RESEARCH-ARTICLE

Holistic Optimization Framework for FPGA Accelerators

STÉPHANE POUGET, University of California, Los Angeles, Los Angeles, CA, United States

MICHAEL LO, University of California, Los Angeles, Los Angeles, CA, United States

LOUIS-NOEL POUCHET, Colorado State University, Fort Collins, CO, United States

JASON CONG, University of California, Los Angeles, Los Angeles, CA, United States

Open Access Support provided by:

University of California, Los Angeles

Colorado State University

Published: 11 November 2025
Online AM: 24 September 2025
Accepted: 05 September 2025
Revised: 23 July 2025
Received: 06 April 2025

Holistic Optimization Framework for FPGA Accelerators

STÉPHANE POUGET, Computer Science Department, University of California, Los Angeles, Los Angeles, United States
MICHAEL LO, ECE Department, University of California, Los Angeles, Los Angeles, United States
LOUIS-NOËL POUCHET, Computer Science Department, Colorado State University, Fort Collins, United States
JASON CONG, Computer Science Department, University of California, Los Angeles, Los Angeles, United States

Customized accelerators have revolutionized modern computing by delivering substantial gains in energy efficiency and performance through hardware specialization. Field-Programmable Gate Arrays (FPGAs) play a crucial role in this paradigm, offering unparalleled flexibility and high-performance potential. High-Level Synthesis (HLS) and source-to-source compilers have simplified FPGA development by translating high-level programming languages into hardware descriptions enriched with directives. However, achieving high Quality of Results (QoR) remains a significant challenge, requiring intricate code transformations, strategic directive placement, and optimized data communication.

This article presents **Prometheus**, a holistic optimization framework that integrates key optimizations - including *task fusion, tiling, loop permutation, computation-communication overlap, and concurrent task execution*-into a unified design space. By leveraging *Non-Linear Programming (NLP) methodologies*, Prometheus explores the optimization space under strict resource constraints, enabling automatic bitstream generation. Unlike existing frameworks, Prometheus considers interdependent transformations and dynamically balances computation and memory access.

We evaluate Prometheus across multiple benchmarks, demonstrating its ability to maximize parallelism, minimize execution stalls, and optimize data movement. The results showcase its superior performance compared to state-of-the-art FPGA optimization frameworks, highlighting its effectiveness in delivering high QoR while reducing manual tuning efforts.

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

Additional Key Words and Phrases: High-level synthesis, non-linear programming, compiler

ACM Reference Format:
Stéphane Pouget, Michael Lo, Louis-Noël Pouchet, and Jason Cong. 2025. Holistic Optimization Framework for FPGA Accelerators. *ACM Trans. Des. Autom. Electron. Syst.* 31, 1, Article 7 (November 2025), 37 pages. https://doi.org/10.1145/3769307

This work was supported in part by the NSF award #CCF-2211557, the CDSC industrial partners and the AMD/HACC Program. The authors would like to thank Prof. Luciano Lavagno and Dr. DJ Wang for their helpful discussions and valuable contributions to this research.
Authors' Contact Information: Stéphane Pouget, Computer Science Department, University of California, Los Angeles, Los Angeles, CA, USA; e-mail: pouget@cs.ucla.edu; Michael Lo, ECE Department, University of California, Los Angeles, Los Angeles, CA, USA; email: milo168@ucla.edu; Louis-Noël Pouchet, Computer Science Department, Colorado State University, Fort Collins, CO, USA; e-mail: Louis-Noel.Pouchet@colostate.edu; Jason Cong, Computer Science Department, University of California, Los Angeles, Los Angeles, CA, USA; e-mail: cong@cs.ucla.edu.

ACM 1084-4309/2025/11-ART7
https://doi.org/10.1145/3769307

1 Introduction

The rise of customized accelerators has revolutionized modern computing, driving significant improvements in energy efficiency and performance through hardware specialization. Among these accelerators, **Field-Programmable Gate Arrays** (**FPGAs**) have gained widespread adoption due to their flexibility, high performance, and adaptability across diverse domains, including machine learning, scientific computing, financial modeling, and embedded systems. Unlike fixed-function hardware such as **Application-Specific Integrated Circuits** (**ASICs**), FPGAs offer reconfigurability, enabling hardware adaptation to workload-specific requirements. However, achieving a good ***Quality of Result*** (**QoR**) on FPGAs remains a complex and resource-intensive process, requiring careful tuning of computation, memory access, and parallelism strategies.

To address this complexity, **High-Level Synthesis** (**HLS**) tools [32, 49, 64, 79] and source-to-source compilers e.g, [9, 38, 42, 62, 81], have emerged as critical enablers of FPGA adoption, allowing developers to write hardware-accelerated programs in high-level languages such as C++ and Python. These tools generate synthesizable hardware descriptions, leveraging pragmas (hardware directives) and code transformations to optimize execution. Despite their advantages, HLS-generated designs still require substantial manual tuning to achieve high QoR, as the compiler's ability to optimize performance remains highly dependent on the structure and directives applied to the input code.

1.1 Challenges in FPGA Optimization

Despite significant advancements in HLS automation and FPGA design methodologies, achieving high-performance and resource-efficient FPGA implementations remains a challenging problem. This complexity arises from the intricate interactions between computation, memory access, and parallel execution strategies. Several key challenges must be addressed to unlock the full potential of FPGA acceleration.

Code Transformation and Design Space Exploration: FPGA performance is highly dependent on how computations are structured and scheduled. Optimizations such as loop tiling, loop permutation, and loop unrolling play a crucial role in exposing parallelism and improving memory access efficiency.

Task Concurrency and Resource Management: Maximizing throughput in FPGA designs requires efficient task concurrency, where multiple computation kernels execute in parallel. However, coordinating parallel tasks introduces challenges in managing resource contention, memory bandwidth, and routing complexity. Overlapping computations effectively demands intelligent scheduling strategies that minimize stall cycles while avoiding bottlenecks caused by limited on-chip storage and off-chip memory latency.

Computation-Communication Overlap: FPGA acceleration is often hindered by inefficient data movement between off-chip memory (DDR/HBM) and on-chip compute units. Many workloads experience severe performance degradation due to memory access bottlenecks rather than computational limitations. Dataflow architectures aim to address this by implementing FIFO-based streaming to pipeline data transfers, but they often fail to exploit intra-task parallelism efficiently. In contrast, shared buffer models [62, 81] improve data reuse, but their rigid memory management strategies make it difficult to overlap computation and communication dynamically. A more adaptive approach is needed to orchestrate data movement in parallel with execution, reducing overall memory stalls.

Scalability and Multi-Super Logic Regions Utilization: FPGA architectures, especially those using **Stacked Silicon Interconnect** (**SSI**) technology, feature multiple **Super Logic Regions** (**SLRs**) to enhance scalability. However, most frameworks, including Merlin-based tools restrict designs to a single SLR, leading to underutilization of FPGA resources. While designing across multiple SLRs is possible, it significantly increases routing congestion, timing closure failures, and

bitstream generation complexity. Without an SLR-aware optimization strategy, large-scale tasks face a high probability of bitstream generation failures or degraded performance due to excessive inter-SLR communication overhead.

Comprehensive Design Space Exploration: Current **design space exploration** (**DSE**) methodologies are often narrow in scope, considering only a limited subset of optimizations at a time. Each transformation—whether it involves loop reordering, tiling, pipelining, or data partitioning—has a profound impact on the overall performance and resource usage. Frameworks that optimize only one aspect of the design fail to account for the interdependencies between different optimizations, leading to suboptimal results. To achieve maximum efficiency, FPGA designs require a global optimization approach that integrates all possible transformations while considering the constraints imposed by memory hierarchy, computation parallelism, and hardware utilization.

Addressing these challenges requires a unified optimization strategy that seamlessly integrates code transformation, task concurrency management, computation-communication overlap, and hardware-aware scheduling into a single framework. The proposed work aims to develop such an approach, ensuring high-performance FPGA implementations while maintaining synthesis feasibility.

1.2 Contributions and Prometheus Method

This work introduces Prometheus, a unified framework that automates FPGA DSE by integrating multiple optimizations into a single methodology specialized *for affine programs* [40]. Specifically, we operate on affine loop nests where statements can be distributed in multiple loops (i.e., loops are permutable [56]). Such programs encompass typical computational pattern in dense linear algebra, data mining, image processing, and so on. as exemplified in the PolyBench benchmarking suite [2, 55]. Unlike existing tools that optimize isolated aspects of FPGA acceleration such as AutoDSE [69], HARP [68], NLP-DSE [61], Sisyphus [62], and Stream-HLS [9], Prometheus holistically addresses code transformation, memory management, task concurrency, and hardware-aware scheduling. It leverages a hybrid execution model that balances shared memory reuse and dataflow streaming to maximize parallelism while minimizing memory overhead.

To explore the vast design space efficiently, Prometheus formulates the optimization process as a **Non-Linear Programming** (**NLP**) problem, enabling automatic selection of loop transformations, tiling strategies, pragma configurations, and memory partitioning policies. Unlike heuristic-based approaches, this formulation captures complex optimization interdependencies and ensures globally efficient designs. Additionally, Prometheus incorporates SLR-aware task scheduling, which partitions tasks across multiple SLRs. This addresses a major limitation in prior works that restrict execution to a single SLR, leading to underutilization of FPGA resources. By integrating SLR-aware optimizations, Prometheus reduces routing congestion, improves scalability, and ensures successful bitstream generation.

The methodology follows a structured process: First, Prometheus performs *affine code analysis* to identify parallelism opportunities and dependencies. It then generates a *dataflow graph* and fuses tasks that share the same outputs to minimize unnecessary communication overhead. Next, Prometheus constructs a comprehensive design space that includes all key optimizations outlined in Section 2, such as *task concurrency via dataflow*, *computation-communication overlap*, and *adaptive parallelism*, while ensuring resource constraints are met for each SLR. The *NLP-based DSE model*, as described in Section 4, is then employed to identify the theoretically optimal set of transformations. Once the best configuration is identified, Prometheus automatically generates the optimized design. Finally, the system produces *OpenCL host code* and compiles the FPGA bitstream with the HLS compiler.

The key contributions of this work are:

— Unified FPGA optimization framework that jointly optimizes loop transformations, pragma insertion, task concurrency and computation-communication overlap while considering interdependencies between computation and data movement.
— Hybrid execution model that dynamically selects between shared buffering and dataflow streaming to maximize parallelism and efficient memory access.
— NLP-based DSE that automates the selection of loop tiling, scheduling, and memory strategies to achieve a globally optimal theoretical performance.
— SLR-aware scheduling and multi-SLR partitioning to balance routing complexity and FPGA resource utilization, overcoming single-SLR limitations in prior works.
— End-to-end compilation and automation, generating optimized *HLS-C++ code*—that is, standard C++ source code annotated with AMD/Vitis-specific HLS pragmas to guide hardware generation—alongside OpenCL host code and FPGA bitstreams, all with minimal manual intervention.
— Comprehensive performance evaluation, demonstrating superior QoR compared to AutoDSE [69], Sisyphus [62], ScaleHLS [81], Stream-HLS [9], and Allo [15].

Our framework, Prometheus, is open source and available at https://github.com/UCLA-VAST/Prometheus.

2 Background and Motivation

Optimizing latency and performance in FPGA designs relies on a combination of code transformations and strategic insertion of directives within HLS [19]. This section explores various HLS optimization techniques, their role in enhancing parallelism and resource utilization, as well as their limitations and challenges.

Table 1 presents a comparative analysis of various frameworks built on top of AMD Vitis HLS. These frameworks are categorized based on their underlying methodologies, including Model-Free, AI-Based, Cost Model Communication, Cost Model Computation, and NLP-Based approaches. The table evaluates their capabilities across key optimization techniques and objectives.

2.1 HLS Optimization

2.1.1 Pragma Insertion. HLS optimizations rely extensively on a range of directives that control loop execution, data access patterns, and computation scheduling to maximize performance. Among these, three fundamental pragmas—*unroll*, *pipeline*, and *array partitioning*—are particularly critical for improving parallelism and reducing execution latency.

The *unroll* directive enables the expansion of loop iterations in the hardware design, allowing multiple iterations to execute in parallel. By eliminating loop control overhead and enabling greater resource utilization, unrolling can significantly boost throughput. However, its effectiveness is limited by FPGA resource constraints, such as available DSP slices, BRAM, and routing congestion.

The *pipeline* pragma restructures loop execution to allow overlapping operations, thereby reducing the initiation interval (II)—the number of cycles required to launch consecutive iterations. This directive is essential for achieving high throughput in applications with iterative computations, such as matrix multiplications and stencil operations. Careful tuning of pipeline depth and initiation interval is necessary to balance resource usage while avoiding performance bottlenecks due to memory access contention.

Array partitioning is another crucial optimization that enhances memory access parallelism. By splitting large arrays into smaller banks stored in separate on-chip BRAM blocks or distributed memory structures, this pragma allows simultaneous data accesses, facilitating more effective

Table 1. *Comparison of HLS-Based FPGA Optimization Frameworks*

	Model Free	AI Model	Cost Model Communication	Cost Model Computation		NLP-Based			
	AutoDSE [69]	HARP [68]	PolyOpt-HLS [58]	ScaleHLS [81] / POM [82]	HeteroCL [38] / Allo [15]	Stream HLS [9]	NLP-DSE [61]	Sisyphus [62]	Prometheus
Pragma Insertion	✓	✓	Limit	✓	✓	✓	✓	✓	✓
Code Transformation (Tiling)	✗	✗	✓	Limit	✓	Limit	✗	✓	✓
Code Transformation (Loop Permutation)	✗	✗	Limit	Limit	✓	✓	✗	✓	✓
Code transformation + pragma insertion (unified)	✗	✗	✓	✗	✗	✗	✗	✓	✓
Task Concurrency	✗	✗	✓	Limit	✗	✓	✗	✗	✓
Dataflow	✗	✗	✓	✗	✓	✓	✗	✗	✓
Computation-Communication Overlap	✗	✗	✓	Limit	✗	✗	✗	✗	✓
Data Packing	✓	✓	✓	✗	✓	✗	✓	✓	✓
Padding (Communication)	✗	✗	✓	✗	✗	✗	✗	✗	✓
Padding (Computation)	✗	✗	✗	✗	✗	✗	✗	✗	✓
Hardware Aware	✗	✗	✗	✗	✗	✗	✗	Limit	✓
Hardware-feasible	Limit	Limit	✗	✗	✗	✗	Limit	Limit	✓
Management of Off-Chip and On-Chip Memory Transfers	✓	✓	✓	✗	✓	✗	✓	✓	✓
Objective	Comm + Comp	Comm + Comp	Comm	Comp	-	Comp	Comm + Comp	Comm + Comp	Comm + Comp
Enumeration (AI, heuristics, ...)	✓	✓	✗	✓	Manual	✗	✗	✗	✗

This table summarizes key features supported by various frameworks, including pragma insertion, code transformations, task concurrency, dataflow modeling, and hardware-awareness, categorized by their underlying optimization strategies.

unrolling and pipelining. Without partitioning, memory conflicts and bandwidth limitations can restrict performance, especially with multiple concurrent memory accesses.

Beyond these fundamental pragmas, additional directives such as *loop flattening*, and *dependency pragmas* can further optimize execution. *Loop flattening* combines nested loops into a single loop to improve hardware efficiency. Dependency pragmas help manage data dependencies to ensure efficient scheduling without unnecessary synchronization delays.

Pragma insertion has been extensively studied using various methodologies to optimize performance and resource utilization in HLS designs. These approaches include bottleneck analysis, as employed by AutoDSE [69], performance estimation through **Graph Neural Networks** (**GNN**) in frameworks like HARP [68], and cost model optimization leveraging NLP in NLP-DSE [60, 61].

Each method presents distinct advantages and limitations. AutoDSE achieves full accuracy by running the HLS compiler for each configuration, ensuring precise performance measurements. However, this approach is computationally expensive and time-consuming, making it impractical for rapid design exploration. HARP, on the other hand, can estimate performance across configurations without direct compilation, significantly reducing evaluation time. However, it requires extensive training datasets and fine-tuning for each new kernel to accurately model compiler behavior. While this approach enables learning-based adaptation to different designs, its effectiveness depends heavily on data quality and model calibration. Conversely, NLP-DSE explores the entire design space efficiently by formulating the optimization as a NLP problem. This method allows for rapid exploration and selection of theoretical optimal configurations. However, it relies on a theoretical cost model, which may introduce inaccuracies if the compiler behaves unpredictably or if certain optimizations are not accurately captured by the model. Consequently, while NLP-DSE provides a fast and scalable solution, its reliability depends on the fidelity of the cost model to actual HLS compilation behavior.

While pragma insertion is a powerful optimization technique, its effectiveness is inherently dependent on the interaction between memory hierarchy, computational resources, and FPGA-specific constraints. Moreover, its impact is tightly coupled with the program's execution schedule, meaning that without appropriate code transformations, the benefits of pragma insertion remain limited. Effective optimization requires a synergy between pragma directives and structural modifications to the code to fully exploit FPGA parallelism and resource utilization.

2.1.2 Code Transformation. Code transformations are fundamental for optimizing execution on FPGAs. While pragma insertion plays a crucial role in enhancing performance, it must be complemented by effective code transformations to fully exploit FPGA resources. Techniques such as loop reordering (permutation), task fusion, and data tiling enhance data locality and increase parallelism, thereby improving both computational efficiency and memory access patterns.

Various code transformation strategies have been developed specifically for FPGAs [18, 43, 45–47, 59, 83, 84]. While transformations originally designed for CPUs and GPUs achieve substantial performance gains by optimizing for their respective architectures, they do not inherently align with FPGA requirements, which prioritize fine-grained parallelism and efficient resource utilization. Several studies have utilized Pluto [14], a leading compiler framework originally designed for CPU optimizations, to transform FPGA kernels [83, 84]. While Pluto excels at tiling and minimizing dependencies to enhance memory reuse, its direct application to FPGA optimization is constrained due to the fundamental differences in optimization strategies required for CPUs and FPGAs. Unlike CPUs, where memory hierarchy and cache locality are primary concerns, FPGA optimization focuses on maximizing parallelism, minimizing resource contention, and efficiently utilizing on-chip memory, making Pluto's conventional transformation techniques less effective in this context. Conversely, studies such as [18, 43, 45–47, 59] focus on code transformations tailored to specific FPGA performance goals. The work in [59] aims to reduce communication overhead between off-chip and on-chip memory, achieving superior QoR for memory-bound kernels. Meanwhile, research efforts in [18, 43, 45–47] concentrate on optimizing pipelining strategies to maximize instruction-level parallelism and resource utilization. Sisyphus [62] introduces a unified approach by integrating code transformation and pragma insertion into a single optimization problem. By formulating this as a NLP problem, Sisyphus efficiently explores the design space to identify theoretical optimal configurations, streamlining FPGA acceleration while maintaining a balance between computation and memory access efficiency.

2.1.3 Task Concurrency. HLS tools like Vitis HLS support the *dataflow* pragma, which structures computations into actors that communicate through FIFO queues. This approach allows overlapping

execution of multiple tasks, significantly reducing overall latency. By enforcing a producer-consumer model, dataflow scheduling enables each task to process data as soon as it becomes available, rather than waiting for an entire stage to complete, which is particularly beneficial.

For computational kernels such as *3 mm* (Listing 4), the dataflow paradigm allows the first two matrix multiplications to execute in parallel while the third begins as soon as its required inputs are produced. This overlapping of execution helps maximize throughput and minimize idle time for computing units. Additionally, dataflow optimizations facilitate task-level parallelism, allowing independent tasks to run concurrently across multiple compute resources, such as DSP blocks and BRAM, ensuring efficient utilization of available FPGA resources.

However, despite its advantages, pure dataflow parallelism presents several challenges. One major limitation is intra-task parallelism, which is constrained by the reliance on FIFO-based communication. Since each FIFO can only transfer up to 512 bits per cycle (as this is the maximum off-chip memory bitwidth supported), this inherently restricts the amount of data that can be processed concurrently within a task. To further enhance intra-task parallelism, alternative approaches must be explored.

Stream-HLS [9] addressed this limitation by increasing the number of FIFOs connecting two tasks, assuming that all data resides on-chip. While this approach simplifies certain aspects of the design space, it is neither scalable nor generalizable for transferring data from off-chip to on-chip memory. By using n FIFOs to transfer n data elements in parallel between two tasks, the method significantly increases resource consumption without proportionally improving efficiency. The additional FIFOs complicate the design, potentially leading to routing congestion and excessive hardware overhead.

Moreover, modern FPGAs feature up to 32 off-chip memory banks, making the multi-FIFO approach impractical for handling off-chip data transfers. A more effective and scalable strategy must be developed to optimize intra-task parallelism while ensuring efficient communication between off-chip and on-chip memory, without introducing unnecessary complexity or resource constraints.

2.1.4 Shared Buffering. Shared buffering is a critical technique in FPGA memory optimization, enabling efficient data access and reuse across multiple computational units. It can be employed at a global level, as seen in frameworks like AutoDSE [69], Sisyphus [62], and ScaleHLS [81], or within individual dataflow tasks to enhance execution efficiency.

This approach involves preloading data buffers into on-chip memory, allowing multiple computations to access shared data without redundant transfers. By reducing memory access latency and improving bandwidth utilization, shared buffering facilitates high parallelism and optimizes overall performance. It is particularly beneficial in applications that are computation-bound.

However, shared buffering presents challenges in maintaining concurrency. While it enables efficient data reuse, it can introduce synchronization overhead, especially when multiple tasks attempt to access the same memory region simultaneously. Managing concurrent access requires arbitration mechanisms, which can lead to increased latency and potential bottlenecks. Additionally, routing congestion can occur due to high interconnect demands, limiting scalability in complex FPGA designs.

One limitation of shared buffering is its impact on initiation interval (II) in pipelined architectures. Unlike dataflow-based designs that rely on FIFO queues for seamless data streaming, shared buffering requires explicit read and write coordination, which may introduce stalls if not carefully managed. Moreover, FPGA resource constraints, such as limited BRAM and URAM availability, impose restrictions on buffer size and allocation strategies.

To address these challenges, advanced techniques such as double-buffering, memory partitioning, and adaptive scheduling have been explored. Double-buffering enables overlapping computation with data transfer, reducing idle cycles. Memory partitioning distributes data across multiple banks

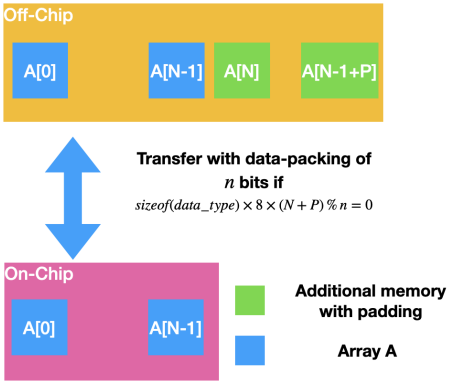

Fig. 1. Illustration of padding strategy to align array size for efficient data packing and memory transfers on AMD FPGAs.

```
// Original loop without padding
for (int i = 0; i < 190; i++){
#pragma HLS unroll factor=UF
// Possible UF values: 1, 2, 5, 10, 19, 38,
      95, 190
    A[i] += 1;
}

// Padded loop
for (int i = 0; i < 190 + 2; i++){ //
     Padding of 2
#pragma HLS unroll factor=UF
// Possible UF values: 1, 2, 3, 4, 6, 8,
      12, 16, 24, 32, 48, 64, 96, 192
    A[i] += 1;
}
```

Listing 1. Effect of Padding on the Space of Unroll Factors for Computation.

to alleviate contention, while adaptive scheduling dynamically assigns buffer access based on task priority and workload demands.

By effectively integrating shared buffering with intelligent memory management strategies, FPGA designs can achieve a balance between computational parallelism and efficient memory access. Future advancements should focus on automated buffer allocation techniques and dynamic access pattern optimization to further enhance performance in HLS workflows.

2.1.5 Computation-Communication Overlap. Overlapping computation and communication is crucial for high-performance FPGA designs. Techniques such as double buffering (ping-pong buffering) and advanced data tiling help mask communication latency while keeping computational units busy. Managing data transfers efficiently between on-chip and off-chip memory ensures that processing units remain active without waiting for data.

2.1.6 Data Packing and Padding. Modern FPGA architectures support high-bandwidth data transfers (up to 512-bit wide on AMD/Xilinx FPGAs). Data packing and padding optimize memory alignment, reducing the number of required memory cycles. For instance, transferring 216 floating-point values using a 256-bit width (8 floats per transfer) requires 27 cycles—compared to 216 cycles without packing. This highlights the importance of efficient data packing to minimize overhead. However, to enable such packing, the transfer vector size must evenly divide the total array size.

To fully exploit data packing, padding must be considered to enable even faster data transfers. Additionally, padding is valuable for achieving finer control over parallelism and resource utilization by expanding the available design space for loop unrolling, thereby enhancing computational throughput and efficiency.

Padding for Communication. Padding must be considered to increase the flexibility of data packing. The original size of the data may impose restrictions on packing efficiency, but by introducing padding, we create a larger space that may allow for a more efficient transfer. However, padding is not a free optimization, as increasing it also increases the amount of data that needs to be transferred.

Figure 1 illustrates the selection of the padding value P for an array A of size N. By leveraging data packing (up to 512 bits), multiple elements can be transferred per cycle. However, due to constraints in AMD compilers and FPGA architectures, the total transfer size must be divisible by a power of two. Padding is therefore introduced to align the data size accordingly and enable more efficient transfers.

In the code presented in Listings 2 and 3, where $J = 190$, transferring all elements along the second dimension (iterated by loop *j*) of array B onto the chip is constrained by the available memory bandwidth. Without padding, the maximum transfer rate is 64 bits per cycle, as 190×32 is divisible by 64 but not by 128. However, by introducing padding with $P = 2$ and adjusting J to 192 (190 + 2), the data alignment enables a significantly higher transfer rate of 512 bits per cycle, as 192×32 is now perfectly divisible by 512. This optimization maximizes memory bandwidth utilization, reducing transfer latency and improving overall data throughput.

Padding for Computation. Padding can also be used for computation, if we take a similar method than Sisyphus [62] which tiled and unroll the intra-tile loops which correspond to the transformation from Listing 2 to Listing 3. We will have an unroll factor which correspond to $I1 \times J1 \times K1$ but we want to have $I1$ divide I, $J1$ divide J and $K1$ divide K because we do not want to use extra resource to only compute a partial tile which would correspond to the execution of the last tile which is not complete. For example, if we have a trip count of 190 and we use an unroll factor of 8 then we will execute 184×8 a full tile and then 6 iterations for the partial tile.

If the unroll factor is restricted to be a divisor of the loop trip count, the design space becomes significantly limited. Padding can be used to adjust the trip count, thereby expanding the set of valid unroll factors to better match available hardware resources. As shown in Listing 1, a loop with a trip count of 190 permits only a limited set of unroll factors: $UF \in \{1, 2, 5, 10, 19, 38, 95, 190\}$. However, by padding the loop to a trip count of 192, the space of legal unroll factors becomes: $UF \in \{1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96, 192\}$. This increased flexibility allows finer control over unrolling and resource utilization, enabling more efficient hardware implementations and better exploitation of computational parallelism.

```
for (i = 0; i < I; i++)
  for (j = 0; j < J; j++)
    for (k = 0; k < K; k++)
      C[i][j] += A[i][k] * B[k][j];
```

Listing 2. *Baseline Implementation of Matrix Multiplication in C.* This code depicts a naive triple-nested loop structure used for matrix-matrix multiplication, serving as the unoptimized reference for further transformations.

```
for (i0 = 0; i0 < I0; i0++)
  for (j0 = 0; j0 < J0; j0++)
    for (k0 = 0; k0 < K0; k0++)
      for (i1 = 0; i1 < I1; i1++)
#pragma HLS unroll
        for (j1 = 0; j1 < J1; j1++)
#pragma HLS unroll
          for (k1 = 0; k1 < K1; k1++)
#pragma HLS unroll
            C[i0*I1+i1][j0*J1+j1] += A[i0*I1+i1][k0*K1+k1]*B[k0*K1+k1][j0*J1+j1];
```

Listing 3. *Matrix Multiplication with Loop Tiling and Fully Unrolled Intra-Tile Computation.* This implementation showcases a performance-optimized matrix multiplication using loop tiling to enhance data locality and fully unrolled intra-tile loops to expose fine-grained parallelism.

2.2 Performance and Hardware Considerations

2.2.1 Performance Evaluation. Performance evaluation in HLS is a multi-stage process that assesses different aspects of a design's efficiency and feasibility. It is typically conducted at three levels: estimation (e.g., Vitis HLS reports), RTL simulation, and FPGA-based evaluation. Each of these stages provides critical insights, but they vary in accuracy, speed, and resource requirements.

(1) Estimation (HLS Reports): The first level of evaluation relies on estimation tools provided by HLS compilers such as Vitis HLS. These reports generate quick approximations of key performance metrics, including latency, resource utilization (DSPs, LUTs, and BRAMs), and throughput. While these estimations are useful for early-stage design exploration, they do not account for low-level placement and routing effects, which can significantly impact real-world performance. The optimistic assumptions of HLS reports often fail to capture routing congestion, memory access delays, and other architectural bottlenecks.
(2) RTL Simulation: To achieve more accurate performance insights, RTL simulation is performed after HLS compilation. This stage involves generating synthesizable RTL code, which is then simulated using **hardware description language** (**HDL**) tools. RTL simulation provides cycle-accurate performance metrics, allowing designers to analyze pipeline behavior, data dependencies, and execution timing. However, while RTL simulation models hardware more precisely than HLS reports, it does not incorporate real-world FPGA constraints such as clock tree synthesis, interconnect delays, and power distribution, which can affect the final implementation.
(3) FPGA-Based Evaluation: The most accurate method of performance evaluation involves deploying the design onto an actual FPGA. This process includes place-and-route, bitstream generation, and execution on hardware. FPGA-based evaluation provides real execution metrics such as operating frequency, power consumption, and effective memory bandwidth. It also reveals practical challenges, including timing closure issues, placement inefficiencies, and resource contention that cannot be detected in earlier stages. This level of evaluation is essential for validating QoR and ensuring the design meets real-world constraints.

The transition from RTL simulation to FPGA implementation often introduces additional challenges, such as increased routing complexity, unexpected timing violations, and suboptimal resource utilization. These discrepancies arise due to the abstraction gap between HLS-generated code and final FPGA hardware constraints. To mitigate these issues, it is essential to develop robust tools that enable rapid design regeneration by modifying specific configurations within a well-defined portion of the design space. Such tools would enable engineers to efficiently iterate on design parameters without requiring a complete code regeneration, which can be time-consuming and may result in a significantly different design that still fails to generate a valid bitstream. Having the ability to selectively modify only the congested parts of the design while preserving the rest of the configuration would be highly valuable, ensuring faster convergence toward an optimized and feasible FPGA implementation.

2.2.2 Hardware Awareness and Resource Constraints. Many existing studies conclude their evaluations at the Vitis HLS report or RTL simulation stage, overlooking the critical impact of placement and routing constraints. However, as the design progresses toward bitstream generation, the available design space becomes increasingly constrained. Addressing hardware limitations early in the HLS process is essential to avoid costly design iterations and ensure convergence toward a feasible and deployable FPGA implementation.

Additionally, SLR-aware optimizations are crucial for multi-SLR FPGA architectures. A SLR is a physically distinct section of an FPGA die in SSI technology. Each SLR contains a subset of the device's computational resources, including DSP, LUT, and FF, similar to those in monolithic FPGA devices.

Effectively mapping tasks across SLRs is critical for balancing resource utilization and minimizing routing congestion, as inter-SLR communication introduces additional latency and can become a performance bottleneck. By integrating SLR-aware task partitioning, optimized scheduling, and efficient data movement strategies, FPGA designs can achieve higher scalability, improved

parallelism, and enhanced synthesis feasibility. Ensuring that computation and memory access patterns align with the physical structure of multi-SLR FPGAs is essential for achieving high QoR while maintaining timing closure and efficient resource allocation.

Some recent frameworks, such as RapidStream-TAPA [27, 28] and PASTA [35], aim to improve the design scalability and performance of HLS programs on modern multi-die FPGAs through hardware-aware co-optimization of HLS and physical design. While RapidStream-TAPA focuses on latency-insensitive, FIFO-based communication and exploits coarse-grained floorplanning and pipelining for timing closure and parallel compilation, PASTA extends this model to include buffer-based inter-task communication using a generalized channel abstraction. Despite their advances, both frameworks still require a manually optimized task-parallel input code, which demands substantial expertise in HLS and hardware design, limiting their accessibility for non-expert users.

2.3 Challenges

While existing HLS optimizations provide substantial performance improvements, several challenges remain. One of the primary limitations is that many prior works restrict their optimization space to specific techniques or separate their optimization processes into multiple independent steps. This fragmented approach can lead to incoherent design decisions, where optimizations applied at one stage may not align with or may even counteract those applied at subsequent stages. Refer to Table 1 for a detailed comparison of these approaches.

A common oversimplification in many frameworks is the assumption that all data resides on-chip e.g., [9, 81]. While this simplifies memory access patterns in DSE, it often results in low QoR when deployed on real hardware. This assumption prevents frameworks from incorporating crucial tiling strategies, which are essential for optimizing data locality, reducing memory access overhead, and ensuring efficient off-chip communication. When off-chip memory management is treated as a separate optimization step, decisions made in earlier phases may restrict the effectiveness of later memory optimizations, leading to suboptimal performance and resource utilization.

The limitations of current HLS optimization frameworks that rely on shared buffering are exemplified through the *3 mm* kernel. It consists of two independent matrix multiplications, whose outputs serve as inputs for a final matrix multiplication. The 3 mm kernel computes: $G = (A \times B) \times (C \times D)$ where intermediate matrices E and F store the results of the two first multiplications.

As illustrated in Listing 4, each matrix multiplication consists of deeply nested loops that are well-suited to parallelization techniques such as loop unrolling, pipelining, and computation-communication overlap. However, conventional shared buffering strategies often fall short in effectively leveraging concurrency, primarily due to their limited exploitation of dataflow principles and insufficient overlap between computation and communication.

Another major challenge is balancing parallelism and routing complexity. Excessive parallelization can lead to routing congestion, increased place-and-route time, and ultimately, failure to generate a valid bitstream. Current methodologies lack adaptive mechanisms to dynamically adjust parallelism levels based on available routing resources and FPGA architecture constraints. Without such mechanisms, achieving high-performance designs often requires manual intervention and iterative fine-tuning.

While pragma-based optimizations such as *unroll* and *pipeline* significantly improve performance, their effectiveness is inherently tied to the underlying code structure. Without proper transformations to expose parallelism and optimize memory access patterns, pragma insertion alone cannot fully unlock the potential of FPGA acceleration.

Addressing these challenges requires a holistic approach that integrates memory-aware optimizations, automated DSE, and adaptive scheduling techniques. Future research should focus on frameworks that jointly optimize data placement, computational parallelism, and resource

utilization while considering real hardware constraints. By developing more cohesive and intelligent HLS optimization strategies, FPGA-based acceleration can achieve greater efficiency, scalability, and deployment feasibility.

Model-free approaches, such as AutoDSE [69] and AI-based frameworks like HARP [68, 75, 76], rely on either HLS compilers or AI models to predict performance and resource utilization. However, these methods are currently limited to pragma insertion and space enumeration. Even though AI-based models provide fast estimations (on the order of milliseconds), exhaustively exploring large design spaces remains infeasible. Our previous work, NLP-DSE [61], addresses this limitation by leveraging NLP techniques to efficiently explore pragma configurations. However, like the other approaches, it remains restricted to pragma insertion.

PolyOptHLS [58] focuses on optimizing designs by minimizing memory transfers. While reducing communication latency can significantly improve performance, it does not always lead to optimal results. In many cases, the primary bottleneck may shift from communication to computation, limiting overall efficiency and potentially degrading the QoR. This approach, therefore, lacks a balanced optimization strategy that considers both computation and communication trade-offs.

ScaleHLS [81] and POM [82] extend optimization beyond pragma insertion by incorporating code transformations. However, their exploration space is constrained by the assumption that data is already on-chip. Additionally, their transformations are based on heuristics, such as permuting the reduction loop to the outermost level. While they employ a cost model to minimize computation latency, they still rely on exhaustive enumeration of possible configurations.

Allo [15] aims to reduce development time and enhance design quality through a composable programming model for hardware accelerator design. By decoupling algorithm specification from hardware customization and enabling modular schedule composition, Allo supports holistic dataflow optimization and verifiable transformations for large-scale, multi-kernel designs. However, it still requires significant manual intervention from an expert, limiting its usability for non-specialists.

Our previous work, Sisyphus [62], enables both code transformations and pragma insertion but is restricted to optimizing a single task. It lacks support for optimizations that dataflow techniques can provide, which would be necessary for broader efficiency improvements.

Stream-HLS [9] effectively leverages dataflow pragmas by selecting a good loop ordering strategy to maximize streaming efficiency. However, it imposes constraints on the design space by assuming that data is on-chip. Moreover, its approach to increasing parallelism relies on multiple FIFOs, which is not generalizable when off-chip memory access is required.

A significant limitation of all these frameworks is that none of them account for hardware constraints in their exploration space. While this simplifies the exploration process, it reduces real-world applicability. Additionally, frameworks such as Merlin-based tools (AutoDSE, HARP, NLP-DSE) and Sisyphus can generate all necessary components, including off-chip memory, for bitstream generation. However, they are constrained to a single SLR, leading to under-utilization of the FPGA board. Although generating a design that spans multiple SLRs is theoretically possible, it significantly increases the risk of bitstream generation failures. Even if the bitstream is successfully generated, timing violations often degrade performance.

2.4 Overview of Prometheus

We introduce Prometheus, a unified optimization framework designed to efficiently navigate complex design spaces using a NLP cost model. Prometheus streamlines FPGA design development by providing an intuitive, engineer-friendly interface with tunable parameters, including tile size, array partitioning, and loop unrolling factors.

The framework integrates all the optimization techniques discussed in previous sections, ensuring a comprehensive and cohesive approach to performance enhancement. Additionally, Prometheus is

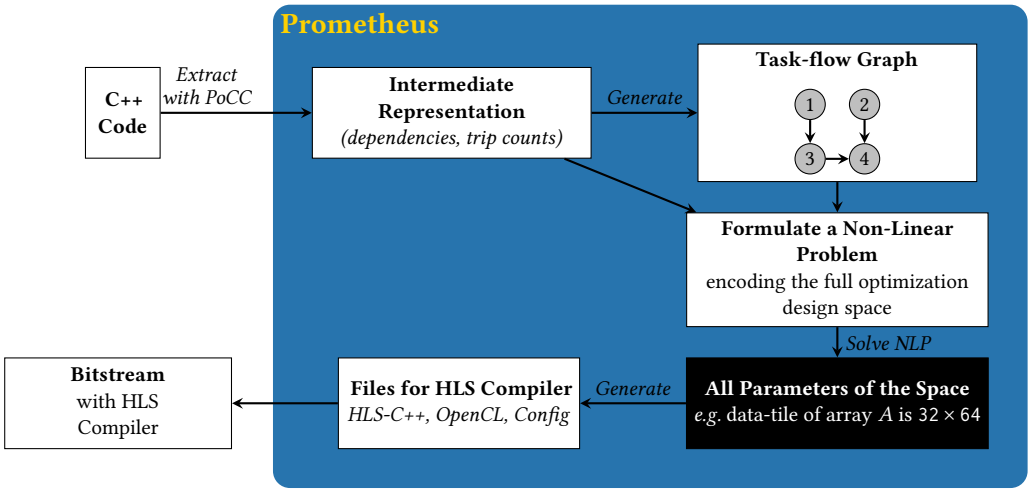

Fig. 2. *Overview of the prometheus framework workflow.* This diagram illustrates the end-to-end pipeline of the prometheus optimization framework, starting from C++ source code and proceeding through intermediate representation extraction, task-flow graph generation, NLP-based DSE, and final compilation into FPGA bitstreams using HLS compilers.

```
for (i = 0; i < 180; i++)  // MM 1
  for (j = 0; j < 190; j++) {
    E[i][j] = 0.0; // S0
    for (k = 0; k < 200; ++k)
      E[i][j] += A[i][k]*B[k][j];}// S1
for (i = 0; i < 190; i++)  // MM 2
  for (j = 0; j < 210; j++) {
    F[i][j] = 0.0; // S2
    for (k = 0; k < 220; ++k)
      F[i][j] += C[i][k]*D[k][j];}// S3
for (i = 0; i < 180; i++)  // MM 3
  for (j = 0; j < 210; j++) {
    G[i][j] = 0.0; // S4
    for (k = 0; k < 190; ++k)
      G[i][j] += E[i][k]*F[k][j];}// S5
```

Listing 4. *Reference Implementation of the 3 mm Kernel from PolyBench.* The 3 mm kernel computes: G = (A × B) × (C × D) where intermediate matrices E and F store the results of the two first multiplications.

```
for (i = 0; i < 180; i++)  // Task 0
  for (j = 0; j < 190; j++)
    E[i][j] = 0.0; // S0
for (i = 0; i < 180; i++)  // Task 1
  for (j = 0; j < 190; j++)
    for (k = 0; k < 200; ++k)
      E[i][j] += A[i][k]*B[k][j];}// S1
for (i = 0; i < 190; i++)  // Task 2
  for (j = 0; j < 210; j++)
    F[i][j] = 0.0; // S2
for (i = 0; i < 190; i++)  // Task 3
  for (j = 0; j < 210; j++)
    for (k = 0; k < 220; ++k)
      F[i][j] += C[i][k]*D[k][j];}// S3
for (i = 0; i < 180; i++)  // Task 4
  for (j = 0; j < 210; j++)
    G[i][j] = 0.0; // S4
for (i = 0; i < 180; i++)  // Task 5
  for (j = 0; j < 210; j++)
    for (k = 0; k < 190; ++k)
      G[i][j] += E[i][k]*F[k][j];}// S5
```

Listing 5. Direct Transformation of Listing 4 Where Each Loop Body Becomes a Separate Task

SLR-aware, enabling intelligent task distribution across SLRs. This capability not only simplifies bitstream generation but also optimizes the trade-off between computational performance and resource utilization.

Figure 2 illustrates the workflow of Prometheus. The process begins with an input C++ code, from which we extract an intermediate representation containing dependencies, trip counts, schedules, and other relevant information. This extraction is performed using PoCC [54], which provides all necessary details for analysis. Next, we construct the task-flow graph based on this extracted information and formulate the corresponding NLP problem. Solving this NLP problem determines the theoretical optimal parameters for the design space. Once the NLP solver provides these

Table 2. *Design Variables and Architectural Constraints in the Prometheus Optimization Space*

Group	Parameter	Description
Design Variables		
Memory	Bit Width (BW_a)	Width of memory transfer in bits;
	Data-tile Size (modeled via Transfer Level $t_{a,l}$ in Section 4)	Size of the data tile of array a transferred either from off-chip to on-chip memory or between tasks. The transfer level determines this size.
	Data-tile Reuse Size (modeled via Reuse Level $d_{a,l}$ in Section 4)	Size of the data tile of array a reused within a task. The reuse level dictates this tile size.
	Buffering (N_a)	Number of buffers (two for double buffering, three if read and write), enabling overlap of load/compute/store.
	Communication Padding	Additional data added to enable wider burst transfers and better bandwidth utilization.
Parallelism	Unroll Factors (TC^l_{intra})	Unroll factor for each task, determined through tiling, where the intra-tile loop is fully unrolled to expose fine-grained parallelism.
	Array Partitioning ($AP_{a,d}$)	Number of partitions for array a along dimension d to support parallel accesses.
	Compute Padding	Padding applied to loop trip counts to increase legal unroll factors.
Code Structure	Loop Permutations	Legal reordering of loops to optimize data locality and parallelism; synchronized across fused statements.
	Tiling	Splitting loops into inter-tile and intra-tile levels for optimization and pipelining.
	SLR Assignment (slr_t)	Mapping of each task t to a SLR to enable spatial task distribution.
Design Constraints		
Resource Limits	Max Array Partitioning	Total partitions across all arrays must not exceed architectural limits.
	DSP Budget	Maximum DSP slices per SLR; used to limit total DSP usage by pipelined/unrolled tasks.
	On-Chip Memory Available	Total memory usage by buffered tiles (considering reuse and buffering) must be within BRAM capacity.

This table outlines the key configurable parameters used in Prometheus—including loop tiling, unrolling, array partitioning, and buffer management—along with hardware constraints such as resource limits and memory capacity. The cost model also accounts for concurrent task execution enabled by the `dataflow` pragma, as well as computation-communication overlap achieved through automatic double or triple buffering. These elements jointly define the valid and efficient design space explored by Prometheus.

parameters, we have all the required information to proceed. At this stage, we automatically generate the HLS-C++ code along with all the necessary files to produce the FPGA bitstream, ensuring an efficient and fully automated compilation process.

To support efficient exploration of the design space, Prometheus defines a set of tunable parameters and architectural constraints that capture the key aspects of FPGA acceleration. Table 2 provides a detailed overview of these parameters, including memory-related configurations (e.g., bitwidth, tiling, and reuse levels), parallelism controls (e.g., loop unrolling and array partitioning), and structural transformations (e.g., loop permutations and SLR assignments). In addition, the table outlines hardware constraints such as DSP budgets, on-chip memory limits, and maximum legal array partitioning. These variables and constraints jointly define the feasible design space

that the NLP-based optimization engine systematically explores to generate high-performance, hardware-aware FPGA implementations.

The pseudo-code in Listing 6 demonstrates how Prometheus processes the *3 mm* kernel (Listing 4). Load and store operations manage data transfers to and from off-chip memory, while send and receive operations handle inter-task communication using FIFO.

Each task*i* in the pseudo-code corresponds to the fully unrolled computation of an intra-tile for statement S*i* in Listing 4. An example of such a task is illustrated in Listing 7.

Arrays *E*, *F*, and *G* are initialized to zero within their respective tasks (e.g., S0, S2, and S4) and are not preloaded, reducing unnecessary memory overhead.

Prometheus enhances computation efficiency by fusing statements that produce the same outputs (e.g., in Listing 4), and it automatically formulates an NLP problem to determine the theoretically optimal parameters, such as loop schedules, array bit widths (e.g., 512 bits), tile sizes, reuse buffer sizes, transfer locations, and padding. The framework dynamically adjusts bit widths for arrays like *F*, *D*, and *G* to achieve a better balance between parallelism and resource usage.

```
/*************** Fused Task 0 ****************/
float B[204][192]; load_B(B);
for (i0 = 0; i0 < 18; i0++)//inter-tile loop
  float A[10][204]; load_A(A);
  for (j0 = 0; j0 < 6; j0++){//inter-tile loop
    float E[10][32];
    task0(E); // S0 + intra-tile loops
    for (k0 = 0; k0 < 51; ++k0)//inter-tile loop
#pragma HLS pipeline II=3
      task1(E, A, B); // S1 + intra-tile loops
    store_E(E); sent_E(E);
/*************** Fused Task 1 ****************/
float C[190][222]; load_C(C);
for (j0 = 0; j0 < 7; j0++)//inter-tile loop
  float D[222][32]; load_D(D);
  for (i0 = 0; i0 < 10; i0++){//inter-tile loop
    float F[19][32];
    task2(F); // S2 + intra-tile loops
    for (k0 = 0; k0 < 74; ++k0)//inter-tile loop
#pragma HLS pipeline II=3
      task3(F, C, D); // S3 + intra-tile loops
    store_F(F); sent_F(F);
/*************** Fused Task 2 ****************/
float E[180][192];
for (j0 = 0; j0 < 7; j0++)//inter-tile loop
   float F[192][32];
   receive_F(F);
  for (i0 = 0; i0 < 18; i0++){//inter-tile loop
    float G[10][32];
    receive_E(E);
    task4(G); // S4 + intra-tile loops
    for (k0 = 0; k0 < 32; ++k0)//inter-tile loop
#pragma HLS pipeline II=3
      task5(G, E, F); // S5 + intra-tile loops
    store_G(G);
```

Listing 6. *Transformed and Fused 3 mm Kernel Pseudocode Generated by Prometheus.* This code highlights the result of Prometheus optimizations, including fused task generation, tiled loops, memory buffer management, and pipelined execution with dataflow pragmas.

Efficient computation-communication overlap is achieved through ping-pong buffering, while concurrent task execution and FIFO-triggered dependent tasks maximize overall performance. Fused Tasks 0 and 1 execute concurrently, while Fused Task 2 begins as soon as the data tiles for *F* and *E* become available.

Table 3. *Measured Throughput of the 3 mm Kernel Using Various FPGA Frameworks*

Metric	Prometheus	Sisyphus	Stream-HLS	Allo	ScaleHLS	AutoDSE
Throughput (GF/s)	**368.36**	178.97	174.00	60.40	43.04	1.74

This table presents the runtime throughput (in GF/s) obtained from RTL simulation of the 3 mm kernel across multiple frameworks. Prometheus is shown to outperform other methods, including Sisyphus, Stream-HLS, and AutoDSE.

To assess the effectiveness of our proposed framework, we evaluate the performance of the 3 mm kernel using several **state-of-the-art** (**SOTA**) HLS-based FPGA optimization tools. As shown in Table 3, Prometheus significantly outperforms existing frameworks in terms of throughput, achieving 368.36 GF/s. This represents a substantial improvement over Sisyphus (178.97 GF/s) and Stream-HLS (174.00 GF/s), and far exceeds the results of Allo, ScaleHLS, and AutoDSE. The superior performance of Prometheus stems from its holistic DSE strategy, which integrates loop transformations, memory tiling, concurrent task execution, and SLR-aware scheduling to deliver optimized and hardware-feasible designs.

3 Code Transformation

In our dataflow model, we adopt a synchronous dataflow where the sizes of the arrays are known during compile time. This compile-time awareness enables us to construct a precise model that facilitates rigorous optimizations. To leverage FPGA parallelism, we implement an acyclic dataflow graph, ensuring parent nodes do not receive data from their children. While this constraint limits graph configurations and may increase resource usage, it reduces overall latency. To support this structure, we inline [16] each function in the input code to generate the required acyclic graph. Our primary objective is to minimize latency within resource constraints by overlapping communication and computation within tasks and executing independent tasks concurrently. To achieve this, we apply various transformations and optimizations, explored in this section, and navigate the design space using an NLP-based approach, detailed in Section 4.

3.1 Dependency Graph Creation

Our process starts with affine C/C++ code as input, which undergoes maximal distribution to ensure each loop body contains only one statement, provided no dependencies exist within the loop. ISCC [71] verifies the legality of these transformations, ensuring dependencies are preserved. After achieving full distribution, we construct a dependency graph using PoCC [54]. PoCC provides the necessary information about the schedule and dependencies, enabling us to build the graph. In this graph, the nodes represent tasks, while the edges capture data communication arising from inter-task dependencies. Tasks with identical outputs are then merged (when legal), creating fused tasks with output-stationary properties. This ensures that each tile's output is handled (loaded, computed, and either stored or transmitted) only once. If a dependency prevents distribution, the framework will still function but with a more limited optimization space, making it more effective when distribution is possible.

Figure 3 shows the task graph of the *3 mm* kernel (Listing 5), where each loop body corresponds to a distinct computation task.
Prometheus's solution space includes:

3.2 Data-tiling and Padding

In the fused dependency graph, edges represent data communication between tasks as well as between tasks and off-chip memory, involving the transfer of data tiles with specified sizes. Data tiling divides loop iterations into smaller tiles, splitting array accesses into subsets. Each loop l

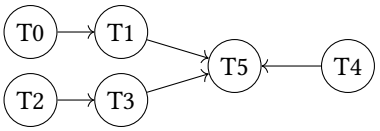

Fig. 3. Dataflow graph of the 3 `mm` kernel. Nodes represent computation tasks; edges denote data dependencies.

iterating over an array a is divided into an outer loop (TC^l_{inter}) and an inner loop (TC^l_{intra}), with feasible permutations applied. For each array and dimension iterated by the loop l, the tile factor is a common choice that influences all arrays iterated by this loop.

To optimize memory transfers, padding is applied to arrays in two ways. Simple padding increases the bit width (BW_a) for efficient transfers while maintaining the original loop trip count. Composite padding adjusts both BW_a and the loop trip count (TC^l) to support unroll factors that do not evenly divide the original trip count, allowing irregular tile sizes and expanding the design space. Tile sizes are consistent within a fused task but vary between tasks. In Listing 6, the tile size of array F is 19×32 in Fused Task 1 and 192×32 in Fused Task 2.

3.3 Fine-grained Parallelism

Data-tile selection involves splitting each loop and permuting them to create two levels of the original loops. Using ISCC [71], we verify the legality of these permutations, resulting in two loop levels: inter-tile (outer) and intra-tile (inner). If permutation is not feasible, the inter-tile loop retains its legal position. For tasks belonging to the same fused task, we merge their inter-tile loops, which are non-reduction. For instance, in Listing 6, the inter-tile loops on Lines 3 and 5 iterate over Task 0 and Task 1. The intra-tile loops are fully unrolled, ensuring that all data accesses within the intra-tile remain on-chip. Array partitioning, determined by the unroll factor, ensures data resides in separate BRAM banks for simultaneous access. Reduction loops across inter-tiles are pipelined with an initiation interval ($II = n > 1$), where n matches the reduction latency. For instance, in Listing 6, additions take 3 cycles, resulting in $II = 3$.

3.4 Loop Order

Due to the full unrolling of the intra-task, there is no need to select the loop order within the intra-tile. We place the inter-tile reduction loops directly above the task and are pipelined. If multiple reduction loops exist, we rank them by the size of their trip counts, placing the loop with the highest trip count innermost to ensure an efficient pipeline. This setup provides the flexibility to choose the order of the inter-tile loops that are not reduction loops. This order will subsequently be determined by the NLP (cf. Section 4). As shown in Listing 6, the loop order of Fused Tasks 1 and 2 is permuted compared to the original program in Listing 4.

3.5 Automatic Overlapping of Communication and Computation

To overlap communication and computation, we use on-chip buffers. The buffer size and data transfer location are determined by various options explored via NLP (cf. Section 4). Since data must stay on-chip for intra-task computation, transfers occur either below an inter-tile loop or before any loops start. The transfer position determines the data tile size, covering all data accessed below it. To improve data reuse, buffer size can match or exceed the transferred data tile. Similar to transfer location, buffer size is determined by its position relative to inter-tile loops or before any loops. Two boolean variables, $d_{a,l}$ and $t_{a,l}$, define and transfer array a under loop l when set to true. If defined or transferred before loops, $l = 0$. Double-buffering is used for read-only or write-only

```
1 for (int j1 = 0; j1 < 32; j1++)
2 #pragma HLS unroll
3   for (int i1 = 0; i1 < 19; i1++)
4 #pragma HLS unroll
5     for (int k1 = 0; k1 < 3; k1++) {
6 #pragma HLS unroll
7       j=j0*32+j1; i=i0*19+i1; k=k0*3+k1;
8       F[i1][j1] += C[i][k] * D[k][j1];}
```

Listing 7. *Implementation of Task 3 in the 3 mm Kernel (from Listing 6).* This listing illustrates the structure of Task 3, where the intra-tile computation is fully unrolled to expose fine-grained parallelism. The unrolling enables concurrent execution of loop iterations.

arrays, and triple-buffering for arrays that are both read and written. Following [19], we perform an initial load, then overlap loading the next tile with computing the current one.

3.6 Coarse-Grained Parallelism: Automatic Execution of Concurrent Tasks

Due to the use of the dataflow pragma, tasks can begin computation as soon as they have sufficient data, allowing for concurrent execution. This concurrency significantly increases overall performance.

3.7 Memory Transfer

Communication latency is reduced by transferring more data per cycle using increased bit width with padding. Read-only arrays are duplicated in off-chip memory for tasks with multiple reads, eliminating feed-through logic. For non-read-only arrays, data passes between tasks via FIFOs, using the same bit width as for off-chip memory transfer.

4 NLP Formulation

To determine the tile size, loop order, bit width, and memory transfers, we formulate a cost model as a NLP problem aimed at minimizing overall latency. This approach builds upon the methodology proposed in our previous work [60–62], which we have further adapted to meet the specific requirements and constraints of our current framework. In our work, we incorporate dataflow considerations along with all the optimizations detailed in Section 3. We employ PoCC [54] to extract compile-time information such as schedules, loop trip counts, dependencies, and operation counts per statement. ISCC [71] then generates all legal permutations for each loop body.

Table 4 delineates the sets, variables, and constants utilized in our NLP formulation.

4.1 Constraints

We now describe the constraints by using the code of Listings 4, 6, and 7.

4.1.1 Data-tiling and Unroll Factor. The intra-tile transformation, as explained in Section 3, can divide either the original loop trip count or the original trip count with padding, thereby increasing the range of possibilities. Equation (1) ensures that the trip count of the intra-tile is a divisor of one of these two possibilities. The user has the option to constrain the padding using Equation (2), which simplifies the solution space for the NLP solver. For instance, in Listing 7, for the array F, the loop j (Line 7 in Listing 4) has been split into $j0$ (Line 14 in Listing 6) and $j1$ (intra-tile). The trip count of the intra-tile loop $j1$, denoted as $TC^{j1}_{\text{intra}} = 32$, does not evenly divide the original trip count $TC^{j}_{\text{ori}} = 210$, but it does divide the trip count of the padded loop $TC^{j} = 224$.

$$\forall l \in \mathcal{L}, TC^{l}_{intra} \% TC^{l}_{ori} == 0 || TC^{l}_{intra} \% TC^{l} == 0, \tag{1}$$

$$(opt) \forall l \in \mathcal{L}, \exists n \in \mathbb{N} \leq N \in \mathbb{N}, \text{ s.t. } TC^{l} = TC^{l}_{ori} + n. \tag{2}$$

Table 4. *Mathematical Notation for Constants, Variables, and Sets in the NLP-Based Optimization Model*

Constants	Description
II_l	II of the loop l
IL_{par}, IL_{red}	Iteration Latency of the operations without (*par*) and with (*red*) dependencies of the statement s
TC^l_{ori}	Original Trip Count of the loop l
$f_{a,l}$	Footprint of the array a if transferred to on-chip after the loop l
N_{FT}	Number of fused task
$DSP_{s_{op}}$	Number of DSP used by the statement s for the operation op
DSP	Number of DSP available for the FPGA used
max_{part}	Maximum array partitioning
SLR	Number of SLR available for the FPGA used
Variables	**Description**
TC^l_{intra}, TC^l_{inter}	TC of the loop l for the intra and inter tile
TC^l	Trip Count of the loop l after padding
S^{last}_a	Size of the last dimension of the array a transferred on-chip
BW_a	Bit width of the array a
$t_{a,l}, d_{a,l}$	Boolean to know if the array a is transferred and defined (respectively) on-chip after the loop l in the inter-tile
p^l_i	Position of the loop l under the ith permutation
slr_t	ID of the SLR use by the task t
Sets	**Description**
$\mathcal{L}, \mathcal{A}, \mathcal{S}$	The set of loops, arrays and statements
$\mathcal{L}_s$	The set of loops which iterate the statement s
$\mathcal{L}^{red}_s$	The set of reduction loops which iterate the statement s
$\mathcal{L}^{last}_a$	The set of loop which iterate the last dimension of the array a
$\mathcal{L}_{inter}$, $\mathcal{L}_{intra}$	The set of loops which belong to the inter-tile and intra-tile, respectively
$\mathcal{B}$	Set of possible burst size for the data type
$\mathcal{C}_{a_d}$	The set of loops which iterates the array a at the dimension d
$AP_{a,d}$	Array Partition for the array a in dimension d
$\mathcal{P}_s$	All permutation of the loops which iterate the statement s
$\mathcal{F}_i$	The set of statements which belong to the fused task i
$\mathcal{T}$	The set of tasks in the dataflow

This table defines the formal notation used in Prometheus' NLP model for DSE, including loop trip counts, bitwidths, SLR mappings, resource limits, and legal transformations.

4.1.2 Bit Width. $\mathcal{B}$ denotes the number of elements that can be transferred simultaneously, determined by the bit width and the data type. Hence, if we have a bit width under 512 bits for *float* the set is $\{1, 2, 4, 8, 16\}$. Equation (3) computes the bit width for each array based on the last dimension of the data-tile transferred on-chip. For example, the array D in Fused task 1 is transferred in Line 15 with a size of 222×32, so $S^{last}_D = 32$, which is divisible by 16. Therefore, it has a bit width of 16.

$$\forall a \in \mathcal{A}, \forall l \in \mathcal{L}^{last}_a, \max_{b \in \mathcal{B}} BW_a = b \text{ s.t. } S^{last}_a \% b = 0. \tag{3}$$

4.1.3 Permutation. Equation (4) requires the NLP to choose identical permutations for loops that are shared by statements fused within the same task. For example, in Fused Task 1, the loops

iterating statement S2 can be permuted as ($i0$, $j0$) or ($j0$, $i0$), and similarly, loops iterating statement S3 can be permuted as ($i0$, $j0$) or ($j0$, $i0$). However, since the loops $i0$ and $j0$ iterate both S2 and S3, they must use the same permutation; either ($i0$, $j0$) for both or ($j0$, $i0$) for both.

$$\forall i \in [\![0, N_{FT}]\!], \forall (ft_0, ft_1) \in \mathcal{F}_i^2,$$
$$\forall (i_0, i_1) \in \mathcal{P}_{ft_0} \times \mathcal{P}_{ft_1}, \forall l \in \mathcal{L}, p_{i0}^l = p_{i1}^l. \tag{4}$$

4.1.4 Transfer and Reuse. Equation (5) permits the selection of a single level where each array can be defined (and reused) and transferred. Equation (6) constrains that the definition of the array must occur lexicographically before or at the same time as the transfer.

The array E, defined on Line 24 in Listing 6, is defined before any loops, so $d_{E,0} = 1$ (0 indicating it is defined before any loops). However, it is transferred under the loop $i0$, so $t_{E,i0} = 1$. Equation (6) simply means that the definition of array E should occur before or at the same level as the transfer. We cannot transfer E under loop $i0$ if E is defined under $k0$. Similarly, in Listing 6, the array A is defined and transferred in line 4, with $d_{A,i0} = 1$ and $t_{A,i0} = 1$.

$$\forall a \in \mathcal{A}, \sum l \in \mathcal{L}_{inter}, t_{a,l} = 1, \sum l \in \mathcal{L}_{inter}, d_{a,l} = 1, \tag{5}$$
$$\forall a \in \mathcal{A}, \sum (l_0, l_1) \in \mathcal{L}_{inter}^2 | d_{a,l_0}, t_{a,l_1} = 1, \text{ then } l_0 \preccurlyeq l_1. \tag{6}$$

4.1.5 On-chip Memory. Equation (7) constrains the footprint of the array to be within the available resources, based on where the array is defined, the number of double buffers used and the footprint of the array transferred at this level.

$$\sum_{a \in \mathcal{A}} \sum_{l \in \mathcal{L}} d_{a,l} \times f_{a,l} \times N_a \leq Mem, \tag{7}$$

with N_a being the number of double buffer for the array a.

4.1.6 Array Partitioning. Equation (8) limits the maximum partitioning of each array. This partitioning is crucial as it impacts the maximum unroll factor, necessitating the distribution of data across different BRAM banks under fully unrolled loops, thereby influencing the utilization of BRAMs. Equation (9) computes the array partitioning needed for each array based on the trip count of the fully unrolled intra-tile loops.

Array `D` in Listing 7 is traversed by two unrolled loops: $k1$, which iterates 3 times ($AP_{D,0} = 3$), and $j1$, which iterates 32 times ($AP_{D,1} = 32$). Therefore, the total number of partitions needed is $3 \times 32 = 96$. Consequently, these 96 values of F are stored in different banks, allowing all of them to be accessed in parallel. However, this value must be less than or equal to max_{part}.

$$\forall a \in \mathcal{A}, \prod_{d \in \mathbb{N}} AP_{a,d} \leq max_{part}, \tag{8}$$
$$\forall a \in \mathcal{A}, \forall d \in \mathbb{N}, \forall l \in C_{a_d}, AP_{a,d} = TC_{intra}^l == 0. \tag{9}$$

4.1.7 DSP Utilization. Equation (10) constrains the number of DSPs used based on the available DSP resources. In contrast to [60–62], we utilize pessimistic DSP utilization. This approach is necessary because concurrent execution and resource reuse between tasks are not feasible when two tasks can run simultaneously. Given $DSP_+ = 2$, $DSP_* = 3$, and $II_{S3} = 3$, the DSP usage for Task 3 is calculated as $(2 + 3) \times 1824$, accounting for the unroll factor. However, since the loop is pipelined with $II = 3$, the HLS compiler optimizes resource usage, effectively reducing the DSP count by approximately dividing by II.

$$\sum_{op \in \{+,-,*,/\}} \sum_{s \in \mathcal{S}} (DSP_{s_{op}} / II_s) \times \prod_{l \in \mathcal{L}_s} TC_{intra}^l \leq DSP. \tag{10}$$

4.1.8 SLR Selection. For each task, the NLP determines the SLR on which the task will be implemented (Equation (11)). Additionally, Equations 7 and 10 are applied to each SLR to manage resource allocation per SLR effectively.

$$\forall t \in \mathcal{T}, slr_t \in [\![0, SLR]\!]. \tag{11}$$

4.2 Objective Function

The objective of our framework is to minimize the total latency of the design, which is modeled as a **directed acyclic graph** (**DAG**). In this dataflow graph, each node represents a computation task (or fused task), and edges represent data dependencies between tasks.

To compute the overall latency, we define the latency of each task $T \in \mathcal{T}$ as the sum of:

— the time at which it becomes ready to start (determined by its dependencies)
— and the duration of the task itself

We denote:

— $\text{Lat}(T)$: the global latency contribution of task T, i.e., the time at which T finishes,
— $\text{Lat}_{\text{task}}(T)$: the execution time (duration) of task T,
— $\text{pred}(T)$: the set of immediate predecessor tasks of T,
— $\text{shift}_{T_i,T}$: the number of cycles after which T can start once T_i has begun (e.g., due to pipelined data production).

Then the latency of each task is recursively defined as:

$$\text{Lat}(T) = \max_{T_i \in \text{pred}(T)} \left[\text{Lat}(T_i) + \text{shift}_{T_i,T}\right] + \text{Lat}_{\text{task}}(T). \tag{12}$$

The overall latency of the design is defined as the latest finish time among all sink tasks (i.e., tasks without successors):

$$\text{Lat}_{\text{total}} = \max_{T \in \mathcal{S}} \left[\text{Lat}(T)\right], \tag{13}$$

where $\mathcal{S} \subseteq \mathcal{T}$ is the set of sink tasks in the graph.

For the graph in Figure 3, the latency can be expressed step-by-step as:

$$\text{Lat}(T_1) = \text{Lat}(T_0) + \text{shift}_{T_0,T_1} + \text{Lat}_{\text{task}}(T_1)$$
$$\text{Lat}(T_3) = \text{Lat}(T_2) + \text{shift}_{T_2,T_3} + \text{Lat}_{\text{task}}(T_3)$$
$$\text{Lat}(T_5) = \max\left(\text{Lat}(T_1) + \text{shift}_{T_1,T_5},\ \text{Lat}(T_3) + \text{shift}_{T_3,T_5},\ \text{Lat}(T_4) + \text{shift}_{T_4,T_5}\right) + \text{Lat}_{\text{task}}(T_5)$$

In the next section, we explain how to compute $\text{Lat}_{\text{task}}(T)$ for each task using intra-task and inter-tile modeling.

4.2.1 Intra-Task and Multi-Level Latency Modeling. Each task T is composed of computation over a tile of data and may be enclosed in multiple inter-tile loop levels (e.g., tiling for cache reuse). At each level, we must account for both computation and communication (i.e., data transfers).

We model the execution as a combination of:

— **Load:** reading data from off-chip memory (or previous task) into on-chip buffers,
— **Compute:** performing the actual tile computation,
— **Store:** writing the result back to off-chip memory (or to a subsequent task).

To enable communication/computation overlap, we use double-buffering or triple-buffering depending on the number of streams. At each inter-tile level n, the total latency is computed as the maximum of the three components, accounting for pipeline shifts and buffer reuse.

Level-Based Latency Recursion. Let:

— Lat_{n+1}: latency of executing the inner level $n+1$,
— $f_{a,n}$: number of bytes transferred for array a at level n,
— BS_a: memory bandwidth for array a,
— α: overlap factor (1 = load or store only, 2 = both).

Then, the latency at level n is:

$$\text{Lat}_n = \max\left(\text{Lat}_{n+1}, \frac{f_{a,n}}{BS_a}\right) + \text{Lat}_{n+1} + \alpha \cdot \frac{f_{a,n}}{BS_a}. \tag{14}$$

This formula captures both the initialization overhead and the overlap between communication and computation.

Base Case: Intra-Task Latency. The base latency Lat_{n+1} is the latency of the innermost computation tile, computed as in [61, 62].

$$\text{Lat}_{\text{intra}} = IL_{\text{par}} + IL_{\text{seq}} \cdot \log_2\left(\prod_{l \in \mathcal{L}_s^{\text{red}}} TC_{\text{intra}}^l\right). \tag{15}$$

If the computation is invoked across multiple tiles in a pipelined loop, the full latency becomes:

$$\text{Lat}_{\text{task}} = \text{Lat}_{\text{intra}} + II \cdot \left(\prod_{l \in \mathcal{L}_s^{\text{red}}} TC_{\text{inter}}^l - 1\right). \tag{16}$$

Final Task Latency. The full latency of a task T, denoted $\text{Lat}_{\text{task}}(T)$, is the latency at the outermost loop level $n = 0$, computed recursively using Equation 14, where the base is given by Equation 16.

5 Code Generation

Prometheus takes as input affine C/C++ code and automatically produces an HLS-C++ file, OpenCL host files, and all necessary files for code verification, RTL simulation, and bitstream generation. The NLP described in Section 4 gives the parameters of the space such as loop order, tiling factor, and so on. However, in order to be efficient, the code generation needs some specific rules.

5.1 Communication with Off-Chip Memory

To enable overlapping communication and computation, a *load* function transfers data on-chip using FIFOs, ensuring effective overlap management by the compiler. Furthermore, to guarantee that the *load* operation transfers data with the correct bit width, we automatically restructure the data in off-chip memory to enable sequential loading. Once data accumulates in the FIFO, a *read* function moves it to the shared data-tile buffer, whose size is determined by the NLP. For outputs, a *write* function transfers data from the buffer to a FIFO, and a *store* function writes it back to off-chip memory.

Listing 8 shows the load and read functions, which first transfer data from off-chip memory to an on-chip FIFO, and then read from the FIFO into the local buffer allocated for Task 1.

5.2 Communication between the Fused Task

The communication within the same fused task is not required as they use shared buffers. Similarly, for communication with off-chip memory, we choose to use FIFOs to facilitate communication between fused tasks, thereby simplifying the overlap. Listing 9 illustrates an example where the write function transfers data produced by Tasks 0 and 1 into a FIFO, which is then consumed by Task 5. As shown, the on-chip buffers used by Tasks 0 and 1 differ from those used by Task 5, allowing greater flexibility in choosing tile sizes for each task independently.

```
void load_vA_for_task1(hls::stream<float16> &fifo_A_from_off_chip_to_S1,
                       float16 vA[2340]) {
#pragma HLS inline off
  for (int i = 0; i < 2340; i++) {
#pragma HLS pipeline II = 1
    fifo_A_from_off_chip_to_S1.write(vA[i]);
  }
}

void read_A_FT0(float A[10][204],
                hls::stream<float16> &fifo_A_from_off_chip_to_S1, int i0) {
#pragma HLS inline off
  if (i0 >= 18) {
    return;
  }
  for (int d0 = 0; d0 < 10; d0++) {
    for (int d1 = 0; d1 < 204; d1 += 16) {
#pragma HLS pipeline II = 1
      float16 tmp_fifo = fifo_A_from_off_chip_to_S1.read();
      for (int j = 0; j < 16; j++){
#pragma HLS unroll
        if (d1 + j < 204)
            A[d0][d1 + j] = tmp_fifo[j];
      }
    }
  }
}
```

Listing 8. Memory transfer for array A in the 3 mm example (Listing 4). The data is first loaded from off-chip memory into a FIFO, then read from the FIFO to populate the on-chip buffer

5.3 Intra-Task

Each intra-task, corresponding to an intra-tile selected by the NLP, is implemented as a fully unrolled, independent function without communication with off-chip memory. Data for these tasks resides entirely on-chip.

If the task involves reduction loops, inter-tile reduction loops are integrated into the function and pipelined. While the pipeline initiation interval (II) is greater than one due to reduction dependencies, other operations in the statement are pipelined efficiently.

Padding is handled at the intra-tile level. Non-reduction loops remain unchanged, allowing computation of padding values without excessive resource usage. For reduction loops, full tiles are computed first, and the intra-tile loop is adjusted to handle padding for partial tiles accurately.

5.4 Inter-Tile Loop

For each fused task and each inter-tile loop, we generate independent functions to facilitate efficient double or triple buffering. Using information from the NLP, we determine which arrays are defined and which are transferred at each level of granularity. Once we identify the data being transferred, we implement double buffering if only reads or writes are involved, or triple buffering if both reads and writes are required. This strategy allows us to overlap the operations of reading, storing, and computing at the innermost level, optimizing both communication and computation overlap.

5.5 Concurrent Execution

Using the *dataflow* pragma, independent tasks execute concurrently without requiring manual rewrites. Computation in a receiving task begins as soon as its shared buffer contains sufficient data.

```
void write_E_FT0(float E[10][32],
                 hls::stream<float16> &fifo_E_from_task1_to_task5, int j0,
                 int i0) {
#pragma HLS inline off
  if (j0 < 0 || i0 < 0) {
    return;
  }
  for (int d0 = 0; d0 < 10; d0++) {
    for (int d1 = 0; d1 < 32; d1 += 16) {
#pragma HLS pipeline II = 1
      float16 tmp_fifo;
      for (j = 0; j < 16; j++){
#pragma HLS unroll
          tmp_fifo[j] = E[d0][d1 + j];
      }
      fifo_E_from_task1_to_task5.write(tmp_fifo);
    }
  }
}

void read_E_FT2(float E[180][192],
                hls::stream<float16> &fifo_E_from_task1_to_task5, int i0,
                int j0) {
#pragma HLS inline off
  if (j0 > 0 || i0 >= 18) {
    return;
  }
  for (int d1_0 = 0; d1_0 < 6; d1_0++) {
    for (int d0 = 0; d0 < 10; d0++) {
      for (int d1_1 = 0; d1_1 < 32; d1_1 += 16) {
        int d1 = d1_0 * 32 + d1_1;
        float16 tmp_fifo = fifo_E_from_task1_to_task5.read();
        for (j = 0; j < 16; j++){
#pragma HLS unroll
            if (d1 + j < 192)
                E[d0 + i0 * 10][d1 + 0 + j] = tmp_fifo[j];
        }
      }
    }
  }
}
```

Listing 9. *FIFO-Based Inter-Task Communication for Intermediate Matrix* E: The first function writes data from the local buffer E into a FIFO, while the second reads from the FIFO to populate the next computation stage's on-chip buffer.

5.6 SLR Management

The NLP determines the SLR ID for each task, specifying where it will be executed. Prometheus generates a separate C++ file for each SLR, and data transfers between SLRs are managed via *ap_axiu* streams, ensuring efficient communication and minimizing transfer overhead.

5.7 Design Regeneration

In cases where bitstream generation fails for a given design, we support automatic regeneration of the design. Several strategies can be applied, such as tightening resource constraints or reducing the maximum unrolling factor. By reducing the available resources, the design becomes smaller, which can help alleviate congestion. Thanks to the flexibility of our NLP-based approach, we can retain parts of the previous solution—such as the SLR assignment—and selectively restrict resources only for the specific task or group of tasks responsible for the congestion. This is the method we employ to regenerate designs when bitstream generation fails.

Table 5. Benchmark Kernels Used for Evaluation: Computational Complexity, Memory Requirements, Data Reuse, and Inter-Task Communication Analysis

Benchmark	Description	Ops Complexity	Mem Complexity	Reuse	Comm. Between Tasks
bicg	BiCG sub-kernel of BiCGStab solver	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$	$\mathcal{O}(1)$	0
madd	Matrix add. (C = A + B)	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$	$\mathcal{O}(1)$	0
mvt	Matrix Vector product and Transpose	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$	$\mathcal{O}(1)$	0
atax	Matrix transpose and vector mult.	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$	$\mathcal{O}(1)$	N
gesummv	Scalar, vector and matrix mult.	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$	$\mathcal{O}(1)$	$2N$
2-madd	2 Matrix add. (D = (A + B) + C)	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$	$\mathcal{O}(1)$	N^2
3-madd	3 Matrix add. (F = (A + B) + (C + D))	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$	$\mathcal{O}(1)$	$2N^2$
gemver	Vector mult. and matrix add.	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$	$\mathcal{O}(1)$	$2N^2 + 2N$
2mm	2 Matrix Mult. ($\alpha ABC + \beta D$)	$\mathcal{O}(N^3)$	$\mathcal{O}(N^2)$	$\mathcal{O}(N)$	N^2
gemm	Matrix-multiply ($C = \alpha AB + \beta C$)	$\mathcal{O}(N^3)$	$\mathcal{O}(N^2)$	$\mathcal{O}(N)$	N^2
syr2k	Symmetric rank-2k update	$\mathcal{O}(N^3)$	$\mathcal{O}(N^2)$	$\mathcal{O}(N)$	N^2
syrk	Symmetric rank-k update	$\mathcal{O}(N^3)$	$\mathcal{O}(N^2)$	$\mathcal{O}(N)$	N^2
trmm	Triangular matrix-mult.	$\mathcal{O}(N^3)$	$\mathcal{O}(N^2)$	$\mathcal{O}(N)$	N^2
3mm	3 Matrix Mult. ($(AB)(CD)$)	$\mathcal{O}(N^3)$	$\mathcal{O}(N^2)$	$\mathcal{O}(N)$	$2N^2$
symm	Symmetric matrix-mult.	$\mathcal{O}(N^3)$	$\mathcal{O}(N^2)$	$\mathcal{O}(N)$	$2N^2$

6 Evaluation

6.1 Setup

We evaluated our method using kernels from Polybench/C 4.2.1 [57] with medium-sized datasets and single-precision floating-point computations. The selected kernels represent both memory-intensive and computation-intensive scenarios. Medium problem sizes were chosen to balance demonstration of efficacy and the feasibility of time-consuming RTL evaluations. Due to Allo's lack of automatic code generation, we limited our experiments to the subset of PolyBench kernels provided in its artifact evaluation package [1], as results for other kernels were not available. We include a comparison with Sisyphus using the *n*-madd kernels, where *n* denotes the number of matrix additions. In the case of 2-madd, the result of the first addition is used as the input to the second. For 3-madd, the results of the first two additions are both used as inputs to the final addition.

Table 5 lists the benchmark kernels used in our evaluation. For each kernel, we report the computational complexity (in terms of the number of operations) and the memory complexity (in terms of footprint of the input/output data). Additionally, we include the data reuse order, which reflects the approximate number of times one data element is reused across different computations within the kernel. Kernels with reuse on the order of $\mathcal{O}(N)$, where N denotes the problem size (e.g., the number of rows or columns in an $N \times N$ input matrix), are typically considered compute-bound as their theoretical arithmetic intensity (assuming perfect reuse on-chip) is $\mathcal{O}(N)$. In contrast, kernels with reuse on the order of $\mathcal{O}(1)$ are generally classified as memory-bound. In practice, compute-bound kernels require very careful bufferization on-chip to minimize off-chip communications and achieve the theoretical arithmetic intensity. The final column (**Communication Between Tasks**) represents the number of data elements that are transferred between tasks in the dataflow design, excluding any initial input loading or setup overhead.

NLP problems were solved using the AMPL description language and the Gurobi solver (version 11.0.0) with the *qp:nonconvex=2* option for non-convex quadratic objectives and constraints. Evaluations included RTL simulation and on-board execution using the Alveo U55C FPGA, with a targeted

frequency of 220 MHz. RTL simulation provided accurate latency estimates, contrasting with the overly optimistic Vitis HLS reports that assume perfect task overlapping with dataflow pragma. The generated code from the frameworks are compiled using AMD/Xilinx Vitis HLS 2023.2 using the Vitis flow [4]. This flow assumes that data initially resides off-chip and has a default latency of 64 cycles to bring onto on-chip memory. All frameworks utilize "unsafe math" optimizations, enabling commutative/associative reduction operations at the expense of precision.

6.2 Experimental Evaluation

The objective of this evaluation is to demonstrate the capability of our framework to generate code with high QoR, whether for memory-bound or computation-bound kernels. To achieve this, we compare our work with Allo, ScaleHLS, Sisyphus, AutoDSE, and Stream-HLS.

For ScaleHLS [81], Allo [15], and Stream-HLS [9], their kernels assume that data is already present in on-chip memory. To ensure a fair comparison, we modified their code to incorporate off-chip to on-chip data transfers. However, Allo's 2 mm and 3 mm kernels already include this transfer mechanism, requiring no modifications. We left Sisyphus and AutoDSE unchanged, as they already optimize bit width according to the problem size. We use AutoDSE with the bottleneck method, setting a DSE timeout of 1,000 minutes and an HLS synthesis timeout of 180 minutes per task. For Allo-generated designs, we utilize kernels from their artifact repository since Allo does not employ a DSE [1]. Sisyphus generated designs are generated using parameters consistent with the article [62]. We observe some differences in Sisyphus results, as our evaluation is conducted on a different board. The design variations between boards may lead to differences in performance.

For RTL simulation, we assume that the frameworks can utilize all the resources on the U55C FPGA with a constraint of partitioning any array of 1,024 due to AMD/Xilinx limitations. For on-board FPGA evaluations, we consider two scenarios. The first scenario utilizes 60% of one SLR, equivalent to 20% of total board resources, for all frameworks. The second scenario is unique to our framework, leveraging all three SLRs, with 60% utilization per SLR. Most frameworks are not place-and-route aware, leading to congestion issues that prevent bitstream generation. Limiting frameworks to one SLR increases the likelihood of meeting timing requirements. AutoDSE, for instance, lacks dataflow support and applies pragmas within a single function, making multi-SLR bitstream generation impossible without manual intervention to split the function across SLRs. If congestion persists within an SLR, we manually adjust the NLP constraints based on the specific congestion issue and regenerate the HLS-C++ code, a process that typically takes only a few minutes. As our NLP model includes many constraints, both global and per SLR. If congestion occurs, we can address it by adjusting the relevant constraint. For example, if the issue is due to resource overutilization, we can retain the previous NLP solution and simply tighten the resource utilization constraint for the specific SLR. It is worth noting that different regeneration strategies are possible, depending on whether we prioritize achieving a high-quality result (QoR) or obtaining a solution quickly. For now, this decision is left to the user. This step is currently performed manually to better understand the root causes of congestion, but it could be automated through analysis of the compiler-generated log files.

It is worth noting that there is a difference in results between RTL simulation and on-board evaluation. This difference is expected, as RTL simulation involves fewer constraints, allowing for more aggressive unrolling and full utilization of available resources. In contrast, on-board evaluation is more complex, and excessive unrolling can easily lead to congestion. Therefore, we compare RTL simulation results with those of other frameworks to demonstrate our performance improvements. More importantly, we show that our approach generates code that can actually be implemented on board.

Table 6. Throughput Comparison (in GF/s) of PolyBench Kernels Across Frameworks Using RTL Simulation

Kernel	Ours	Sisyphus	ScaleHLS	Allo	AutoDSE	Stream-HLS
2 mm	308.38	195.09	37.13	46.58	0.41*	150.27
3 mm	368.36	178.97	43.04	60.40*	1.74*	174.75
Atax	3.56	2.32	1.58	1.96	1.97*	1.71
Bicg	15.41	2.32	1.70	14.17	0.99*	1.73
Gemm	419.14	227.09	40.53	37.50	110.81*	203.48
Gesummv	10.21	2.28	1.78	8.85	1.98*	1.72
Mvt	14.65	5.54	7.39	8.77	7.80*	13.31
Symm	212.20	200.30	0.06	16.72	14.68*	N/A
Syr2k	267.31	149.89	0.08	41.15	12.35*	N/A
Syrk	158.25	105.59	0.27	20.57	23.16*	N/A
Trmm	193.47	166.72	0.07	5.10	0.02*	N/A
PI (Avg)	1.00x	2.39x	927.20x	8.58x	973.14x	3.46x
PI (gmean)	1.00x	2.03x	48.03x	4.92x	25.82x	2.71x

6.3 Comparison

Table 6 presents the RTL simulation results for Prometheus, Sisyphus [62], AutoDSE [69], ScaleHLS [81], and Allo [15]. The last lines show the average and geometric mean performance improvement (PI) of Prometheus across evaluated kernels.

Results marked with ∗ are from the Vitis HLS report, showing minimum latency as an optimistic estimate. The RTL simulation for *3mm* with Allo was incomplete after two days. The N/A values in the table for Scale-HLS correspond to kernels that have at least one loop with a non-constant trip count. Stream-HLS does not handle these cases, so we cannot make a comparison.

Prometheus consistently achieves superior QoR across all evaluated kernels compared to other frameworks. While Sisyphus [62], designed primarily for computation-bound kernels, demonstrates competitive results for these kernels, it shows a weakness for *2 mm* and *3 mm*. This disparity is due to Sisyphus lacking concurrent execution capabilities for independent matrix multiplications. The performance advantage of Prometheus stems from both concurrent task execution and efficient overlapping of computation and communication.

Although for *bicg*, Allo and Prometheus do not use the same code transformation, the results are similar. Allo retains the original code structure by permuting the reduction loop outermost. The non-reduction loop is fully unrolled, and the reduction loop is pipelined. Prometheus partially unrolls both loops and pipelines the reduction loop.

Table 7 presents a comparison between Prometheus and Sisyphus on additional kernels. As shown, Prometheus generally achieves higher throughput and better overall resource utilization, except for BRAM usage, which is higher due to the use of double buffering. Notably, the 3-madd kernel shows a significant performance gain, as it enables concurrent execution of independent tasks. While similar benefits are also observed for 2 mm and 3 mm, the improvement is less pronounced. Despite achieving a 2.65x speedup on Mvt, we observe that Prometheus consumes more resources than Sisyphus. This is because Sisyphus attempts to unroll the loop with a large unroll factor and apply pipelining, but the compiler fails to achieve an initiation interval (II) of one and instead settles for II = 36. As a result, although they lose performance, the larger II significantly reduces the resource usage.

Table 7. RTL Evaluation of Performance and Resource Utilization for Sisyphus and Prometheus

	Sisyphus					Prometheus				
Kernel	GF/s	BRAM (%)	DSP (%)	FF (%)	LUT (%)	GF/s	BRAM (%)	DSP (%)	FF (%)	LUT (%)
Madd	1.96	1	29	22	38	2.71	3	7	10	11
2-madd	3.89	1	58	59	91	8.99	6	14	21	24
3-madd	3.91	2	58	75	117	14.00	9	21	31	37
2 mm	195.09	3	73	49	72	308.38	19	71	41	63
3 mm	178.97	16	98	86	122	368.36	20	83	52	85
Gemm	227.09	6	51	42	72	419.14	18	53	33	45
Gemver	17.93	24	116	56	68	22.51	31	53	34	54
Mvt	5.54	23	2	6	10	14.65	6	22	16	51

Table 8. On-Board Evaluation of Performance and Resource Utilization Across Frameworks and SLR Configurations

	Kernel	T (ms)	GF/s	DSP	BRAM	L(K)	FF(K)	F (MHz)
1 SLR Sisyphus	2 mm	1.20	30.57	556	510	213	276	220
	3 mm	1.52	29.89	984	611	230	300	220
	Atax	0.62	1.03	173	450	240	250	220
	Bicg	0.63	1.02	173	451	238	265	217
1 SLR AutoDSE	2 mm	92.25	0.40	963	353.5	287	292	205
	3 mm	110.34	0.41	1,117	470	278	306	220
	Atax	2.88	0.22	452	630.5	170	212	220
	Bicg	1.13	0.56	196	867.5	168	217	214
1 SLR Ours	2 mm	0.56	65.13	1,941	635.5	371	454	216
	3 mm	0.87	51.95	1,551	635.5	342	423	220
	Atax	0.24	2.62	1,081	533.5	234	287	184
	Bicg	0.15	4.04	732	311.5	250	302	220
3 SLR Ours	2 mm	0.29	125.54	2,752	546	428	549	220
	3 mm	0.34	134.07	4,379	600	684	840	207
	Atax	0.20	3.10	1,823	634.5	405	539	137
	Bicg	0.14	4.34	1,226	241	291	380	177

On Board Evaluation Table 8 presents the results for the on-board evaluation. The column $T(ms)$ indicates the kernel execution time in milliseconds, GF/s represents the throughput in Giga Floating Operations per second, and the resource usage is detailed in the thousands for both LUTs (L) and FFs. URAM utilization is excluded as no kernels use it. Column F (MHz) shows the frequency achieved by each design, with a target frequency of 220 MHz for all designs.

For *Bicg* and *Atax*, targeting 60% of resources on one SLR caused congestion, requiring regeneration with a 55% constraint. The *3 mm* bitstream from AutoDSE succeeded only with a 15% constraint.

Prometheus achieves a 77.16× performance improvement over AutoDSE, the SOTA model-free approach. This gain is accompanied by an average increase in resource utilization: 2.38x more DSPs, 1.08x more BRAM, 1.36x more LUTs, and 1.42x more FFs, reflecting the trade-offs made to achieve such high efficiency. When compared to Sisyphus, a recent NLP-based approach, Prometheus demonstrates a notable 2.59× performance boost, leveraging an average of 3.88× more DSPs, 1.04×

Table 9. Fusion, Loop Order and Data-tile Size Found by the NLP for the Kernel Evaluated on Table 8 for 1 SLR

	Fused Statement	Loop Order	Data Tile Size
2 mm	FT0: S0, S1, FT1: S2, S3	FT0: i,j,k, FT1: i,j,k	tmp(FT0): 4 × 32, B: 212 × 192, A: 4 × 212, D: 4 × 32, C:192 × 32, tmp(FT1): 4 × 192
3 mm	FT0: S0, S1, FT1: S2, S3, FT2: S4, S5	FT0: i,j,k, FT1: j,i,k FT2: j,i,k	E(FT0): 9 × 16, A: 9 × 200, B: 200 × 192, F(FT1): 10 × 10, C: 10 × 224, D: 224 × 10, G: 9 × 10, E(FT2): 9 × 192, F(FT2): 192 × 10,
Atax	FT0: S1, S2, FT1: S0, S3	FT0: i,j, FT1: j,i	tmp(FT0): 56, y: 16, A(FT0): 392 × 416, A(FT1): 392 × 16, tmp(FT1): 392
Bicg	FT0: S1, S2, FT1: S0, S3	FT0: j,i FT1: i,j	s: 16, A (FT0): 416 × 16, r: 416, q: 10, A (FT1): 10 × 400, p: 400,

more BRAM, 1.31× more LUTs, and 1.33× more FFs, illustrating its ability to deliver substantial improvements while effectively utilizing additional resources.

Table 9 shows the parameters found by the NLP. We use the same name of the iterator as Polybench 4.2.1 [57] code. *Si* represent the statement in position *i* in the code. The second column gives the statement fused inside the same tasks, the third column sets the fused task order and loop order found by the NLP for the fused task and the last column supplies the data-tile size found by the NLP, if the array is present in a different fused task the fused-task (defined in the second column) is precise.

Permutations are evident in the implementations of *3 mm*, *Atax*, and *Bicg*. For *2 mm* and *3 mm*, the NLP opted to fully transfer array B instead of overlapping computation and load, as the load operation had higher latency than computation. In *2 mm*, the NLP retains the original loop order. Array *tmp* is transferred from the first task to the second, with both tasks iterating over the first dimension using loop *i*. This enables a 4 × 32 data tile to be sent to the second task, which starts computation once a 4 × 192 tile of *tmp*(FT1) is filled.

As other frameworks cannot generate bitstreams for 3 SLRs without human intervention, we compare results for 1 and 3 SLRs using our framework. For *2 mm* and *3 mm*, performance improves due to increased resource utilization. However, for *atax* and *bicg*, where the bottleneck is memory transfer between off-chip and on-chip rather than parallelism, the improvement is negligible.

6.4 Scalability

Our NLP solver includes the option to set a timeout, returning the best design found so far without guaranteeing optimality. This feature enables efficient exploration of the solution space while ensuring adherence to time constraints when necessary.

When analyzing the time required to solve the NLP problem in Sisyphus, we observe that our framework, Prometheus, achieves similar solution times for most benchmarks. However, there is a notable exception: the 3 mm benchmark, which times out after four hours in Sisyphus, while Prometheus successfully finds a solution.

This efficiency stems from Prometheus' ability to explore a larger optimization space while still maintaining fast solution times. The key reason is that all optimization techniques are seamlessly integrated within the design space. This structured integration ensures that when a decision is made for one optimization parameter, it naturally constrains the possible choices for others, thereby reducing the search complexity.

In the case of 3 mm, which has six statements (including initialization), Sisyphus evaluates the product of all possible permutations since it relies on shared buffers. In contrast, Prometheus employs dataflow, which requires preserving the order of data between tasks that communicate

Table 10. Time Required (in seconds) for the NLP Solver to Find an Optimal Solution: Sisyphus vs. Prometheus Across Benchmarks (Timeout Set at 14,400s)

Benchmark	Sisyphus	Prometheus
2 mm	22.37	15.98
3 mm	14,400.08	21.37
Atax	3.18	7.03
Bicg	3.17	5.09
Gemm	2.19	4.31
Gesummv	1.57	8.07
Mvt	1.71	1.26
Symm	7.13	11.46
Syr2k	5.29	6.07
Syrk	2.56	3.94
Trmm	4.34	5.96
Average	1,313.96	8.23
Geo Mean	7.95	6.50

through FIFOs. As a result, although each statement could in principle lead to 2! or 3! loop permutations, many of these permutations become invalid under dataflow constraints when tasks must communicate via FIFOs. This substantially reduces the effective search space. Consequently, even with a broader search space, Prometheus converges efficiently to a solution in just 21.37 seconds.

A detailed comparison of solution times can be found in Table 10, which highlights the impact of our approach in managing the design space effectively.

7 Related Work

Recent years have seen the development of numerous frameworks and tools addressing the challenges of optimizing FPGA accelerators. While many tackle specific subproblems—such as pipelining, tiling, or dataflow—few propose holistic solutions integrating all stages of the transformation and scheduling pipeline. We categorize the most relevant related works according to key technical axes addressed by our framework: dataflow generation, shared-buffer optimization, code transformation, and tiling with padding. We also compare against general-purpose frameworks when applicable.

Dataflow – Dataflow principles have been extensively studied in models such as Kahn Process Networks [34], Dennis Dataflow [22], synchronous dataflow languages [12, 41], and for FPGA applications [3, 5, 11, 13, 15, 26, 53, 78]. DaCe [11] introduces Stateful DataFlow multiGraphs to separate program definition from optimization, enabling transformations like tiling and double-buffering, though requiring user intervention. Stream-HLS [9] automatically generates dataflow; however, it assumes that data are already on-chip, thereby overlooking communication with off-chip memory. Additionally, it offers very limited opportunities for parallelism. Flower [5] automates FPGA dataflow development but limits parallelism. Frameworks like [15, 78], built on HeteroCL [38], optimize data placement and compute scheduling for heterogeneous systems, maximizing data reuse and bandwidth utilization. Systolic arrays [10, 20, 33, 39, 73] offer efficient computation for specific patterns but lack generalization. Application-specific frameworks [6, 17, 21, 23, 29, 36, 51, 65] demonstrate dataflow advantages but do not generalize across domains. RapidStream-TAPA [27, 28] enhances the performance of dataflow designs and automates SLR placement. However, it requires an optimized kernel with a dataflow structure as input.

Shared Buffer – Shared buffer utilization through HLS has been extensively explored using methods such as NLP-based pragma insertion [60–62], bottleneck DSE [69], and GNN-based latency and resource estimation [8, 66–68, 70, 75, 77]. However, these approaches lack effective integration of dataflow optimization techniques.

Code Transformation – Code transformation has been explored for CPUs [7, 14, 37, 56], GPUs [72], and FPGAs [18, 43, 45–47, 59, 83, 84]. Pluto [14] minimizes communication and improves locality but can limit parallelism. FPGA-specific adaptations [59] leverage FIFOs for overlapping communication and computation but are restricted in parallelism. Recent works [83, 84] selectively use Pluto for latency minimization while avoiding non-HLS-friendly code. While [18, 43, 45–47] focus on optimizing pipelining techniques, they do not address parallelism or the coordination of computation and communication overlap, which are crucial for our objectives. The [15, 31, 38, 80, 81] compilers perform code transformations and pragma insertion, their modifications are primarily heuristic and based on loop properties. The article [62] described in Section 2 generates design with high QoR, but the absence of dataflow utilization hinders concurrent task execution. Additionally, their approach avoids padding, limiting the unroll factor to divisors of the loop's trip count and constraining tiling space.

Tiling and Padding – Tiling is essential for balancing computation and communication. While prior works [44, 48] use cost models to minimize communication, our approach extends this to reduce overall latency. Techniques like NLP-based tiling [44] focus on CPUs, while Wedler [63] optimizes DNNs on GPUs by fusing operators, enhancing data reuse, and employing padding to prevent bank conflicts. Padding is well-studied for reducing cache misses [30, 52] and improving memory transfers [74], but its use for varying unroll factors on FPGAs remains underexplored.

Discussion – In summary, while existing works address specific optimization aspects such as dataflow generation, pragma insertion, or tiling, they remain fragmented solutions. Prometheus distinguishes itself by integrating these axes into a unified optimization flow. In particular, it combines dataflow construction, shared-buffer optimization, and code transformation with tiling and padding in a single framework. Furthermore, Prometheus automates the generation of HLS-compatible C++ and OpenCL code and introduces SLR-aware scheduling. This holistic approach, together with public availability on GitHub, provides a significant advance over prior FPGA optimization frameworks.

8 Scopes of Application

Prometheus operates on affine C/C++ programs—a deliberate and powerful design choice. This class of programs captures a vast number of real-world, compute-intensive workloads found in domains such as dense linear algebra, signal processing, image filtering, and many AI kernels. These patterns are not only ubiquitous but also performance-critical in modern FPGA applications, and our evaluation covers typical patterns of computation and data reuse occurring in these computations.

Through MLIR [40] and emerging tools like Allo and Stream-HLS, our framework can seamlessly target programs written in higher-level languages such as Python. By translating these to affine MLIR representations and regenerating C/C++ code [50], we enable a broad spectrum of users—from ML practitioners to scientific programmers—to benefit from hardware acceleration without sacrificing optimization quality.

Affine computations can be *exactly* analyzed at compile-time, at the level of every operation and data element accessed [24, 25], leading to accurate analysis of loop bounds, memory access patterns, data dependencies, and data reuse. This enables precise performance modeling, aggressive design space pruning, and robust application of advanced HLS optimizations such as tiling, unrolling, pipelining, and memory partitioning—tasks that are notoriously difficult in general-purpose compilers or non-affine settings.

Extending to non-affine constructs is possible, but would require significant compromises. For instance, in the presence of indirect accesses like $A[B[i]]$, memory access patterns are no longer statically analyzable, preventing efficient tiling or partitioning. One must either fall back on high-latency off-chip access or resort to over-approximations, e.g., conservatively load entire arrays on-chip—approaches that often degrade parallelism or resource utilization. Our methodology remains applicable, but the design space becomes potentially much more limited due to superfluous constraints arising from over-approximations of the program description.

To achieve full automation and high-performance, our framework relies on the ability to distribute statements in loop nests into distinct loop nests, creating tasks for each generated loop nests afterwards. This property is required for maximal efficiency of our method, and fits typical dense linear algebra kernels, for example. But some applications may not expose such property: for example, the time-dependent stencil computations such as Jacobi-2D involve temporal dependencies that prevent full loop distribution, making them less amenable to end-to-end optimization within our current framework. However, the computation performed at each time step can still be optimized effectively using our methodology.

9 Conclusion

In this work, we introduced Prometheus, a holistic optimization framework that unifies loop transformations, task concurrency, computation-communication overlap, and hardware-aware scheduling for FPGA accelerators. By formulating the optimization process as a NLP problem, Prometheus enables a structured exploration of the vast design space while considering hardware constraints, memory bandwidth, and task parallelism.

Our framework addresses key limitations in existing methodologies, which often optimize only isolated aspects of FPGA acceleration. Prometheus surpasses these approaches by integrating SLR-aware scheduling, dynamic memory management, and hybrid execution models that effectively balance dataflow streaming and shared buffering. This enables more efficient utilization of FPGA resources, leading to significant improvements in latency, throughput, and scalability.

Through extensive performance evaluations on computation-bound kernels, we demonstrated that Prometheus outperforms SOTA frameworks. Furthermore, Prometheus' automatic DSE significantly reduces the manual effort required in FPGA development. By automatically generating HLS-C++ code, OpenCL host code, and FPGA bitstreams, the framework streamlines the deployment process, making FPGA acceleration more accessible to a broader range of applications.
In summary, Prometheus provides an innovative and effective approach to FPGA optimization, delivering high-performance solutions while minimizing the complexity of hardware design.

References

[1] 2024. Allo Artifact: Retrieved from https://github.com/cornell-zhang/allo-pldi24-artifact

[2] Miguel Á. Abella-González, Pedro Carollo-Fernández, Louis-Noël Pouchet, Fabrice Rastello, and Gabriel Rodríguez. 2021. PolyBench/python: benchmarking python environments with polyhedral optimizations. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction* (Virtual, Republic of Korea) *(CC 2021)*. Association for Computing Machinery, New York, NY, USA, 59–70. DOI: https://doi.org/10.1145/3446804.3446842

[3] Mariem Abid, Khaled Jerbi, Mickaël Raulet, Olivier Déforges, and Mohamed Abid. 2013. System level synthesis of dataflow programs: HEVC decoder case study. In *Proceedings of the 2013 Electronic System Level Synthesis Conference (ESLsyn)*. 1–6.

[4] AMD/Xilinx. 2024. *AMD/Xilinx Vitis 2023.2 Documentation*. Retrieved January 6, 2025 from https://docs.amd.com/r/en-US/ug1399-vitis-hls/Target-Flow-Overview

[5] Puya Amiri, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, Roland Leißa, and Sebastian Hack. 2021. FLOWER: A comprehensive dataflow compiler for high-level synthesis. In *Proceedings of the 2021 International Conference on Field-Programmable Technology (ICFPT)*. 1–9. DOI: https://doi.org/10.1109/ICFPT52863.2021.9609930

[6] Marco Bacis, Giuseppe Natale, Emanuele Del Sozzo, and Marco Domenico Santambrogio. 2017. A pipelined and scalable dataflow implementation of convolutional neural networks on FPGA. In *Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 90–97. DOI: https://doi.org/10.1109/IPDPSW.2017.44

[7] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 193–205.

[8] Yunsheng Bai, Atefeh Sohrabizadeh, Yizhou Sun, and Jason Cong. 2022. Improving GNN-based accelerator design automation with meta learning. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC'22)*. Association for Computing Machinery, New York, NY, USA, 1347–1350. DOI: https://doi.org/10.1145/3489517.3530629

[9] Suhail Basalama and Jason Cong. 2025. Stream-HLS: Towards automatic dataflow acceleration. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA'25)*. Association for Computing Machinery, New York, NY, USA, 103–114. DOI: https://doi.org/10.1145/3706628.3708878

[10] 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. DOI: https://doi.org/10.1145/3570928

[11] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N. Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC'19)*. Association for Computing Machinery, New York, NY, USA, Article 81, 14 pages. DOI: https://doi.org/10.1145/3295500.3356173

[12] Albert Benveniste, Paul Caspi, Paul Le Guernic, and Nicolas Halbwachs. 1994. Data-flow synchronous languages. In *Proceedings of the A Decade of Concurrency Reflections and Perspectives: REX School/Symposium Noordwijkerhout, The Netherlands June 1–4, 1993 Proceedings*. Springer, 1–45.

[13] Shuvra S. Bhattacharyya, Gordon Brebner, Jörn W. Janneck, Johan Eker, Carl von Platen, Marco Mattavelli, and Mickaël Raulet. 2009. OpenDF: A dataflow toolset for reconfigurable hardware and multicore systems. *SIGARCH Comput. Archit. News* 36, 5 (jun 2009), 29–35. DOI: https://doi.org/10.1145/1556444.1556449

[14] 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. DOI: https://doi.org/10.1145/1375581.1375595

[15] Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhichen Zeng, Mengjia Dai, and Zhiru Zhang. 2024. Allo: A programming model for composable accelerator design. *Proc. ACM Program. Lang.* 8, PLDI, Article 171 (June 2024), 28 pages. https://doi.org/10.1145/3656401

[16] W.Y. Chen, P.P. Chang, T.M. Conte, and W.W. Hwu. 1993. The effect of code expanding optimizations on instruction cache design. *IEEE Trans. Comput.* 42, 9 (1993), 1045–1057. DOI: https://doi.org/10.1109/12.241594

[17] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. 2018. SODA: Stencil with optimized dataflow architecture. In *Proceedings of the 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. DOI: https://doi.org/10.1145/3240765.3240850

[18] Young-kyu Choi and Jason Cong. 2018. HLS-based optimization and design space exploration for applications with variable loop bounds. In *Proceedings of the 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. DOI: https://doi.org/10.1145/3240765.3240815

[19] J. Cong, Z. Fang, Y. Hao, P. Wei, C. H. Yu, C. Zhang, and P. Zhou. 2018. Best-effort FPGA programming: A few steps can go a long way. *ArXiv*, abs/1807.01340 (2018).

[20] Jason Cong and Jie Wang. 2018. PolySA: Polyhedral-based systolic array auto-compilation. In *Proceedings of the 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. DOI: https://doi.org/10.1145/3240765.3240838

[21] Johannes de Fine Licht, Andreas Kuster, Tiziano De Matteis, Tal Ben-Nun, Dominic Hofer, and Torsten Hoefler. 2021. StencilFlow: Mapping large stencil programs to distributed spatial computing systems. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 315–326. DOI: https://doi.org/10.1109/CGO51591.2021.9370315

[22] J. B. Dennis. 1974. First version of a data flow procedure language. In *Proceedings of the Programming Symposium, Proceedings Colloque Sur La Programmation*. Springer-Verlag, Berlin, Heidelberg, 362–376.

[23] A. Denzler et al. 2023. Casper: Accelerating stencil computations using near-cache processing. In *IEEE Access* 11 (2023), 22136–22154. DOI: 10.1109/ACCESS.2023.3252002

[24] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *Int. J. Parallel Program.* 20, 1 (1991), 23–53.

[25] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. Part II. multidimensional time. *Int. J. Parallel Program.* 21, 6 (1992), 389–420.

[26] Paul Grigoraş, Xinyu Niu, Jose G. F. Coutinho, Wayne Luk, Jacob Bower, and Oliver Pell. 2013. Aspect driven compilation for dataflow designs. In *Proceedings of the 2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*. 18–25. DOI: https://doi.org/10.1109/ASAP.2013.6567545

[27] Licheng Guo, Yuze Chi, Jason Lau, Linghao Song, Xingyu Tian, Moazin Khatti, Weikang Qiao, Jie Wang, Ecenur Ustun, Zhenman Fang, Zhiru Zhang, and Jason Cong. 2023. TAPA: A scalable task-parallel dataflow programming framework for modern FPGAs with co-optimization of HLS and physical design. *ACM Trans. Reconfigurable Technol. Syst.* 16, 4, Article 63 (Dec. 2023), 31 pages. DOI: https://doi.org/10.1145/3609335

[28] Licheng Guo, Pongstorn Maidee, Yun Zhou, Chris Lavin, Eddie Hung, Wuxi Li, Jason Lau, Weikang Qiao, Yuze Chi, Linghao Song, Yuanlong Xiao, Alireza Kaviani, Zhiru Zhang, and Jason Cong. 2023. RapidStream 2.0: Automated parallel implementation of latency–insensitive FPGA designs through partial reconfiguration. *ACM Trans. Reconfigurable Technol. Syst.* 16, 4, Article 59 (Sept. 2023), 30 pages. DOI: https://doi.org/10.1145/3593025

[29] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. 2014. Darkroom: Compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph.* 33, 4, Article 144 (jul 2014), 11 pages. DOI: https://doi.org/10.1145/2601097.2601174

[30] Changwan Hong, Wenlei Bao, Albert Cohen, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, J. Ramanujam, and P. Sadayappan. 2016. Effective padding of multidimensional arrays to avoid cache conflict misses. *SIGPLAN Not.* 51, 6 (jun 2016), 129–144. DOI: https://doi.org/10.1145/2980983.2908123

[31] 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. DOI: https://doi.org/10.1109/TC.2021.3123465

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

[33] Liancheng Jia, Liqiang Lu, Xuechao Wei, and Yun Liang. 2020. Generating systolic array accelerators with reusable blocks. *IEEE Micro* 40, 4 (2020), 85–92. DOI: https://doi.org/10.1109/MM.2020.2997611

[34] Gilles Kahn. 1974. The semantics of a simple language for parallel programming. In *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974*, Jack L. Rosenfeld (Ed.). North-Holland, 471–475.

[35] Moazin Khatti, Xingyu Tian, Yuze Chi, Licheng Guo, Jason Cong, and Zhenman Fang. 2023. PASTA: Programming and automation support for scalable task-parallel HLS programs on modern multi-die FPGAs. In *Proceedings of the 2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 12–22. DOI: https://doi.org/10.1109/FCCM57271.2023.00011

[36] Guilherme Korol, Michael Guilherme Jordan, Mateus Beck Rutzig, and Antonio Carlos Schneider Beck. 2022. AdaFlow: A framework for adaptive dataflow CNN acceleration on FPGAs. In *Proceedings of the 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 244–249. DOI: https://doi.org/10.23919/DATE54114.2022.9774727

[37] Michael Kruse, Hal Finkel, and Xingfu Wu. 2020. Autotuning search space for loop transformations. In *Proceedings of the 2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*. IEEE, 12–22.

[38] 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. DOI: https://doi.org/10.1145/3289602.3293910

[39] Yi-Hsiang Lai, Hongbo Rong, Size Zheng, Weihao Zhang, Xiuping Cui, Yunshan Jia, Jie Wang, Brendan Sullivan, Zhiru Zhang, Yun Liang, Youhui Zhang, Jason Cong, Nithin George, Jose Alvarez, Christopher Hughes, and Pradeep Dubey. 2020. SuSy: A programming model for productive construction of high-performance systolic arrays on FPGAs. 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 73, 9 pages. DOI: https://doi.org/10.1145/3400302.3415644

[40] 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 *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. DOI: https://doi.org/10.1109/CGO51591.2021.9370308

[41] Edward Ashford Lee and David G. Messerschmitt. 1987. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.* C-36, 1 (1987), 24–35. DOI: https://doi.org/10.1109/TC.1987.5009446

[42] 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. DOI: https://doi.org/10.1145/3373087.3375320

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

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

[45] Junyi Liu, Samuel Bayliss, and George A. Constantinides. 2015. Offline synthesis of online dependence testing: Parametric loop pipelining for HLS. In *Proceedings of the 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. 159–162. DOI: https://doi.org/10.1109/FCCM.2015.31

[46] Junyi Liu, John Wickerson, Samuel Bayliss, and George A Constantinides. 2017. Polyhedral-based dynamic loop pipelining for high-level synthesis. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* 37, 9 (2017), 1802–1815.

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

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

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

[50] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to polyhedral MLIR. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (Virtual Event) *(PACT'21)*. Association for Computing Machinery, New York, NY, USA, 12 pages.

[51] Giuseppe Natale, Giulio Stramondo, Pietro Bressana, Riccardo Cattaneo, Donatella Sciuto, and Marco D. Santambrogio. 2016. A polyhedral model-based framework for dataflow implementation on FPGA devices of iterative stencil loops. In *Proceedings of the 2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. DOI: https://doi.org/10.1145/2966986.2966995

[52] P.R. Panda, H. Nakamura, N.D. Dutt, and A. Nicolau. 1999. Augmenting loop tiling with data alignment for improved cache performance. *IEEE Trans. Comput.* 48, 2 (1999), 142–149. DOI: https://doi.org/10.1109/12.752655

[53] Francesco Peverelli, Marco Rabozzi, Emanuele Del Sozzo, and Marco D. Santambrogio. 2018. OXiGen: A tool for automatic acceleration of c functions into dataflow FPGA-based kernels. In *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 91–98. DOI: https://doi.org/10.1109/IPDPSW.2018.00023

[54] PoCC. 2009. *PoCC, the Polyhedral Compiler Collection 1.3*. Retrieved from http://pocc.sourceforge.net

[55] Polybench. 2003. PolyBench/C: the Polyhedral Benchmark suite. Retrieved from http://tinyurl.com/m7ztgex

[56] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. 2011. Loop transformations: Convexity, pruning and optimization. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL'11)*. ACM, New York, NY, USA, 549–562. DOI: https://doi.org/10.1145/1926385.1926449

[57] Louis-Noel Pouchet and Tomofumi Yuki. Polybench: The polyhedral benchmark suite. Retrieved from http://polybench.sourceforge.net

[58] Louis-Noël Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. 2013. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the 21st ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'13)*. ACM Press, Monterey, California.

[59] 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. DOI: https://doi.org/10.1145/2435264.2435273

[60] Stéphane Pouget, Louis-Noël Pouchet, and Jason Cong. 2024. Automatic hardware pragma insertion in high-level synthesis: A non-linear programming approach. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey,CA,USA) *(FPGA'24)*. Association for Computing Machinery, New York, NY, USA, 184. DOI: https://doi.org/10.1145/3626202.3637593

[61] Stéphane Pouget, Louis-Noël Pouchet, and Jason Cong. 2025. Automatic hardware pragma insertion in high-level synthesis: A non-linear programming approach. *ACM Trans. Des. Autom. Electron. Syst.* 30, 2, Article 26 (Feb. 2025), 44 pages. DOI: https://doi.org/10.1145/3711847

[62] Stéphane Pouget, Louis-Noël Pouchet, and Jason Cong. 2025. A unified framework for automated code transformation and pragma insertion. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA'25)*. Association for Computing Machinery, New York, NY, USA, 187–198. DOI: https://doi.org/10.1145/3706628.3708873

[63] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. 2023. Welder: Scheduling deep learning memory access via tile-graph. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 701–718. Retrieved from https://www.usenix.org/conference/osdi23/presentation/shi

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

[65] Gagandeep Singh, Dionysios Diamantopoulos, Christoph Hagleitner, Juan Gomez-Luna, Sander Stuijk, Onur Mutlu, and Henk Corporaal. 2020. NERO: A near high-bandwidth memory stencil accelerator for weather prediction modeling. In *Proceedings of the 2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. 9–17. DOI: https://doi.org/10.1109/FPL50879.2020.00014

[66] Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong. 2022. Automated accelerator optimization aided by graph neural networks. In *Proceedings of the 2022 59th ACM/IEEE Design Automation Conference (DAC)*.

[67] Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong. 2022. Automated accelerator optimization aided by graph neural networks. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC'22)*. Association for Computing Machinery, New York, NY, USA, 55–60. DOI: https://doi.org/10.1145/3489517.3530409

[68] Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong. 2023. Robust GNN-based representation learning for HLS. In *Proceedings of the 2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. 1–9. DOI: https://doi.org/10.1109/ICCAD57390.2023.10323853

[69] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. 2021. AutoDSE: Enabling software programmers design efficient FPGA accelerators. In *Proceedings of 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. DOI: https://doi.org/10.1145/3431920.3439464

[70] Ecenur Ustun, Chenhui Deng, Debjit Pal, Zhijing Li, and Zhiru Zhang. 2020. Accurate operation delay prediction for FPGA HLS using graph neural networks. In *Proceedings of the 2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–9.

[71] Sven Verdoolaege. 2011. Counting affine calculator and applications. In *Proceedings of the 1st International Workshop on Polyhedral Compilation Techniques (IMPACT'11), Chamonix, France*.

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

[73] Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A polyhedral compiler for high-performance systolic arrays on FPGA. In *Proceedings of the 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA'21)*. Association for Computing Machinery, New York, NY, USA, 93–104. DOI: https://doi.org/10.1145/3431920.3439292

[74] Yuxin Wang, Peng Zhang, Xu Cheng, and Jason Cong. 2012. An integrated and automated memory optimization flow for FPGA behavioral synthesis. In *Proceedings of the 17th Asia and South Pacific Design Automation Conference*. 257–262. DOI: https://doi.org/10.1109/ASPDAC.2012.6164955

[75] N. Wu, Y. Xie, and C. Hao. 2022. Iron Man-Pro: Multi objective design space exploration in HLS via reinforcement learning and graph neural network-based modeling. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 3 (2022), 900–913. DOI: 10.1109/TCAD.2022.3185540

[76] 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 Trans. Comput.-Aided Des. Integr. Circuits Syst.* 42, 3 (2023), 900–913. DOI: https://doi.org/10.1109/TCAD.2022.3185540

[77] Nan Wu, Hang Yang, Yuan Xie, Pan Li, and Cong Hao. 2022. High-level synthesis performance prediction using GNNs: Benchmarking, modeling, and advancing. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC'22)*. Association for Computing Machinery, New York, NY, USA, 49–54. DOI: https://doi.org/10.1145/3489517.3530408

[78] Shaojie Xiang, Yi-Hsiang Lai, Yuan Zhou, Hongzheng Chen, Niansong Zhang, Debjit Pal, and Zhiru Zhang. 2022. HeteroFlow: An accelerator programming model with decoupled data placement for software-defined FPGAs. 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, 78–88. DOI: https://doi.org/10.1145/3490422.3502369

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

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

[81] 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. DOI: https://doi.org/10.1145/3489517.3530631

[82] Weichuang Zhang, Jieru Zhao, Guan Shen, Quan Chen, Chen Chen, and Minyi Guo. 2024. An optimizing framework on MLIR for efficient FPGA-based accelerator generation . In *Proceedings of the 2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 75–90. DOI: https://doi.org/10.1109/HPCA57654.2024.00017

[83] R. Zhao and J. Cheng. 2021. Phism: Polyhedral high-level synthesis in MLIR. *ArXiv* abs/2103.15103 (2021).

[84] R. Zhao, J. Cheng, W. Luk, and G. A. Constantinides. 2022. POLSCA: Polyhedral high-level synthesis with compiler transformations. *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. Belfast, United Kingdom. 235–242. DOI: 10.1109/FPL57034.2022.00044

Received 6 April 2025; revised 23 July 2025; accepted 5 September 2025