RecMII)$. $RecMII = \max_i \lceil \frac{delay(c_i)}{distance(c_i)} \rceil$ with $delay(c_i)$ the total latency in dependency cycle c_i and $distance(c_i)$ the total distance in dependency cycle c_i. We suppose that $ResMII = 1$, as we do not know how the resource will be used by the compiler. Hence, if the loop is a reduction loop then the $II \geq \frac{IL_{reduction}}{1}$ with $IL_{reduction}$ the iteration latency of the operation of reduction. For a kernel like the Listing 9 the $II \geq \lceil \frac{IL_+}{2} \rceil$.

```
1 for (j = 0; j < N; j++)
2     y[j] = y[j-2] + 3;
```

Listing 9. Demonstration of a code snippet showcasing a scenario where a loop pipelined with a dependency of distance 2 results in an initiation interval (II) that satisfies $II \geq \lceil \frac{IL_+}{2} \rceil$.

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

Theorem 4.8 (Minimal latency of a pipelined loop with known II). *Given a loop l with trip count TC_l and loop body L. Given available resources R_{op} and latencies $L(op) \geq 1$. Then the minimal latency of l if executed in a pipelined fashion is bounded by:*

$$Lat^{l,P}_{R_{op}} \geq Lat^{L}_{R_{op}} + II * (TC_l - 1)$$

In Listing 5, loop $L1$ is pipelined. However, loop $L1$ serves is a reduction loop for statement $S2$, meaning that we must await the completion of writing $s[j]$ at iteration i before we can read and write again at iteration $i + 1$. Consequently, the initiation interval (II) is bounded by or greater than the latency (IL_+) of the addition operation, which constitutes the reduction operation.

4.2.4 Loop pipelining and unrolling. A loop l with trip count TC_l can be pipelined and partially unrolled with $UF < TC_l$, in this case there is loop splitting where the trip count of the innermost loop equal to the unroll factor and the trip count of the outermost loop equal to $\frac{TC_l}{UF}$.

Theorem 4.9 (Minimal latency of a pipelined loop with known II and partially unrolled). *Given a loop l with trip count TC_l, partially unrolled by an unroll factor $UF < TC_l$ and a loop body L. 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 pipelined fashion is bounded by:*

$$Lat^{l,P}_{R_{op}} \geq Lat^{L^{'}}_{R_{op}} + II * (\frac{TC_l}{UF} - 1)$$

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

Definition 4.10 (Lower bound on latency of a non-parallel, non-pipelined loop under resources constraints). Given a loop l with trip count TC_l which is neither pipelined nor parallelized, that is, iteration $i + 1$ starts after the full completion of iteration i, for all iterations. Given $Lat^L_{R_{op}}$ the minimal latency of its loop body. Then

$$Lat^l_{R_{op}} \geq TC_l * Lat^L_{R_{op}}$$

4.2.6 Coarse-Grained parallelization. Coarse-grained parallelization is a performance enhancement technique involving the unrolling of a loop which iterates a loop body not fully unrolled i.e., containing at least a pipelined loop or a loop executed sequentially. It is therefore impossible to do a coarse-grained parallelization with a reduction loop because the n sub loop body are dependent on each other.

It follows a bound on the minimal latency of a coarse-grained unrolled loop:

Theorem 4.11 (Minimal latency of coarse-grained unrolled loop). *Given a loop l, which is not a reduction loop, with trip count TC_l, an unroll factor $UF \leq TC_l$ and L the loop body iterated by the loop l with a latency lower bound $Lat^L_{R_{op}}$. 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:*

$$Lat^{l,S}_{R_{op}} \geq \lfloor TC/UF \rfloor \times Lat^{L^{'}}_{R_{op}}$$

4.2.7 Program latency lower bound under resource constraints. We now focus on the latency lower bound of a program, under 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 resources than predicted may be feasible, itself possibly leading to a better latency.

Theorem 4.12. *Given a loop body L, the set of set of statements $\mathcal{S}_{seq}$ non executed in parallel, $\#L^s_{op}$ the number of operations op for the statements s, DSP_{op} the number of resources (DSPs) used for the operation op, MCU^s_{op} the maximal number of computational units the statement s can use in parallel at any given time, and the configuration of pragma $\vec{PV_i}$ for each loop. The minimal number of resource (DSPs) consumed, R^{min}_{used}, 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:*

$$R^{min}_{used} = \sum_{op} \max_{\mathcal{S} \in \mathcal{S}_{seq}} \left(\sum_{s \in \mathcal{S}} \#L^s_{op} \times DSP_{op} \times MCU^s_{op} \right)$$

Given a program and the available resource of DSP DSP_{avail}, if $R^{min}_{used} < DSP_{avail}$ the lower bound is valid and the program does not over-utilize the resources.

In Listing 5, statements S0 and S1 entail no DSP usage as they solely serve for initialization purposes. S2 and S3 undergo unrolling N times and operate independently, enabling parallel execution. Particularly, S3 necessitates a tree reduction since it acts as a reduction for loop $L2$.

Assuming 1 DSP for addition and 2 for multiplication (denoted as DSP_+ and DSP_* respectively), the maximum number of computational units utilized by S2 and S3 for both addition and multiplication is N. Therefore, $MCU^{S2}_+ =$

N, $MCU_*^{S2} = N$, $MCU_+^{S3} = N$, and $MCU_*^{S3} = N$. Furthermore, each statement entails only one addition and one multiplication, hence $\#L_+^{S2} = 1$, $\#L_*^{S2} = 1$, $\#L_+^{S3} = 1$, and $\#L_*^{S3} = 1$.

Consequently, the minimum number of DSPs utilized by Listing 5 is $N \times (1 + 2) \times 2$.

4.2.8 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 4.13 (Lower bound of the memory transfer latency for an array). *Given a loop body L, the set of array $\mathcal{A}$, an array $a \in \mathcal{A}$, and LAT_a^{mem} the latency to transfer the array a from off-chip to on-chip (inputs) and from on-chip to off-chip (outputs). $\forall a \in \mathcal{A}, LAT_a^{mem} \geq (\mathbb{1}_{a \in V_O^L} + \mathbb{1}_{a \in V_I^L}) \times footprint_a / max\_burst\_size$. With $\mathbb{1}_{cond} = 1$ if cond = true else 0.*

Within Listing 5, the matrix A is solely read and possesses a footprint of $N \times M \times 32$ bits. Given the FPGA's limitation to process a maximum of 512 bits per cycle, the latency required for transferring A amounts to $\frac{N \times M}{16}$ cycles.

Theorem 4.14 (Lower bound of the memory transfer latency). *Given a loop body L, the set of array $\mathcal{A}$, the number of cycles to transfer the array a is bounded by $\max_{a \in \mathcal{A}}(\mathbb{1}_{a \in V_O^L} + \mathbb{1}_{a \in V_I^L}) \times footprint_a / max\_burst\_size$.*

In Listing 5, when transferring all arrays before any computation, we can concurrently transfer each array (assuming they are in different DRAM banks). The primary bottleneck arises from the transfer of array A, which incurs a latency of $\frac{N \times M}{16}$ cycles. Upon program completion, arrays s and q need to be transferred back to DRAM, with a minimal latency of $\max(\frac{N}{16}, \frac{M}{16})$ cycles each. Thus, the total communication latency amounts to $\frac{N \times M}{16} + \max(\frac{N}{16}, \frac{M}{16})$ cycles.

4.3 Summary

By composing all the theorems, this allows us to end up with the final latency lower bound of the program which is presented in theorems B.20 for the computation and B.21 for the computation and communication.

Theorem 4.15 (Computation Latency Lower bound of a Program). *Given available resource DSP_{avail}, the properties vector $\vec{PV}_i$ for each loop and a program which contains a loop body L. The properties vector allows to give all the information concerning the trip counts and the II of the pipelined loops and to decompose the loop body L with a set of loops $\mathcal{L}_L^{non\ reduction}$ potentially coarse-grained unrolled with $\forall l \in \mathcal{L}_L, UF_l$ and a set of reduction loops executed sequentially $\mathcal{L}_L^{reduction}$ which iterates a loop body L_{pip}. By recursion the loop body L_{pip} contains a pipelined loop l_{pip} which iterate a loop body L_{fg} fully unrolled. The loop body L_{fg} contains operations which can be done in parallel with a latency $Lat_{R_{op}}^{L_{par}}$ and operations which are reduction originally iterated by the loops $\mathcal{L}_{L_{fg}}^{reduction}$ with a latency $Lat_{L_{seq}}$.*

The computation latency lower bound of L, which respected $DSP_{ued}^{min} \leq DSP_{avail}$, executed with tree reduction is:

$$Lat_{R_{op}}^{L} \geq \prod_{l \in \mathcal{L}_L^{par}} \frac{TC_l}{UF_l} \times \prod_{l \in \mathcal{L}_L^{reduction}} TC_l \times Lat_{R_{op}}^{L_{pip}}$$

with

$Lat_{R_{op}}^{L_{pip}} = (Lat_{R_{op}}^{L_{fg}} + II \times (\frac{TC_{l_{pip}}}{l_{pip}\_UF} - 1))$ *and* $Lat_{R_{op}}^{L_{fg}} = Lat_{L_{par}} + Lat_{L_{seq}} \times \prod_{l \in \mathcal{L}_{L_{fg}}^{reduction}} \frac{TC_l}{UF_l} \times \log_2(UF_l)$.

In Listing 5, let's denote the loop body with loop $L0$ and statement $S0$ as $LB0$, and the one with loops $L1$ and $L2$, along with statements $S1$, $S2$, and $S3$, as $LB1$.

For $LB0$, where the loop is unrolled with a factor uf and no dependency exists between loop iterations, the latency is bounded by $\frac{N}{uf} \times 1$, assuming the initialization takes at least one cycle.

Regarding $LB1$, loop $L2$ is fully unrolled, with no dependencies between iterations for statement $S2$. As there is no dependency between $S2$ and $S3$, they can be executed in parallel, and the latency of the unrolled part equals the critical path, which is the maximum between the unrolled parts of $S2$ and $S3$ in this scenario. However, for statement $S3$, despite the absence of dependencies among multiplication operations enabling parallel execution, a tree reduction is involved. Given the pipelined nature of loop $L1$, functioning as a reduction for statement $S2$, we observe $II > IL_+$, where IL_+ denotes the latency of the reduction operation. Consequently, the latency of $LB1$ is bounded by $(\max(IL_+ + IL_*, IL_+ \log_2(N) \times IL_+) + IL_+ \times (M-1))$.

Theorem 4.16 (Latency lower bound of a program optimized with Merlin pragmas). *Given available resource DSP_{avail} and a program which contains a loop body L with a computation latency $Lat_L^{computation}$ and a communication latency $Lat_L^{communication}$.*

The lower bound for L which respected $DSP_{ued}^{min} \leq DSP_{avail}$ and where the computation and communication can not be overlap is:

$$Lat_L = Lat_L^{computation} + Lat_L^{communication}$$

5 NON-LINEAR FORMULATION FOR PRAGMA INSERTION

We now present the complete set of constraints and variables employed to encode the latency and resource model as a non-linear program. This section presents a modeling of section 4 in the practical case.

Let $\mathcal{L}$ be the set of loops, $\mathcal{A}$ the set of arrays, $\mathcal{S}$ the set of statements and O_s 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, $O_{s_{par}}$ and the reduction operations, i.e., associative/commutative operators to reduce one or more values into a single value, leading to loop-carried dependencies, $O_{s_{red}}$.

Let $\mathcal{P}$ be the set of different possible pipeline configurations. Let $\forall p \in \mathcal{P}$ define $\mathcal{L}_{pip}^p$ the set of loops pipelined and $\forall l \in \mathcal{L}_{pip}^p$, $\mathcal{L}_{under\_pip_l}^p$ the set of loops under a loop pipelined and $\mathcal{L}_{above\_pip_l}^p$ the set of loops above the loop pipelined l. Let $\forall s \in \mathcal{S}$ define the set of nested loops which iterate the statement s, $\mathcal{L}_s$. $\forall a \in \mathcal{A}$ and for d a dimension of the array a, let C_{a_d} be the set of loops which iterates 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 II_s be the II of the loop pipelined 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 [29] 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).

Table 4, summarize the sets, variable and constants we use.

Set	Description
$\mathcal{L}$	the set of loops
$\mathcal{A}$	the set of arrays
$\mathcal{S}$	the set of statements
O_s	the list of operations of the statements s
$O_{s_{par}}$	the operation which can be done in parallel, i.e., does not have any loop-carried dependence
$\mathcal{P}$	the set of different possible pipeline configurations
$\mathcal{L}^p_{pip}$	the set of loops pipelined
$\mathcal{L}^p_{under\_pip_l}$	the set of loops under a loop pipelined
$\mathcal{L}^p_{above\_pip_l}$	the set of loops above the loop pipelined l
$\mathcal{L}_s$	the set of nested loops which iterate the statement s
C_{a_d}	the set of loops which iterates the array a at the dimension d
d_l	the maximum dependency distance of the loop l
Variable	**Description**
$loop_l\_UF$	Unroll factor of the loop l
$loop_l\_tile$	TC of the innermost loop after strip-mining of the loop l
$loop_l\_pip$	Boolean to know if the loop l is pipelined
$loop_l\_cache\_array\_a$	Boolean to know the the array a is transferred on-chip before the loop l
Constant	**Description**
TC_l	Trip Count of the loop l
II_l	II of the loop l
IL_{par}	Iteration Latency of the operations whitout dependencies of the statement s
IL_{red}	Iteration Latency of the operations whit 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
$MAX\_PARTITIONING$	Maximum array partitioning defined by the user or the DSE (cf. Section 6)
$footprint\_array_a\_loop_l$	Footprint of the array a if transferred on-chip before the loop l

Table 4. Description of the sets, variables, and constants utilized in the formulation of the Nonlinear Problem (NLP) aimed at modeling latency and resource consumption of a design

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 $loop_l\_UF == TC_l$ (Eq. 15). Considering this constraint, it is important to note that for each statement, only one of the loops that iterate the statement can have a pragma pipeline (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 $\{uf + 1, \ldots, n\}$ 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 $loop_l\_UF == 1$ (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

$$\forall l \in \mathcal{L}, 1 \leq loop_l\_UF \leq TC_l \tag{1}$$

$$\forall l \in \mathcal{L}, 1 \leq loop_l\_tile \leq TC_l \tag{2}$$

$$\forall l \in \mathcal{L}, loop_l\_pip \in \{0, 1\} \tag{3}$$

$$\forall l \in \mathcal{L}, \forall a \in \mathcal{A}, loop_l\_cache\_array_a \in \{0, 1\} \tag{4}$$

$$\forall s \in \mathcal{S}, \sum_{l \in \mathcal{L}_s} loop_l\_pip \leq 1 \tag{5}$$

$$\forall l \in \mathcal{L}, loop_l\_UF \% TC_l == 0 \tag{6}$$

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

$$\forall l \in \mathcal{L}, \text{ if } dd_l > 1, loop_l\_UF \leq d_l \tag{8}$$

$$\forall l \in \mathcal{L}, \forall l' \in \mathcal{L}_{above\_l}, loop_l\_pip * loop_{l'}\_UF <= 1 \tag{9}$$

$$\forall s \in \mathcal{S}, \prod_{l \in L_s} loop_l\_UF \leq MAX\_PARTITIONING \tag{10}$$

$$DSPs\_used_{optimistic} = \sum_{op \in \{+,-,*,/\}} \max_{s \in \mathcal{S}}(DSP_{s_{op}}/II_s) \leq DSP_{available} \tag{11}$$

$$\begin{cases} \sum_{a \in \mathcal{A}} \sum_{l \in \mathcal{L}} loop_l\_cache\_array_a \\ \times footprint\_array_a\_loop_l \leq Mem \end{cases} \tag{12}$$

$$\begin{cases} \forall a \in \mathcal{A}, \forall (d, d') \in \mathbb{N}^2 \text{ with } d \neq d', \forall l \in C_{a_d}, \forall l' \in C_{a_{d'}}, \\ loop_l\_UF \times loop_{l'}\_UF \leq MAX\_PARTITIONING \end{cases} \tag{13}$$

$$\begin{cases} \forall p \in \mathcal{P}, \forall lp \in \mathcal{L}^{p}_{pip}, \forall lbp \in \mathcal{L}_{under\_pip_{lp}} \forall a \in \mathcal{A}, \\ loop_{lbp}\_cache\_array_{a} == 0 \end{cases} \tag{14}$$

$$\begin{cases} \forall p \in \mathcal{P}, \forall lp \in \mathcal{L}^{p}_{pip}, \forall lbp \in \mathcal{L}_{under\_pip_{lp}}, \\ loop_{lp} \times loop_{lbp}\_UF == loop_{lp} \times TC_{lbp} \end{cases} \tag{15}$$

5.4 Objective function

Lastly, we need to define the objective function (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: $TC_{ap} \times (IL + II \times (\frac{TC}{UF} - 1))$, 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.

$$\begin{cases} TC_{ap} = \prod_{l \in \mathcal{L}^{par}_{above\_pip}} \frac{TC_l}{loop_l\_UF} \times \prod_{l \in \mathcal{L}^{red}_{above\_pip}} TC_l \\ IL = IL_{par} + IL_{seq} \times \prod_{l \in \mathcal{L}^{red}_{under\_pip}} \frac{TC_l}{loop_l\_UF} \times \log_2(loop_l\_UF) \\ L_{mem} = \sum_{l \in \mathcal{L}} \max_{a \in \mathcal{A}} (loop_l\_cache_a \times footprint_a\_loop_l) \\ obj\_func = TC_{ap} \times (IL + II \times (\frac{TC_{lp}}{loop_{lp}\_UF} - 1)) + L_{mem} \end{cases}$$

5.5 Example

```
Loop0: for(i=0; i<2100; i++)
    S0:y[i] = 0;
Loop1: for(i=0; i<1900; i++) {
    S1:t[i] = 0;
    Loop2: for(j=0; j<2100; j++)
```

```
''

        S2:t[i]+=A[i][j]*x[j];
    Loop3: for(j=0; j<2100; j++)
        S3:y[j]+=A[i][j]*t[i];
}
```

Listing 10. AtAx code for Large problem size: $t = A * x$; $y = A * t$

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 operation 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_S1 and IL_S2. Between S2 and S3 the distance is log(N) so the final equation will be IL_S1 + IL_S2 * log(N) + IL_S3 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 $for(j = 2; j < N; j++)y[j] = y[j-2] + 3;$, a straightforward approach to handling this type of dependency is to impose a constraint such as $loop_l\_UF \leq 2$, 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 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.

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 [27]. The complexities and sizes of the problems are detailed in Table 8 located in the Appendix. In addition we add a kernel of Convolution Neural Network (CNN). A single-precision floating point is

Data: *kernel* // without Pragma
Data: *Space_Array_Partitioning* // e.g., {∞, 2048, 1024, 512, 256, 128, 64, 32, 16, 8, 1}
Data: *timeout_HLS*, *timeout_NLP*
Result: *kernel* // with Merlin Pragma
$nlp\_file \leftarrow generate\_nlp\_file(kernel)$, $min\_lat \leftarrow \infty$;
for $max\_array\_partitioning \in Space\_Array\_Partitioning$ **do**
  for $parallelism \in \{coarse + fine, fine\}$ **do**
    $current\_nlp\_file \leftarrow$ change_max_array_partitioning($copy(nlp\_file)$, $max\_array\_partitioning$);
    if $parallelism == fine$ **then**
      add_constraint_only_fine_grained_parallelism($current\_nlp\_file$);
    end
    $pragma\_configuration, lower\_bound \leftarrow SOLVER(current\_nlp\_file, timeout\_NLP)$;
    if $lower\_bound < min\_lat$ **then**
      $current\_kernel \leftarrow$ introduce_pragma($copy(kernel)$, $pragma\_configuration$);
      $hls\_lat, valid \leftarrow MERLIN(kernel, timeout\_HLS)$;
      if *valid* **then** // no over-utilization
        $min\_lat \leftarrow min(min\_lat, hls\_lat)$;
      end
    end
  end
end

Algorithm 1: NLP-DSE

utilized as the default data type in computations to compare to AutoDSE. Computations operate on medium and large datasets from PolyBench/C [27] in order to have kernels with large footprint and have a large enough space to explore. Selecting medium and large problem sizes is crucial to accurately reflect the complexities encountered in various fields such as scientific simulations, data analytics, and artificial intelligence. These sizes pose challenges that mirror the practical limitations of memory transfer, where efficiently managing data movement becomes paramount due to its potential to bottleneck performance. In such contexts, the careful orchestration of memory transfers is essential to ensure optimal resource utilization and prevent computational inefficiencies. Additionally, the scale of these problems often exceeds on-chip memory capacity, necessitating strategies for effectively handling data footprints that surpass available memory, further emphasizing the need for meticulous memory management and optimization techniques. The problem size and loop order of CNN are J,I=256, P,Q=5 H,W=224. A description of each benchmark can be found in Tables 8 and 5. The *ludcmp*, *deriche* and *nusinnov* kernels are not present as PolyOpt-HLS [28] 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.

A double-precision floating point is utilized as the default data type in computations to compare to HARP. We chose to use the problem size used by HARP in order to reuse their model. The problem size and kernel use by HARP can be found on the Table 9.

We evaluate designs with AMD/Xilinx Merlin [43]. 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 [29]. We modified and extended for our work. Employing the AMPL description language to solve the NLP problems, we ran the commercial BARON solver [31, 39] 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

7.2.1 AutoDSE. We compare our method with AutoDSE [38], 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 does not 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 5 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 5). For our method we take the same parameters, and we add a timeout for BARON of 30 minutes. The space given as input to NLP-DSE is $\{\infty, 2048, 1024, 512, 256, 128, 64, 32, 16, 8, 1\}$.

7.2.2 HARP. The evaluation vs. HARP is done with the same parameters as the evaluation vs. AutoDSE. We change the space given as input due to the small problem size and we choose $\{\infty, 1024, 750, 512, 256, 128, 64, 32, 16, 8, 1\}$.

We run HARP for one hour in order to have a similar DSE time as NLP-DSE . This enables the exploration of an average of 75,000 distinct pragma configurations for each kernel. HARP's DSE method navigates the space by iteratively adjusting the pragma in a bottom-up manner. It synthesizes the top 10 designs discovered by the DSE, employing a timeout of 3 hours for the HLS compiler, similar to the approach used in the NLP-DSE framework.

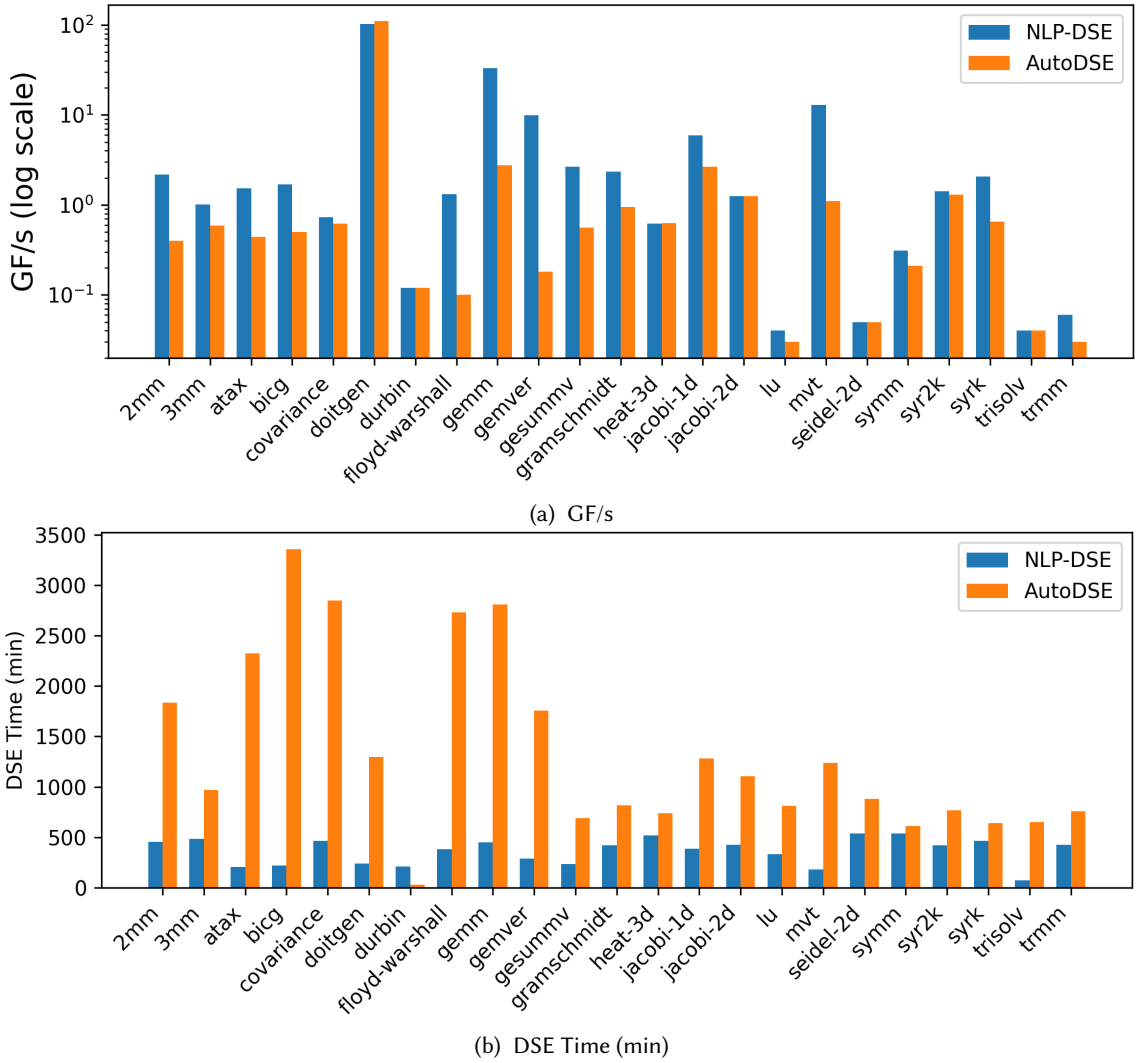
7.3 Comparison with AutoDSE

Figures 2 and 3 show the comparison with AutoDSE for Large and Medium problem size respectively. Table 5 shows the details of 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.



(a) GF/s

(b) DSE Time (min)

Fig. 2. Comparison between the throughput (GF/s) and Design Space Exploration (DSE) time (min) of NLP-DSE and AutoDSE for large problem sizes in Polybench.

					FS	NLP-DSE				AutoDSE					Perf. Improvement	
Kernel	**NL**	**ND**	**S**	**Space S**	**GF/s**	**GF/s**	**T**	**DE**	**DT**	**GF/s**	**T**	**DE**	**DT**	**ER**	**T**	**GF/s**
covariance	7	34	M	1.80E+11	0.08	0.75	336	21	8	0.28	645	161	15	115	1.92x	2.64x
covariance	7	34	L	1.92E+13	0.39	0.73	466	21	9	0.62	2,849	209	68	118	6.11x	1.16x
2mm	6	13	M	1.37E+10	13.19	117.48	70	18	0	0.41	1,870	101	37	49	26.71x	288.88x
2mm	6	13	L	1.15E+12	0.57	2.17	456	17	3	0.40	1,835	291	38	240	4.02x	5.41x
3mm	9	19	M	1.20E+15	13.86	138.73	242	18	0	0.39	698	82	15	57	2.88x	354.12x
3mm	9	19	L	6.18E+17	0.18	1.01	486	19	3	0.59	968	112	18	81	1.99x	1.71x
atAx	4	12	M	1.40E+05	1.96	1.96	194	10	1	1.98	1,653	175	13	136	8.52x	0.99x
atAx	4	12	L	1.60E+07	0.47	1.52	205	11	2	0.44	2,325	166	30	106	11.34x	3.46x
bicg	3	10	M	1.90E+04	0.99	0.99	248	12	1	0.98	729	65	2	28	2.94x	1.01x
bicg	3	10	L	4.44E+05	1.68	1.68	218	12	1	0.50	3,355	236	42	176	15.39x	3.38x
cnn	6	2	-	6.43E+06	0.39	97.99	213	16	1	97.99	1,292	28	19	480	6.06x	1.00x
doitgen	5	30	M	8.64E+06	19.75	19.75	193	13	0	18.95	819	296	14	248	4.24x	1.04x
doitgen	5	30	L	3.63E+07	0.08	102.62	241	20	1	110.66	1,299	222	24	169	5.39x	0.93x
durbin	4	55	M	1.08E+02	0.01	0.20	193	7	4	0.12	134	25	0	23	0.69x	1.65x
durbin	4	55	L	9.00E+00	0.12	0.12	212	3	1	0.12	31	7	0	5	0.15x	1.00x
gemm	4	6	M	2.30E+06	105.18	105.18	185	21	1	68.91	1,345	86	27	34	7.27x	1.53x
gemm	4	6	L	1.47E+07	32.98	32.98	450	18	7	2.77	2,810	188	47	133	6.24x	11.89x
gemver	7	13	M	7.72E+11	0.78	9.45	218	21	4	2.99	847	65	5	28	3.89x	3.16x
gemver	7	13	L	1.28E+13	9.94	9.94	290	21	7	0.18	1,756	221	206	10	6.06x	54.69x
gesummv	2	17	M	6.12E+02	1.97	1.97	220	14	3	1.97	836	80	8	47	3.80x	1.00x
gesummv	2	17	L	6.33E+03	1.82	2.64	236	18	1	0.56	692	94	29	60	2.93x	4.73x
gramschmidt	6	34	M	1.75E+07	1.58	1.58	364	7	3	0.44	934	109	92	8	2.57x	3.56x
gramschmidt	6	34	L	1.22E+08	2.34	2.34	420	6	4	0.95	819	265	11	239	1.95x	2.47x
lu	5	16	M	2.28E+03	0.03	0.03	614	19	11	0.04	849	193	1	159	1.38x	0.98x
lu	5	16	L	3.99E+03	0.03	0.04	335	9	2	0.03	812	258	5	219	2.42x	1.03x
mvt	4	6	M	1.38E+07	7.77	7.77	212	17	1	7.77	893	166	6	106	4.21x	1.00x
mvt	4	6	L	7.41E+07	12.90	12.90	181	20	3	1.10	1,240	249	20	189	6.85x	11.78x
symm	3	33	M	2.31E+04	0.04	0.20	63	5	0	0.20	691	142	35	89	10.97x	1.00x
symm	3	33	L	6.64E+04	0.21	0.31	540	8	5	0.21	612	731	0	692	1.13x	1.52x
syr2k	4	6	M	2.32E+04	0.07	1.74	224	16	2	1.20	685	230	17	203	3.06x	1.45x
syr2k	4	6	L	6.67E+04	1.30	1.42	420	15	7	1.30	768	293	21	262	1.83x	1.09x
syrk	4	6	M	2.32E+04	0.49	1.32	224	16	2	0.61	631	280	4	264	2.82x	2.15x
syrk	4	6	L	6.67E+04	0.94	2.07	466	17	7	0.65	643	410	0	398	1.38x	3.16x
trisolv	2	13	M	3.60E+02	0.03	0.03	69	12	0	0.04	694	69	33	23	10.06x	0.98x
trisolv	2	13	L	6.30E+02	0.04	0.04	75	18	0	0.04	651	127	2	98	8.68x	0.99x
trmm	3	8	M	2.31E+04	0.01	0.05	20	16	0	0.04	630	401	4	367	31.50x	1.29x
trmm	3	8	L	6.64E+04	0.02	0.06	425	17	2	0.03	760	167	159	4	1.79x	1.79x
floyd-warshall	3	21	M	8.65E+04	0.17	0.61	246	17	3	0.10	1,605	60	22	29	6.52x	6.15x
floyd-warshall	3	21	L	3.29E+06	0.17	1.31	381	20	6	0.10	2,728	150	71	77	7.16x	13.09x
heat-3d	7	42	M	3.04E+07	0.23	3.75	402	17	7	3.75	928	75	33	35	2.31x	1.00x
heat-3d	7	42	L	4.37E+07	0.13	0.62	520	13	6	0.63	740	109	70	35	1.42x	0.99x
jacobi-1d	3	14	M	1.48E+03	11.43	11.43	55	4	0	11.53	948	126	3	74	17.24x	0.99x
jacobi-1d	3	14	L	4.00E+04	5.95	5.95	386	9	6	2.65	1,283	173	21	123	3.32x	2.25x
jacobi-2d	5	22	M	8.07E+06	0.20	3.32	379	12	5	3.32	674	158	26	98	1.78x	1.00x
jacobi-2d	5	22	L	1.14E+07	0.26	1.25	427	9	6	1.25	1,106	231	39	171	2.59x	1.00x
seidel-2d	3	27	M	4.26E+03	0.05	0.05	365	5	1	0.05	796	103	27	68	2.18x	1.01x
seidel-2d	3	27	L	1.77E+05	0.05	0.05	540	13	7	0.05	880	91	33	48	1.63x	1.00x
Average					5.38	15.11	296			7.44	1,123				5.69x	17.24x
Geo. Mean					0.51	1.54	247			0.65	916				3.70x	2.38x

Table 5. Comparison of DSE time and Throughput for NLP-DSE , NLP-DSE-FS, and AutoDSE across Polybench kernels for different problem sizes

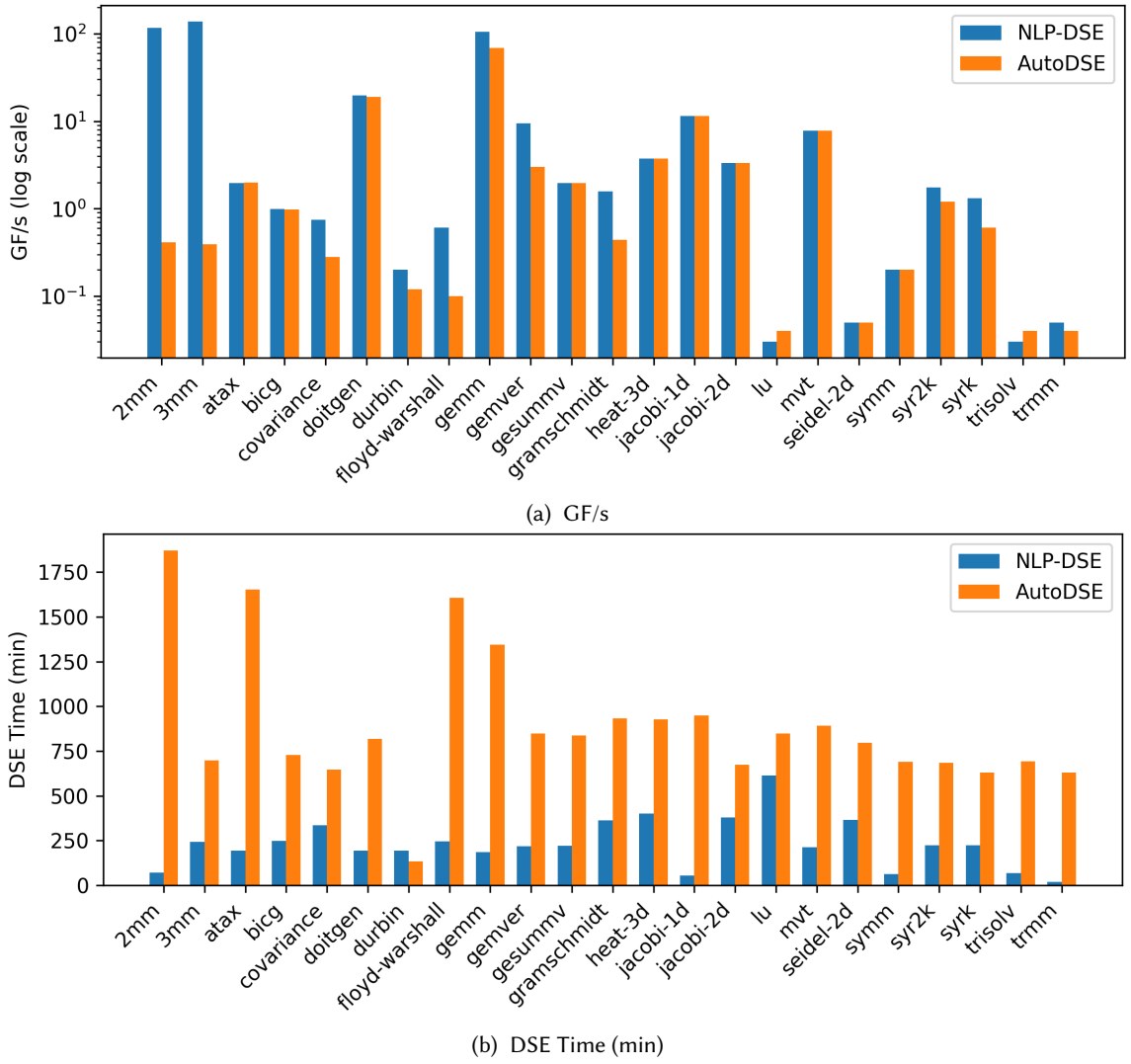


Fig. 3. Comparison between the throughput (GF/s) and Design Space Exploration (DSE) time (min) of NLP-DSE and AutoDSE for medium problem sizes in Polybench.

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 examples of *2mm* and *3mm*, 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. 10), 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 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 Comparison with HARP

Utilizing the PolyBench problem size of HARP allows for direct reusability of the model, facilitating comparison and benchmarking against the framework HARP. This also allows for the utilization of data that will enable achieving the best results with HARP.

Evaluating on other problem sizes would have required the creation of a database to at least fine-tune the model with the kernel and problem size in question.

Figure 4 shows the comparison with HARP for Small and Medium problem size. Table 9 in Appendix, shows the details of the comparison.

The throughput is 1.45x higher on average (1.20x for geo-mean) for the kernel evaluated in similar DSE time. For 20/23 kernels the method identifies a design with a throughput similar to (+/- 10%), or better than, HARP.

We note a variation in the enhancement of performance compared to the evaluation in Section 7.3 vs. AutoDSE, stemming from various factors. Primarily, the breadth of the exploration space plays a significant role. HARP has the capacity to traverse an average of 150,000 designs, enabling it to nearly exhaustively explore the entire space. Additionally, HARP is trained and/or fine-tuned with precise knowledge of the kernel and problem size, granting it deep insight into scenarios where pragmas are not applied and have enough training on these specific kernel to estimate the latency. This confers an advantage over AutoDSE, which treats the compiler as a black box.

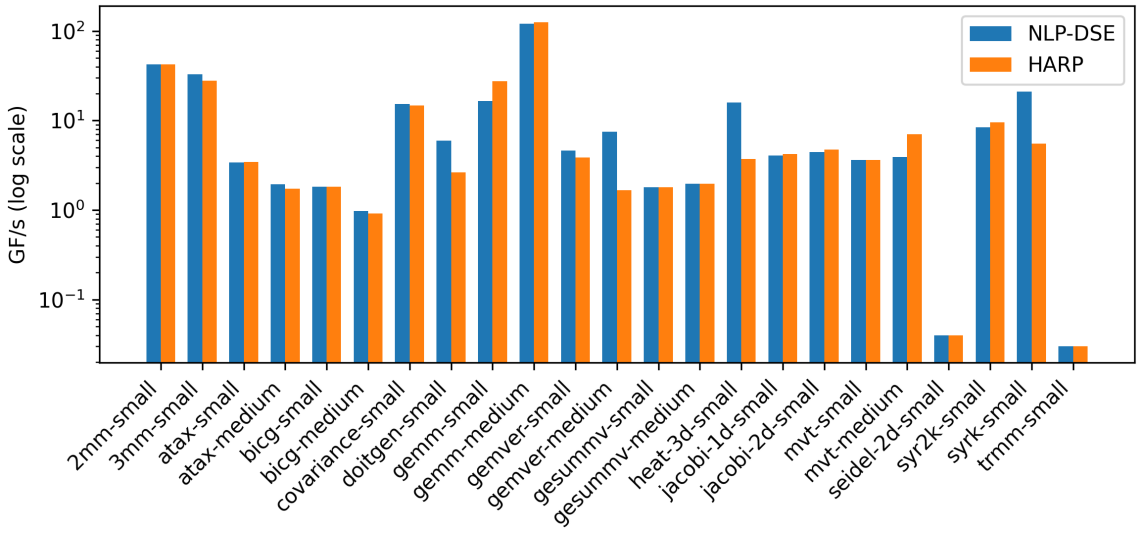


Fig. 4. Comparison between the throughput (GF/s) of NLP-DSE and HARP for small and medium problem sizes in Polybench.

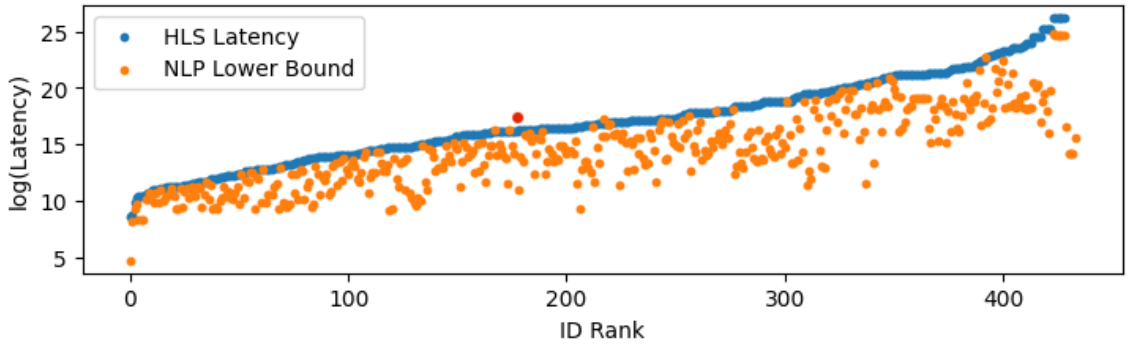
In the case of the medium-sized mvt kernel, a notable slowdown occurs because Merlin is transferring the array A twice. This array comprises 159,900 double-precision floats. The kernel is already constrained by memory bandwidth when transferring A once. The duration for a single transfer of A is 20,000 cycles, utilizing a bitwidth of 512 (maximum), amounting to a total of 40,000 cycles. Consequently, the kernel's overall latency is 40,726 cycles. However, HARP successfully identified a configuration that enables Merlin to transfer A only once.

With Gemver, we can attain a speedup by leveraging our capability to explore the entire space within a single optimization problem. The space of Gemver with medium size encompasses over 10^{11} designs, making it impractical to thoroughly explore, even with HARP's estimation per design hovering around the millisecond range.

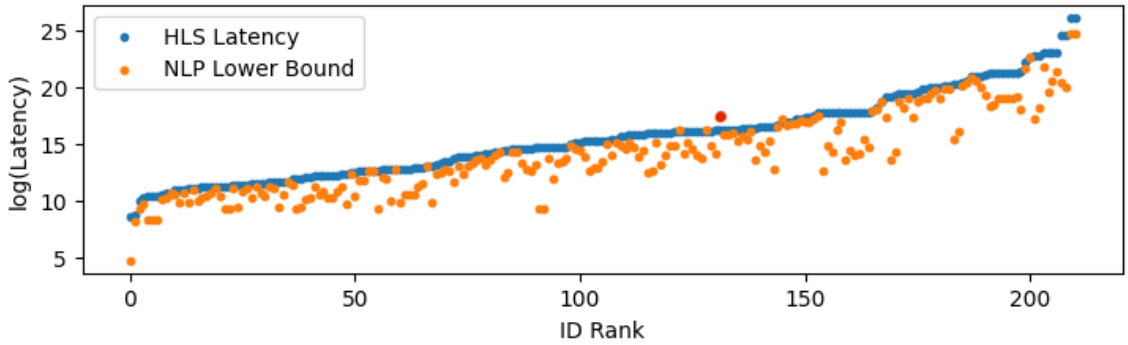
7.5 Accuracy

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 5a and 5b 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 5b 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. Coarse-grained pragmas are typically not applied to kernels that do not have an outermost reduction loop and thus can theoretically have coarse-grained parallelization, which is present in most linear algebra kernels such as 2mm, 3mm, gemver, etc. 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 5b 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. Internally, Merlin transforms

the size of the arrays according to the program's unroll factors and in certain cases does not allow transfers with a bitwidth of 512 bits.



(a) Comparison of the latency between the design reported in the HLS report and the lower bound estimate provided by the nonlinear problem for all explored designs



(b) Comparison of the latency between the design reported in the HLS report and the lower bound estimate provided by the nonlinear problem specifically for cases where pragmas were applied

Fig. 5. Comparison of the latency between the design reported in the HLS report and the lower bound estimate provided by the nonlinear problem for all explored designs and specifically for cases where pragmas were applied.

We observe in Fig. 5a and Fig. 5b one configuration where the lower bound property is not maintained (shown in red). This corresponds to a configuration of the Heat-3d kernel, where the pragma **loop_flatten** has been applied automatically, which changes the program structure. Overall unless Vitis applies loop_flatten automatically, which we do not model, our estimate is a lower bound for the cases evaluated. Our model can easily implement the automatic flattened loop optimization: We must multiply the TC of the loop pipeline by the TC of all the perfectly nesteed loops above pipeline loop (and remove them in the first products). Because this optimization is rarely applied, we prioritize having a tight lower bound.

Additionally, we evaluate the number of DSE steps needed to achieve the design with the best Quality of Results (QoR) of our DSE and the number of syntheses required before terminating the DSE due to finding a lower bound (LB) greater than the latency of an already synthesized kernel, sorting by latency estimation provided by the NLP in ascending order.

Kernel	To find the best QoR		To find a LB > than HLS result	
Problem Size	Large	Medium	Large	Medium
2mm	4	6	13	9
3mm	7	2	11	7
atAx	6	2	18	14
bicg	2	2	16	18
covariance	12	8	16	18
doitgen	1	0	4	10
durbin	0	16	21	20
fdtd-2d	12	1	18	8
floyd-warshall	8	16	20	20
gemm	7	4	13	7
gemver	5	9	12	11
gesummv	0	4	14	16
gramschmidt	8	14	10	16
heat-3d	17	20	19	15
jacobi-1d	16	0	18	10
jacobi-2d	6	18	16	20
lu	18	0	20	20
mvt	1	0	4	6
seidel-2d	17	10	20	20
symm	12	10	16	21
syr2k	13	18	16	20
syrk	17	17	18	20
trisolv	0	0	17	20
trmm	16	4	20	17

Table 6. The count of designs evaluated to identify the HLS design yielding the optimal Quality of Results (QoR), and the count at which the Design Space Exploration (DSE) ceases upon discovering a lower bound (LB) surpassing the latency of a design already synthesized with the HLS compiler

The results are presented in Table 6. On average, it takes 8 steps of the DSE to discover the design with the best QoR and 15 steps to terminate the DSE. We can observe that for some kernel we find the design with the best QoR at the first iteration of the DSE (which correspond to first shoot method in Table 5) but the DSE need more stop to stop the guarantee that we cannot obtain better latency, which implies that the lower bound is not perfectly tight.

7.6 Scalability

To mitigate prolonged solving times for specific kernels and problem sizes, we have implemented a 30-minute timeout constraint for the AMPL BARON solver. While this timeout does not guarantee achieving optimality, it ensures that the solver provides the best solution it has found within the time limit. In Table 7, 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%.

Size	ND T/O	ND NT/O	Avg Time	Avg Time NT/O
Medium	7	469	55s	29s
Large	119	361	479s	43s
All	126	830	268s	35s

Table 7. Study of the scalability of the NLP solver across various sizes of Polybench and CNN. Comparison of the number of designs that timeout (ND T/O), the number of designs that do not timeout (ND NT/O), the average time to solve the problem (Avg Time), and the average time to solve the problem for non-timeout designs (Avg Time NT/O).

8 EXAMPLES

In this section, we illustrate the significance of our method by contrasting it with AutoDSE and highlighting the advantages of our Design Space Exploration (DSE) approach. Our method excels in addressing domain-specific constraints, providing superior convergence in complex scenarios compared to AutoDSE. Furthermore, our DSE demonstrates adaptability and efficiency, proving to be robust in handling intricate design spaces, offering a more versatile and high-performing solution.

8.1 2mm Medium

```
#pragma ACCEL PIPELINE PIPE_L0
#pragma ACCEL TILE FACTOR=TILE_L0
#pragma ACCEL PARALLEL FACTOR=PARA_L0
    Loop0: for (i1 = 0; i1 < 180; i1++) {
#pragma ACCEL PIPELINE PIPE_L2
#pragma ACCEL TILE FACTOR=TILE_L2
#pragma ACCEL PARALLEL FACTOR=PARA_L2
      Loop1: for (j1 = 0; j1 < 190; j1++) {
        S0: tmp[i1][j1] = 0.0;
#pragma ACCEL PARALLEL FACTOR=PARA_L4
        Loop2: for (k1 = 0; k1 < 210; ++k1) {
          S1: tmp[i1][j1] += alpha * A[i1][k1] * B[k1][j1];
        }
      }
    }
```

```
#pragma ACCEL PIPELINE PIPE_L1
#pragma ACCEL TILE FACTOR=TILE_L1
#pragma ACCEL PARALLEL FACTOR=PARA_L1
    Loop3: for (i2 = 0; i2 < 180; i2++) {
#pragma ACCEL PIPELINE PIPE_L3
#pragma ACCEL TILE FACTOR=TILE_L3
#pragma ACCEL PARALLEL FACTOR=PARA_L3
      Loop4: for (j2 = 0; j2 < 220; j2++) {
        S2: D[i2][j2] *= beta;
#pragma ACCEL PARALLEL FACTOR=PARA_L5
        Loop5: for (k2 = 0; k2 < 190; ++k2) {
          S3: D[i2][j2] += tmp[i2][k2] * C[k2][j2];
        }
      }
    }
  }
}
```

Listing 11. Implementing the 2mm code with pragma directives for pipelining, tiling, and parallelization for each loop: $D = \alpha \times A \times B \times C + \beta \times D$.

2mm serves as a linear algebra kernel, acting as a surrogate for transformer inference, such as Bert. The code snippet in Listing 11 illustrates the Medium-sized configuration with potential pragma options. The PIPE pragma can be either flattened or off (default), while PARA and TILE can be any divisor of the loop trip count, defaulting to 1.

8.1.1 AutoDSE. The optimal design identified by AutoDSE involves: PARA_L5 = 220, PIPE_L3 = flatten, PARA_L4 = 4 with all other parameters set to 1 or off. However, AutoDSE faces challenges in achieving a high Quality of Results (QoR) for 2mm due to two primary reasons:

For 2mm AutoDSE does not allow you to find a design with good QoR for two main reasons:

- three out of four of workers initially over-utilize parallelism by flattening PIPE_L0 and \ or PIPE_L1 (and hence unroll the innermost loops), causing timeouts in current High-Level Synthesis (HLS) tools. Even without considering timeouts, these designs exceed array partitioning limits, preventing the reading of all data in a single cycle as expected by the unrolling process. Moreover, these designs strain hardware resources, requiring backtracking, which extends the search duration.
- one out of four mainly optimize a single loop body and is not able to optimize the second loop body. Moreover, even the loop body optimizer is not perfectly optimized and can be more parallelized.

8.1.2 NLP-DSE . Now, we delve into how NLP-DSE overcomes these challenges to discover designs with superior QoR, emphasizing the usefulness of the Design Space Exploration (DSE) described in section 6.

The initial NLP-DSE design features parameters: PARA_L0 = 3, PIPE_L2 = flatten, PARA_L4 = 210, PARA_L1 = 6, PIPE_L3 = flatten, PARA_L5 = 190 achieving 13 GFLOPS/s. However, the compiler fails to apply PARA_L0 and PARA_L1 pragmas, creating a performance gap.

The second design is with the parameters: PIPE_L2 = flatten, PARA_L2 = 2, PARA_L4 = 210, PARA_L3 = 5, PIPE_L3 = flatten, PARA_L5 = 190 which allows to achieve a throughput of 85 GFLOPS/s. However the unroll factor of PARA_L3

does not allow the compiler Xilinx Merlin to transfer the data from off-chip to on-chip efficiently which does not allow to achieve the lower bound. Hence we continue the search and we found our the design with the best QoR in our space at the iteration 8 with the parameters: PIPE_L2 = flatten, PARA_L2 = 2, PARA_L4 = 210, PARA_L3 = 2, PIPE_L3 = flatten, PARA_L5 = 190 We can note that Xilinx Merlin did not allow the data to be transferred optimally so we did not achieve the lower bound and we continue the search. From iteration 10 the lower bound found by the NLP is greater than the latency (HLS report) of design 8, which makes it possible to stop the search because even if we reach the lower bound we will have a latency greater than what we have already obtained.

The Figure 6 summarize the result achieved at each step of the DSE. The step 4,5 and 6 (red) found the same configurations as the step 2 and 3 so we do not need to run the synthesis as we already have the results.

We execute our Design Space Exploration (DSE) using 8 threads, allowing us to concurrently evaluate multiple designs in parallel. This approach anticipates that certain designs may not achieve the desired performance, and by running 8 designs simultaneously, we efficiently explore the design space. For this specific example, we perform a single iteration of the DSE.

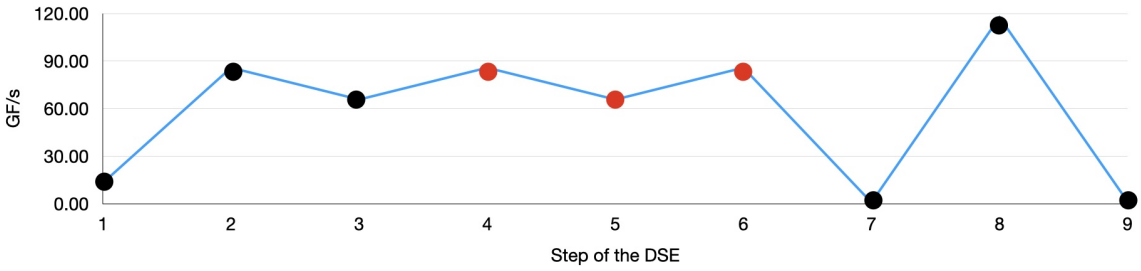


Fig. 6. Representation of the throughput (GF/s) achieved for each design obtained at each stage of the NLP-DSE for the 2mm kernel.

9 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. These model-free Design Space Exploration (DSE) techniques, as exemplified by works such as [13, 38, 47], employ a methodology where the compiler acts as a black-box, and they dynamically adapt their exploration strategies based on the outcomes of previous iterations. In these approaches, each candidate design is evaluated by generating a High-Level Synthesis (HLS) report. However, the time required for the synthesis or report generation can extend over several hours, significantly limiting the breadth of the explored design space. Moreover, it is worth noting that certain DSE methodologies, including those described in [38], might encounter challenges such as converging to local minima, which can impede the discovery of the globally optimal solution.

To circumvent the constraints imposed by synthesis time, novel approaches in Design Space Exploration (DSE) have emerged, including model-based DSEs and AI-driven DSEs. These methods leverage sophisticated techniques such as cost modeling [1, 52, 53, 55], Neural Networks (NN) [17, 21, 23, 32, 41, 45, 54], Graph Neural Networks (GNN) [34, 36, 42], or decision trees (DT) [24, 26, 48] to estimate the Quality of Results (QoR) of each design rapidly. By utilizing these techniques, the evaluation time for a single design can be reduced to mere milliseconds. However, despite this acceleration, assessing a large number of designs still entails a significant time investment. Furthermore, while these rapid evaluations provide valuable insights, they may not perfectly align with the outcomes obtained

from High-Level Synthesis (HLS) reports in terms of accuracy. Consequently, relying solely on HLS validation for the top-n results may lead to suboptimal solutions, as the rapid evaluation methods might not capture all pertinent design intricacies. Therefore, a more comprehensive approach that combines the strengths of both rapid evaluation techniques and traditional HLS validation is necessary to ensure optimal design outcomes.

In contrast, alternative methodologies offer one-shot optimization through code transformations and pragma insertion, as evidenced by works like [15, 18, 46]. However, the efficacy of these approaches is constrained by the limited scope of available hardware directives and code transformations. While some endeavors concentrate on predefined micro-architectures, as seen in [3, 40], their applicability is restricted. Moreover, specialized applications such as Deep Neural Networks (DNN) [49, 50], stencil computations [5], sparse linear algebra operations [10], and neural networks [2, 37], including Convolutional Neural Networks (CNNs) [30], have garnered attention. Yet, these methods encounter challenges when extrapolated beyond their designated domains, rendering generalizations difficult. Hence, while these approaches offer streamlined optimization strategies and tailored solutions for specific problem domains, their broader applicability beyond their respective niches remains a challenge.

NLP-DSE presents a hybrid methodology, leveraging a NLP-based cost model to swiftly explore expansive design spaces within minutes, potentially outpacing existing models in speed. Nevertheless, to ensure precise performance evaluation, reliance on High-Level Synthesis (HLS) remains integral to our approach. Recent advances in optimization solvers such as BARON [31, 39] 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. Unlike prior works [4, 29, 57] that frame the cost model as Linear Programming (LP) problems, NLP-based methods can handle more complex constraints without necessitating approximations, such as estimating communication volumes across loops [4, 29] or simplifying the space by exposing direct parallelization in the problem [57]. It is noteworthy that the comparison landscape lacked other NLP-based methodologies, while Linear Programming methods were deemed less suitable due to their incapacity to manage nonlinear constraints effectively. The inclusion of multiple product terms within the objective function and constraints, such as the product of Unroll Factors (UF) for perfectly nested loops, underscores the necessity for accurate modeling, which is challenging to approximate with linear approximations.

The selection of tile sizes remains fundamental for the final QoR. Similar to our approach, [22] 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.

Approaches that do not rely on precise static analysis, such as [13, 34, 38, 42, 47], can take as input any C/C++ kernel supported by the HLS compiler, thus expanding their applicability to a broader spectrum of programs. In contrast, NLP-DSE , akin to other model-based Design Space Exploration (DSE) approaches [1, 52, 53, 55], is constrained to affine programs to ensure precise analysis and facilitate the modeling of latency and resource utilization. While this encompasses a significant subset of programs, including AI kernels, and aligns with the MLIR affine dialect, it does not match the versatility of other frameworks. Consequently, there exists a tradeoff between accuracy and generality. Opting for a subset of programs provides more detailed information, thereby accelerating DSE and potentially improving the QoR.

10 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.

By formulating this model as a Non-Linear Program (NLP), the theoretically optimal pragma configuration can be determined. Our framework facilitates efficient design-space exploration by leveraging the latency lower bound property, allowing for the rapid elimination of points in the search space using a lightweight DSE process.

REFERENCES

[1] André Bannwart Perina, Jürgen Becker, and Vanderlei Bonato. 2019. Lina: Timing-Constrained High-Level Synthesis Performance Estimator for Fast DSE. In *2019 International Conference on Field-Programmable Technology (ICFPT)*. 343–346. https://doi.org/10.1109/ICFPT47387.2019.00063

[2] Suhail Basalama, Atefeh Sohrabizadeh, Jie Wang, Licheng Guo, and Jason Cong. 2023. FlexCNN: An End-to-end Framework for Composing CNN Accelerators on FPGA. *ACM Trans. Reconfigurable Technol. Syst.* 16, 2, Article 23 (mar 2023), 32 pages. https://doi.org/10.1145/3570928

[3] Suhail Basalama, Jie Wang, and Jason Cong. 2023. A Comprehensive Automated Exploration Framework for Systolic Array Designs. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC56929.2023.10248016

[4] 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

[5] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. 2018. SODA: Stencil with Optimized Dataflow Architecture. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1145/3240765.3240850

[6] Jason Cong, Muhuan Huang, Peichen Pan, Yuxin Wang, and Peng Zhang. 2016. Source-to-Source Optimization for HLS. In *FPGAs for Software Programmers*, Dirk Koch, Frank Hannig, and Daniel Ziener (Eds.). Springer, 137–163. http://dblp.uni-trier.de/db/books/collections/KHZ2016.html#CongHPW016

[7] Jason Cong, Muhuan Huang, Peichen Pan, Di Wu, and Peng Zhang. 2016. Software Infrastructure for Enabling FPGA-Based Accelerations in Data Centers: Invited Paper. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design* (San Francisco Airport, CA, USA) *(ISLPED '16)*. Association for Computing Machinery, New York, NY, USA, 154–155. https://doi.org/10.1145/2934583.2953984

[8] 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

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[10] Yixiao Du, Yuwei Hu, Zhongchun Zhou, and Zhiru Zhang. 2022. High-Performance Sparse Linear Algebra on HBM-Equipped FPGAs Using HLS: A Case Study on SpMV. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '22)*. Association for Computing Machinery, New York, NY, USA, 54–64. https://doi.org/10.1145/3490422.3502368

[11] Venmugil Elango, Fabrice Rastello, Louis-Noël Pouchet, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2015. On characterizing the data access complexity of programs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

[12] P. Feautrier. 1992. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *International Journal of Parallel Programming* 21, 6 (1992), 389–420.

[13] 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

[14] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. 2006. Semi-Automatic Composition of Loop Transformations. *International Journal of Parallel Programming* 34, 3 (June 2006), 261–317.

[15] 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

[16] Intel. 2023. Intel. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html

[17] David Koeplinger, Raghu Prabhakar, Yaqi Zhang, Christina Delimitrou, Christos Kozyrakis, and Kunle Olukotun. 2016. Automatic Generation of Efficient Accelerators for Reconfigurable Hardware. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 115–127. https://doi.org/10.1109/ISCA.2016.20

[18] 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

[19] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[20] 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

[21] 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.

[22] 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

[23] Shuangnan Liu, Francis CM Lau, and Benjamin Carrion Schafer. 2019. Accelerating FPGA Prototyping through Predictive Model-Based HLS Design Space Exploration. In *Proceedings of the 56th Annual Design Automation Conference 2019* (Las Vegas, NV, USA) *(DAC '19)*. Association for Computing Machinery, New York, NY, USA, Article 97, 6 pages. https://doi.org/10.1145/3316781.3317754

[24] H. Mohammadi Makrani, F. Farahmand, H. Sayadi, S. Bondi, S. Pudukotai Dinakarrao, H. Homayoun, and S. Rafatirad. 2019. Pyramid: Machine Learning Framework to Estimate the Optimal Timing and Resource Usage of a High-Level Synthesis Design. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE Computer Society, Los Alamitos, CA, USA, 397–403. https://doi.org/10.1109/FPL.2019.00069

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

[26] Kenneth O'Neal, Mitch Liu, Hans Tang, Amin Kalantar, Kennen DeRenard, and Philip Brisk. 2018. HLSPredict: Cross Platform Performance Prediction for FPGA High-Level Synthesis. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1145/3240765.3240816

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

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

[29] 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

[30] Enrico Reggiani, Marco Rabozzi, Anna Maria Nestorov, Alberto Scolari, Luca Stornaiuolo, and Marco Santambrogio. 2019. Pareto Optimal Design Space Exploration for Accelerated CNN on FPGA. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 107–114. https://doi.org/10.1109/IPDPSW.2019.00028

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

[32] B Carrion Schafer and Kazutoshi Wakabayashi. 2012. Machine learning predictive modelling high-level synthesis design space exploration. *IET computers & digital techniques* 6, 3 (2012), 153–159.

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

[34] 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)*.

[35] 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

[36] 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

[37] Atefeh Sohrabizadeh, Jie Wang, and Jason Cong. 2020. End-to-End Optimization of Deep Learning Applications. 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, 133–139. https://doi.org/10.1145/3373087.3375321

[38] 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

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

[40] Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA. 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, 93–104. https://doi.org/10.1145/3431920.3439292

[41] Shuo Wang, Yun Liang, and Wei Zhang. 2017. FlexCL: An analytical performance model for OpenCL workloads on flexible FPGAs. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/3061639.3062251

[42] 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

[43] AMD Xilinx. 2023. Merlin. https://github.com/Xilinx/merlin-compiler

[44] AMD Xilinx. 2023. Vitis. https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html

[45] Sotirios Xydis, Gianluca Palermo, Vittorio Zaccaria, and Cristina Silvano. 2015. SPIRIT: Spectral-Aware Pareto Iterative Refinement Optimization for Supervised High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, 1 (2015), 155–159. https://doi.org/10.1109/TCAD.2014.2363392

[46] 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

[47] 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

[48] Mang Yu, Sitao Huang, and Deming Chen. 2021. Chimera: A Hybrid Machine Learning-Driven Multi-Objective Design Space Exploration Tool for FPGA High-Level Synthesis. In *Intelligent Data Engineering and Automated Learning – IDEAL 2021*, Hujun Yin, David Camacho, Peter Tino, Richard Allmendinger, Antonio J. Tallón-Ballesteros, Ke Tang, Sung-Bae Cho, Paulo Novais, and Susana Nascimento (Eds.). Springer International Publishing, Cham, 524–536.

[49] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2018. DNNBuilder: an Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1145/3240765.3240801

[50] Xiaofan Zhang, Hanchen Ye, Junsong Wang, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2020. DNNExplorer: a framework for modeling and exploring a novel paradigm of FPGA-based DNN accelerator. In *Proceedings of the 39th International Conference on Computer-Aided Design* (Virtual Event, USA) *(ICCAD '20)*. Association for Computing Machinery, New York, NY, USA, Article 61, 9 pages. https://doi.org/10.1145/3400302.3415609

[51] 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

[52] 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

[53] 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

[54] Guanwen Zhong, Alok Prakash, Siqi Wang, Yun Liang, Tulika Mitra, and Smail Niar. 2017. Design Space exploration of FPGA-based accelerators with multi-level parallelism. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. 1141–1146. https://doi.org/10.23919/DATE.2017.7927161

[55] Wei Zuo, Warren Kemmerer, Jong Bin Lim, Louis-Noël Pouchet, Andrey Ayupov, Taemin Kim, Kyungtae Han, and Deming Chen. 2015. A polyhedral-based SystemC modeling and generation framework for effective low-power design space exploration. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 357–364. https://doi.org/10.1109/ICCAD.2015.7372592

[56] Wei Zuo, Warren Kemmerer, Jong Bin Lim, Louis-Noël Pouchet, Andrey Ayupov, Taemin Kim, Kyungtae Han, and Deming Chen. 2015. A polyhedral-based systemc modeling and generation framework for effective low-power design space exploration. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 357–364.

[57] Wei Zuo, Louis-Noel Pouchet, Andrey Ayupov, Taemin Kim, Chung-Wei Lin, Shinichi Shiraishi, and Deming Chen. 2017. Accurate High-Level Modeling and Automated Hardware/Software Co-Design for Effective SoC Design Space Exploration. In *Proceedings of the 54th Annual Design Automation Conference 2017* (Austin, TX, USA) *(DAC '17)*. Association for Computing Machinery, New York, NY, USA, Article 78, 6 pages. https://doi.org/10.1145/3061639.3062195

A PROBLEM SIZE

Kernel	$O(ops)$	$O(Mem)$	Large	Medium	Small
2mm	$O(n^3)$	$O(n^2)$	NI 800, NJ 900, NK 1100, NL 1200	NI 180, NJ 190, NK 210, NL 220	NI 40, NJ 50, NK 70, NL 80
3mm	$O(n^3)$	$O(n^2)$	NI 800, NJ 900, NK 1000, NL 1100, NM 1200	NI 180, NJ 190, NK 200, NL 210, NM 220	NI 40, NJ 50, NK 60, NL 70, NM 80
adi	$O(n^3)$	$O(n^2)$	TSTEPS 500, N 1000	TSTEPS 100, N 200	TSTEPS 40, N 60
atAx	$O(n^2)$	$O(n^2)$	M 1900, N 2100	M 390, N 410	M 116, N 124
bicg	$O(n^2)$	$O(n^2)$	M 1900, N 2100	M 390, N 410	M 116, N 124
cholesky	$O(n^3)$	$O(n^2)$	N 2000	N 400	N 120
correlation	$O(n^3)$	$O(n^2)$	M 1200, N 1400	M 240, N 260	M 80, N 100
covariance	$O(n^3)$	$O(n^2)$	M 1200, N 1400	M 240, N 260	M 80, N 100
deriche	$O(n^2)$	$O(n^2)$	W 4096, H 2160	W 720, H 480	W 192, H 128
doitgen	$O(n^4)$	$O(n^2)$	NQ 140, NR 150, NP 160	NQ 40, NR 50, NP 60	NQ 20, NR 25, NP 30
durbin	$O(n^2)$	$O(n)$	N 2000	N 400	N 120
floyd-warshall	$O(n^3)$	$O(n^2)$	N 2800	N 500	N 180
ftdt-2d	$O(n^3)$	$O(n^2)$	TMAX 500, NX 1000 NY 1200	TMAX 100, NX 200, NY 240	TMAX 40, NX 60 , NY 80
gemm	$O(n^3)$	$O(n^2)$	NI 1000, NJ 1100, NK 1200	NI 200, NJ 220, NK 240	NI 60, NJ 70, NK 80
gemver	$O(n^2)$	$O(n^2)$	N 2000	N 400	N 120
gesummv	$O(n^2)$	$O(n^2)$	N 1300	N 250	N 90
gramschmidt	$O(n^3)$	$O(n^2)$	M 1000, N 1200	M 200, N 240	M 60, N 80
heat-3d	$O(n^4)$	$O(n^3)$	TSTEPS 500, N 120	TSTEPS 100, N 40	TSTEPS 40, N 20
jacobi-1d	$O(n^2)$	$O(n)$	TSTEPS 500, N 2000	TSTEPS 100, N 400	TSTEPS 40, N 120
jacobi-2d	$O(n^3)$	$O(n^2)$	TSTEPS 500, N 1300	TSTEPS 100, N 250	TSTEPS 40, N 90
lu	$O(n^3)$	$O(n^2)$	N 2000	N 400	N 120
ludcmp	$O(n^3)$	$O(n^2)$	N 2000	N 400	N 120
mvt	$O(n^2)$	$O(n^2)$	N 2000	N 400	N 120
nussinov	$O(n^3)$	$O(n^2)$	N 2500	N 500	N 180
seidel-2d	$O(n^3)$	$O(n^2)$	TSTEPS 500, N 2000	TSTEPS 100, N 400	TSTEPS 40, N 120
symm	$O(n^3)$	$O(n^2)$	M 1000, N 1200	M 200, N 240	M 60, N 80
syr2k	$O(n^3)$	$O(n^2)$	M 1000, N 1200	M 200, N 240	M 60, N 80
syrk	$O(n^3)$	$O(n^2)$	M 1000, N 1200	M 200, N 240	M 60, N 80
trisolv	$O(n^2)$	$O(n^2)$	N 2000	N 400	N 120
trmm	$O(n^3)$	$O(n^2)$	M 1000, N 1200	M 200, N 240	M 60, N 80

Table 8. Complexity analysis of the number of operations and memory requirements for Polybench's problem sizes categorized as large, medium, and small.

Kernel	Problem Size	GF/s NLP-DSE	GF/s Harp	Perf. Improvement
2mm	Small	42.33	42.33	1.00
3mm	Small	33.04	27.66	1.19
atAx	Small	3.38	3.44	0.98
atAx	Medium	1.94	1.72	1.13
bicg	Small	1.82	1.83	0.99
bicg	Medium	0.98	0.92	1.07
covariance	Small	15.31	14.76	1.04
doitgen	Small	5.98	2.63	2.27
gemm	Small	16.41	27.57	0.60
gemm	Medium	120.63	125.59	0.91
gemver	Small	4.63	3.84	1.21
gemver	Medium	7.47	1.66	4.51
gesummv	Small	1.80	1.80	1.00
gesummv	Medium	1.96	1.96	1.00
heat-3d	Small	15.85	3.70	4.29
jacobi-1d	Small	4.07	4.24	0.96
jacobi-2d	Small	4.44	4.71	0.94
mvt	Small	3.62	3.62	1.00
mvt	Medium	3.93	7.07	0.56
seidel-2d	Small	0.04	0.04	1.01
syr2k	Small	8.40	9.49	0.88
syrk	Small	21.05	5.54	3.80
trmm	Small	0.03	0.03	1.00
Average				1.45
Geo. Mean				1.20

Table 9. Comparison between the throughput (GF/s) achieved by NLP-DSE and HARP, along with the performance improvement realized over HARP.

B 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.

B.1 A Formal Model for Latency

Definition B.1 (Straight-line code). An n-ary operation takes n scalar operands $\vec{i}$ 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 B.2 (Live-in set). The live-in set V_I^L 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 B.3 (Live-out set). The live-out set V_O^L of region L is the set of variables and memory locations that written to during any possible valid execution of L.

We can compute the directed acyclic graph made of all statements (i.e., all n-ary operations), connecting all producer and consumer operations, to build the operation graph:

Definition B.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 $< \{N, V_I, root, V_O\}, E >$ such that $\forall S_i \in L,\ N_{S_i} \in N$; and for every operation with output o and inputs $\vec{i}$ in L $\forall S_i, \forall i_k \in \vec{i}_{S_i}, e_{i_k,o} \in E$, $\forall S_i : (o_{S_i}, \vec{i}_{S_i}) \in L, S_j : (o_{S_j}, \vec{i}_{S_j}) \in L$ with $S_i \neq S_j$ then we have $E_{S_i,S_j} \in E$ iff $o_{Si} \cap \vec{i}_{S_j} \neq \emptyset$. 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 $V_{val} \in V_I$ (resp. V_O) for this input (resp. output) value. If $dim(\vec{i}_{S_i}) = 0$ then an edge e_{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.

Definition B.5 (Operation Graph critical path). Given OG^L an operation graph for region L. Its critical path OG_{cp} is the longest of all the shortest paths between every pairs $(v_i, v_o) \in < \{V_I, root\}, V_O >$. Its length is noted $\#OG^L_{cp}$.

B.1.1 Latency Lower Bound. We can build a simple a lower bound on the latency of an operation graph:

Theorem B.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 $LAT^L_{cp} \geq \#OG^L_{cp}$ is a lower bound on the minimal feasible latency to execute L.*

Proof Th B.6. Every operation S_i is associated with at least one edge with a source in $< \{V_I, root\} >$, so there is a path between one of these nodes and every operation by construction in Def. B.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 operation may be removed by dead code elimination. Therefore the shortest path sp between a pair of nodes $(v_i, v_o) \in < \{V_I, root\}, V_O >$ 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. B.6, then this path must take at least $\#sp$ cycles to complete. As we take the largest of the shortest paths between all possible pairs (v_i, v_o) then OG^L_{cp} 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 $\#OG^L_{cp}$ 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.

Theorem B.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 $LO(\#OG^L_{cp}) = \sum_{n \in cp} LO(n)$ the critical path weighted by latency of its operations. The minimal latency of a region L is bounded by*

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

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

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

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.

We now need to integrate loops and enable the composition of latency bounds.

B.1.2 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:

Corollary B.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. B.1.*

Proof Co B.8. By construction the process of fully unrolling all the loops creates a straight-line code region without loop control, which therefore fits Def. B.1. □

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

Theorem B.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:*

$$Lat^{l}_{R_{op}} \geq \max(LO(\#OG^{L}_{cp}), \max_{o \in op}(\lceil \#L(o) \times L_o / R_o \rceil))$$

Proof Th B.9. By Corollary B.8. □

B.1.3 Loop Unrolling: partial unroll. Loop unrolling is an 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 $TC_l \mod UF_l \neq 0$ then an epilogue code to execute the remaining $TC_l \mod UF_l$ iterations is needed.

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 B.10 (Minimal latency of a partially unrolled loop with factor UF). *Given a loop l with trip count TC_l and loop body L, and unroll factor $UF \leq TC$. 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:*

$$Lat^{l,S}_{R_{op}} \geq \lfloor TC/UF \rfloor \times Lat^{L'}_{R_{op}}$$

Proof Th B.10. By construction and Theorem B.7, $Lat^{L'}_{R_{op}}$ is a lower bound on the latency of L', that is the sub-program made of UF iterations of the loop. $\lfloor TC/UF \rfloor \leq TC_l/UF$ 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 $Lat^{L'}_{R_{op}}$. □

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

$$Lat^{l,S}_{R_{op}} \geq \lfloor TC/UF \rfloor * Lat^{L}_{R_{op}}$$

Proof Th B.11. Given OG^i and OG^j two CDAGs, for a pair of distinct iterations i, j of loop l.

If $V^i_O \cap V^j_O = \emptyset$, then the graph $OG^{i,j}$ made of the two iterations i, j cannot have a smaller critical path length than OG^i and OG^j: there is no edge crossing OG^i and OG^j in $OG^{i,j}$ since outputs are distinct, therefore $Lat(OG^{i,j}) \geq \max(Lat(OG^i), Lat(OG^j))$.

If $V^i_O \cap V^j_O \neq \emptyset$. 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^i or OG^j and hence $Lat(OG^{i,j}) \geq \max(Lat(OG^i), Lat(OG^j))$.

□

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

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

$$Lat^{l,S}_{R_{op}} \geq \lfloor TC/UF \rfloor \times Lat^{L}_{R_{op}} \times \lfloor \log_2(UF) \rfloor$$

Proof Th B.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 added to the same memory cell. However each data can be adding independently two by two and the result of this independent addition can also be adding 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 executed independently in $Lat^{L}_{R_{op}}$ cycles, the straight line code has a latency greater or equal to $Lat^{L}_{R_{op}} \times \lfloor \log_2(UF) \rfloor$. And then similarly to Th. B.11 and B.12 the sequential execution of the loops without pragma repeat this process $\lfloor TC/UF \rfloor$ times.

□

```
L1: for (i = 0; i < 8; i++)  {
    c += a[i];
}
```

Listing 12. Example of code demonstrating a reduction, where a tree reduction technique can be applied, as depicted in Figure 1.

B.1.4 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 pipelined loop and the iteration latency of the operations of the statements during the NLP generation. Let *RecMII* and *ResMII* be the recurrence constraints and the resource constraints of the pipelined loop, respectively. We have $II \geq \max(ResMII, RecMII)$. $RecMII = \max_i \lceil \frac{delay(c_i)}{distance(c_i)} \rceil$ with $delay(c_i)$ the total latency in dependency cycle c_i and $distance(c_i)$ the total distance in dependency cycle c_i. We suppose that $ResMII = 1$, as we do not know how the resource will be used by the compiler. Hence, if the loop is a reduction loop then the $II \geq \frac{IL_{reduction}}{1}$ with $IL_{reduction}$ the iteration latency of the operation of reduction.

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

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

$$Lat^{l,P}_{R_{op}} \geq Lat^{L}_{R_{op}} + II * (TC_l - 1)$$

Proof Th B.13. $Lat^{L}_{R_{op}}$ is the minimal latency to complete one iteration of l by Theorem B.7. The initiation interval measures the number of elapsed cycles before the next iteration can start, it takes therefore at least $TC_l * II - 1$ to start $TC_l - 1$ 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. □

B.1.5 Loop pipelining and unrolling. A loop l with trip count TC_l can be pipelined and partially unrolled with $UF < TC_l$, in this case there is loop splitting where the trip count of the innermost loop equal to the unroll factor and the trip count of the outermost loop equal to $\frac{TC_l}{UF}$.

Theorem B.14 (Minimal latency of a pipelined loop with known II and partially unrolled). *Given a loop l with trip count TC_l, partially unrolled by an unroll factor $UF < TC_l$ and a loop body L. 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 pipelined fashion is bounded by:*

$$Lat^{l,P}_{R_{op}} \geq Lat^{L'}_{R_{op}} + II * (\frac{TC_l}{UF} - 1)$$

Proof Th B.14. By construction and Theorem B.9 the latency $Lat^{L'}_{R_{op}}$ is a lower bound of L'. As the loop was split due to the partial unrolling, the trip count of the pipelined loop is $\frac{TC_l}{UF}$. Theorem B.13 gives us the lower bound of the latency for a loop with a trip count equal to $\frac{TC_l}{UF}$.

□

B.1.6 Non-Parallel, Non-Pipelined Loops. We continue with a trivial case: if the loop is not optimized by any directive (including any automatically inserted by the compilers), i.e., not parallelized nor pipelined, then every next iteration of the loop starts only after the end of the prior iteration.

Definition B.15 (Lower bound on latency of a non-parallel, non-pipelined loop under resources constraints). Given a loop l with trip count TC_l which is neither pipelined nor parallelized, that is, iteration $i + 1$ starts after the full completion of

iteration i, for all iterations. Given $Lat^{L}_{R_{op}}$ the minimal latency of its loop body. Then

$$Lat^{l}_{R_{op}} \geq TC_l * Lat^{L}_{R_{op}}$$

B.1.7 Coarse-Grained parallelization. Coarse-grained parallelization is a performance enhancement technique involving the unrolling of a loop which iterates a loop body not fully unrolled i.e., containing at least a pipelined loop or a loop executed sequentially. It is therefore impossible to do a coarse-grained parallelization with a reduction loop because the n sub loop body are dependent on each other.

It follows a bound on the minimal latency of a coarse-grained unrolled loop:

Theorem B.16 (Minimal latency of coarse-grained unrolled loop). *Given a loop l, which is not a reduction loop, with trip count TC_l, an unroll factor $UF \leq TC_l$ and L the loop body iterated by the loop l with a latency lower bound $Lat^{L}_{R_{op}}$. 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:*

$$Lat^{l,S}_{R_{op}} \geq \lfloor TC/UF \rfloor \times Lat^{L'}_{R_{op}}$$

Proof Th B.16. By construction, Definition B.15, Theorem B.13 and definition of the loop body L, $Lat^{L}_{R_{op}}$ is a lower bound of the loop body L. As the loop is not a reduction loop there is no dependency between the loop bodies of L for different iteration of l and then the loop bodies can be executed in parallel. If $UF < TC_l$, then $\lfloor TC_l/UF \rfloor \leq TC_l/UF$ 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 $Lat^{L'}_{R_{op}}$.

□

B.1.8 Program latency lower bound under resource constraints. We now focus on the latency lower bound of a program, under 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 resources than predicted may be feasible, itself possibly leading to a better latency.

Theorem B.17. *Given a loop body L, the set of set of statements $\mathcal{S}_{seq}$ non executed in parallel, $\#L^{s}_{op}$ the number of operations op for the statements s, DSP_{op} the number of resources (DSPs) used for the operation op, MCU^{s}_{op} the maximal number of computational units the statement s can use in parallel at any given time, and the configuration of pragma $\vec{PV_i}$ for each loop. The minimal number of resource (DSPs) consumed, R^{min}_{used}, 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:*

$$R^{min}_{used} = \sum_{op} \max_{\mathcal{S} \in \mathcal{S}_{seq}} \left(\sum_{s \in \mathcal{S}} \#L^{s}_{op} \times DSP_{op} \times MCU^{s}_{op} \right)$$

Proof Th B.17. 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 $\#L^s_{op} \times DSP_{op} \times MCU^s_{op}$ DSPs. If a set of statements $\mathcal{S}$ are executed in parallel they cannot share the resource so the execution in parallel of the statement $s \in \mathcal{S}$ will require $(\sum_{s\in\mathcal{S}} \#L^s_{op} \times DSP_{op} \times MCU^s_{op})$ DSPs. By considering the maximum across all statements, we can guarantee that at least one set of statement executed in parallel will require $\max_{s\in\mathcal{S}_{seq}}(\sum_{s\in\mathcal{S}} \#L_{op} \times DSP_{op} \times MCU^s_{op})$ 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. □

Given a program and the available resource of DSP DSP_{avail}, if $R^{min}_{used} < DSP_{avail}$ the lower bound is valid and the program does not over-utilize the resources.

B.1.9 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 B.18 (Lower bound of the memory transfer latency for an array). *Given a loop body L, the set of array $\mathcal{A}$, an array $a \in \mathcal{A}$, and LAT^{mem}_a the latency to transfer the array a from off-chip to on-chip (inputs) and from on-chip to off-chip (outputs). $\forall a \in \mathcal{A}, LAT^{mem}_a \geq (\mathbb{1}_{a\in V^L_O} + \mathbb{1}_{a\in V^L_I}) \times footprint_a/max\_burst\_size$. With $\mathbb{1}_{cond} = 1$ if cond = true else 0.*

Proof Th B.18. In order to transfer all the elements of the array a we can use packing with a maximum packing allowed by the target device of max_burst_size, which means 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 transfers are done in at least one cycle, the minimum latency is $footprint_a/max\_burst\_size$ to transfer once 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., $a \in V^L_I$ and from on-chip to off-chip for the outputs i.e., $a \in V^L_O$.
□

Theorem B.19 (Lower bound of the memory transfer latency). *Given a loop body L, the set of array $\mathcal{A}$, the number of cycles to transfer the array a is bounded by* $\max_{a\in\mathcal{A}}(\mathbb{1}_{a\in V^L_O} + \mathbb{1}_{a\in V^L_I}) \times footprint_a/max\_burst\_size$.

Proof Th B.19. According to Th. B.18 the lower bound to transfer one array a is $(\mathbb{1}_{a\in V^L_O}+\mathbb{1}_{a\in V^L_I})\times footprint_a/max\_burst\_size$. 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 has a latency greater or equal to $(\mathbb{1}_{a\in V^L_O} + \in \mathbb{V}^{\mathbb{L}}_{\mathbb{I}}) \times footprint_a/max\_burst\_size$ and hence the memory latency is equal to $\max_{a\in\mathcal{A}}(\mathbb{1}_{a\in V^L_O} + \mathbb{1}_{a\in V^L_I}) \times footprint_a/burst\_size$. □

B.2 Summary

By composing all the theorems, this allows us to end up with the final latency lower bound of the program which is presented in theorems B.20 for the computation and B.21 for the computation and communication.

Theorem B.20 (Computation Latency Lower bound of a Program). *Given available resource DSP_{avail}, the properties vector $\vec{PV}_i$ for each loop and a program which contains a loop body L. The properties vector allows to give all the*

information concerning the trip counts and the II of the pipelined loops and to decompose the loop body L with a set of loops $\mathcal{L}_L^{non\ reduction}$ *potentially coarse-grained unrolled with* $\forall l \in \mathcal{L}_L, UF_l$ *and a set of reduction loops executed sequentially* $\mathcal{L}_L^{reduction}$ *which iterates a loop body* L_{pip}*. By recursion the loop body* L_{pip} *contains a pipelined loop* l_{pip} *which iterate a loop body* L_{fg} *fully unrolled. The loop body* L_{fg} *contains operations which can be done in parallel with a latency* $Lat_{R_{op}}^{L_{par}}$ *and operations which are reduction originally iterated by the loops* $\mathcal{L}_{L_{fg}}^{reduction}$ *with a latency* $Lat_{L_{seq}}$*.*

The computation latency lower bound of L, which respected $DSP_{ued}^{min} \leq DSP_{avail}$*, executed with tree reduction is:*

$$Lat_{R_{op}}^{L} \geq \prod_{l \in \mathcal{L}_L^{par}} \frac{TC_l}{UF_l} \times \prod_{l \in \mathcal{L}_L^{reduction}} TC_l \times Lat_{R_{op}}^{L_{pip}}$$

with
$Lat_{R_{op}}^{L_{pip}} = (Lat_{R_{op}}^{L_{fg}} + II \times (\frac{TC_{l_{pip}}}{l_{pip\_UF}} - 1))$ *and* $Lat_{R_{op}}^{L_{fg}} = Lat_{L_{par}} + Lat_{L_{seq}} \times \prod_{l \in \mathcal{L}_{L_{fg}}^{reduction}} \frac{TC_l}{UF_l} \times \log_2(UF_l)$.

Proof Th B.20. Through composition and the application of Theorems B.9, B.10 and B.11, $Lat_{R_{op}}^{L_{fg}}$ serves as a computation latency lower bound for the fully unrolled sub-loop body of L, denoted as L_{fg}, where $Lat_{L_{par}} + Lat_{L_{seq}} \times \prod_{l \in \mathcal{L}_{L_{fg}}^{reduction}} \frac{TC_l}{UF_l} \times \log_2(UF_l)$ represent the critical path of $Lat_{R_{op}}^{L_{fg}}$.

By employing composition alongside Theorems B.13 and B.14, $Lat_{R_{op}}^{L_{pip}}$ stands as a computation latency lower bound for L_{pip}.

Utilizing composition, Theorem B.16, and Definition B.15, $Lat_{R_{op}}^{L}$ emerges as a computation latency lower bound for L.

□

Theorem B.21 (Latency lower bound of a program optimized with Merlin pragmas). *Given available resource* DSP_{avail} *and a program which contains a loop body L with a computation latency* $Lat_L^{computation}$ *and a communication latency* $Lat_L^{communication}$*.*

The lower bound for L which respected $DSP_{ued}^{min} \leq DSP_{avail}$ *and where the computation and communication can not be overlap is:*

$$Lat_L = Lat_L^{computation} + Lat_L^{communication}$$

Proof Th B.21. The AMD/Xilinx Merlin compiler does not overlap computation and communication. Hence the computation and communication is a sum of the latency of computation and latency of communication. By composition and Theorems B.20 and B.19.

□