2026 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) | 979-8-3315-9288-2/26/$31.00 ©2026 IEEE | DOI: 10.1109/CGO68049.2026.11395206

Selene: Cross-Level Barrier-Free Pipelining for Irregular Nested Loops in High-Level Synthesis

Sungwoo Yun
Yonsei University
Seoul, Republic of Korea
sungwoo@yonsei.ac.kr

Seonyoung Cheon
Yonsei University
Seoul, Republic of Korea
seonyoung@yonsei.ac.kr

Dongkwan Kim
Yonsei University
Seoul, Republic of Korea
dongkwan@yonsei.ac.kr

Heelim Choi
Yonsei University
Seoul, Republic of Korea
heelim@yonsei.ac.kr

Kunmo Jeong
Yonsei University
Seoul, Republic of Korea
kunmo@yonsei.ac.kr

Chan Lee
Yonsei University
Seoul, Republic of Korea
chan.lee@yonsei.ac.kr

Yongwoo Lee
DGIST
Daegu, Republic of Korea
yongwoo@dgist.ac.kr

Hanjun Kim
Yonsei University
Seoul, Republic of Korea
hanjun@yonsei.ac.kr

***Abstract*—The growing demand for domain-specific accelerators in fields such as machine learning, graph analytics, and scientific computing has highlighted the need for productive and efficient hardware design methodologies. High-level synthesis (HLS) offers an attractive solution by generating hardware from high-level code, with loop pipelining as a cornerstone for maximizing throughput in regular computations. However, existing static and dynamic HLS approaches fail to achieve high pipeline utilization for irregular loop nests characterized by data-dependent bounds, unpredictable memory access patterns, and loop-carried dependencies.**

To address the pipeline underutilization in irregular loop, this work proposes a new barrier-free pipeline architecture with a cross-level scheduling strategy and the corresponding HLS compiler Selene, that automatically synthesizes the proposed architecture. Our approach introduces a fine-grained pipeline controller and an outer-loop iteration interleaving mechanism, enabling concurrent execution of multiple outer-loop iterations and efficient handling of data dependencies. Implemented within a commercial HLS flow, Vitis HLS, Selene delivers significant speedups of 4.74× and 5.46× over both standard static and dynamic HLS tools on a range of irregular workload benchmarks, demonstrating its effectiveness in overcoming challenges to efficient hardware generation for data-dependent applications.

***Index Terms*—High-level Synthesis, Loop Pipelining, Irregular Nested Loops, Scheduling, Compilers**

I. Introduction

The increasing demand for domain-specific accelerators in fields such as machine learning [1]–[6], graph analytics [7]–[10], and scientific computing [11], [12] has created a pressing need for productive and efficient hardware design methodologies. To achieve high performance, these accelerators employ highly parallel specialized datapaths that exploit both task-level and fine-grained parallelism, but growing application diversity and specialization have increased design complexity and turnaround pressure. High-level synthesis (HLS) addresses these challenges by automatically translating high-level programming languages (e.g., C, C++) into hardware circuits, significantly improving design productivity and accessibility. Commercial tools such as AMD Vitis HLS [13] and Intel HLS Compiler [14] support rapid prototyping and architectural exploration, and have been successfully used to implement highly efficient FPGA accelerators for deep neural network inference [15]–[18], high-performance sparse matrix operations [19]–[21], and large-scale graph processing [22], [23].

Loop pipelining is widely regarded as the heart of HLS optimization, as it enables the concurrent execution of multiple loop iterations and delivers substantial performance benefits in hardware accelerators [24], [25]. While commercial HLS tools [13], [14] provide a variety of additional optimizations [26], such as loop unrolling and array partitioning, loop pipelining remains the primary technique for improving throughput. By allowing new iterations to enter the hardware pipeline before previous ones complete, loop pipelining overlaps computation across loop iterations and fully exploits operation-level parallelism in the design. This approach is most effective for perfect loops, which have regular structure, predictable loop bounds, and no loop-carried dependencies, allowing synthesized circuits to closely approach their theoretical performance limits [25].

Despite the effectiveness of loop pipelining for perfect loops, many real-world accelerator applications are dominated by irregular loop nests, where loop bounds, memory access patterns, or control flow depend on runtime data. This irregularity is pervasive in graph analytics, where the number of neighbors per node varies, and in sparse matrix computations, where nonzero elements are distributed unpredictably. Such dynamic behavior leads to data-dependent loop-carried dependencies, unpredictable memory accesses, and unbalanced iteration workloads. In these cases, existing HLS tools and methodologies struggle to extract sufficient parallelism, as pipeline initiation intervals grow and hardware stalls increase.

In irregular loop nests, pipeline utilization is often restricted by two key challenges: execution barriers at task boundaries and inner-loop data dependencies. Execution barriers can cause the pipeline to remain idle until all inner-loop computations for the current outer-loop iteration are complete, preventing the efficient overlap of work across iterations. Additionally, inner-loop dependencies (*e.g.,* accumulation) can increase the

Authorized licensed use limited to: Yonsei Univ. Downloaded on April 07,2026 at 10:18:45 UTC from IEEE Xplore. Restrictions apply.

initiation interval and directly reduce throughput, thereby limiting the benefits of pipelining. Dynamic HLS techniques [27]–[29] attempt to improve pipeline utilization in irregular loop nests, but still suffer from throughput loss due to either unresolved dependencies or increased critical path, limiting their effectiveness for data-dependent applications.

To address the pipeline underutilization, this work proposes a *barrier-free pipeline* with a *hybrid cross-level scheduling strategy*. The barrier-free pipeline architecture shifts from traditional approaches, which allocate and schedule work at the granularity of entire inner-loop tasks, to a method that promotes a single iteration of the inner loop as a task. This method enables a fine-grained pipeline controller to dispatch new outer-loop iterations. As soon as all inner-loop computations for a given outer-loop iteration have been initiated, the pipeline controller can immediately begin dispatching the next outer-loop iteration without waiting for all computations to fully complete. This fine-grained pipeline control fundamentally removes the structural barrier that has previously limited concurrency, thereby enabling overlapping computations within irregular loop nests. Moreover, the hybrid cross-level scheduling strategy minimizes the pipeline stalls caused by true data dependencies within the inner loop. The proposed scheduling keeps the pipeline active even under inner-loop delays by statically analyzing dependencies and interleaving outer-loop iterations. Rather than waiting for a single outer-loop task to fully resolve its inner-loop dependencies, the scheduler dynamically selects other ready outer-loop tasks for execution. This outer-loop interleaving effectively hides stalls, improves overall pipeline utilization, and sustains high throughput in irregular, data-dependent workloads.

Building on these ideas, we present Selene, an HLS framework that automatically synthesizes the proposed architecture. Selene features two core transformation components, namely, *barrier-free pipeline generator* and *cross-level static scheduler*. The pipeline generator promotes an inner-loop body to the outer-loop-level task and constructs a unified context-aware datapath, enabling the barrier-free dispatch of both inner-loop and outer-loop iterations. The *cross-level static scheduler* determines an initiation interval (II) through critical path analysis and constructs a control logic that interleaves the outer-loop iterations to effectively hide pipeline stalls.

We implemented Selene within a commercial HLS flow and evaluated it against static standard HLS, coarse-grained dynamic HLS [27], and fine-grained dynamic HLS [30] across ten representative benchmarks featuring irregular workloads. The evaluation results show that Selene achieves 4.74×, 4.07×, and 5.46× speedups over static, coarse-grained dynamic, and fine-grained dynamic HLS tools, respectively. For resource overhead, the LUT/FF usage increases by 3.21×, 1.93×, and 0.86×, and the DSP usage increases by 1.06×, 1.00×, and 2.09× over the same three HLS tools. These results demonstrate that Selene effectively overcomes challenges in generating efficient hardware for irregular loop nests.

The contributions of this work are:

- A new HLS framework, Selene, supports efficient hardware generation for irregular nested loops.
- A new barrier-free pipeline architecture for concurrent outer-loop execution without task-boundary synchronization.
- A new hybrid cross-level scheduling that minimizes pipeline stalls by interleaving outer-loop iterations.
- Selene compiler analysis and optimizations that support barrier-free pipelining and cross-level scheduling.

II. Background

A. High-Level Synthesis

High-level synthesis (HLS) tools such as Vitis HLS [13] and Intel HLS Compiler [14] systematically transform high-level programs into synthesizable hardware circuits through a series of structured compiler and hardware mapping steps [31]. The HLS process begins with *Intermediate Representation (IR) Generation* and *Program Dependence Analysis*, constructing a program dependence graph (PDG) [32] to capture data and control dependencies (*e.g.,* Figure 1). In the *Operation Scheduling* phase, algorithms such as list scheduling [31] assign each IR operation to a specific clock cycle, guided by dependency constraints, resource availability, and user directives such as pipelining or unrolling. This is followed by *Datapath and Control Logic Generation*, where it synthesizes the computational datapath and constructs the finite state machine (FSM) or equivalent control structure. At this stage, the compiler analyzes initiation intervals (II), the minimum number of clock cycles between initiating consecutive request for pipeline, as well as operation latencies, and dependencies to determine how states are sequenced and how operations are triggered. Finally, the process concludes with *RTL Code Generation*, emitting synthesizable Verilog or VHDL for implementation on FPGA or ASIC hardware.

B. Statically Scheduled HLS and Pipelining

Static scheduling in HLS refers to the compile-time determination of the exact timing and order of all operations in the input program. The HLS tool constructs a PDG to analyze all control and data dependencies, and then uses scheduling algorithms to assign each operation to a specific clock cycle. The strength of statically scheduled circuits is that they enable highly optimized control logic with minimal runtime overhead, leading to compact FSMs, efficient resource utilization, and predictable, easily analyzable timing behavior. The synthesis process considers all possible control-flow paths and hardwires this sequencing logic into the final circuit.

To realize this static schedule in hardware, HLS tools generate an FSM as the main control structure for the synthesized design. The FSM contains states corresponding to program points or groups of operations, and transitions are driven by the program's control flow. For simple for-loops, the FSM implements a sequence of states that iterates through loop bodies and updates the loop index, terminating when the loop condition fails. In the case of nested for-loops, the FSM tracks the counters and loop conditions for each level, managing entry, body execution, and exit for each nesting depth, resulting in

Authorized licensed use limited to: Yonsei Univ. Downloaded on April 07,2026 at 10:18:45 UTC from IEEE Xplore. Restrictions apply.

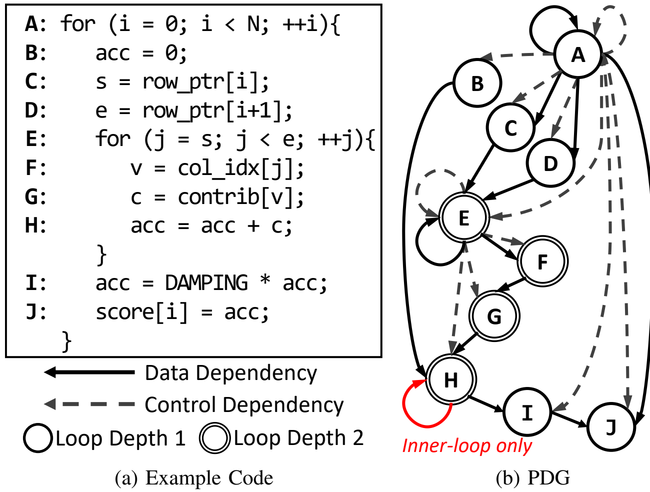


Fig. 1: Simplified PageRank [33] algorithm code and program dependence graph (PDG) example.

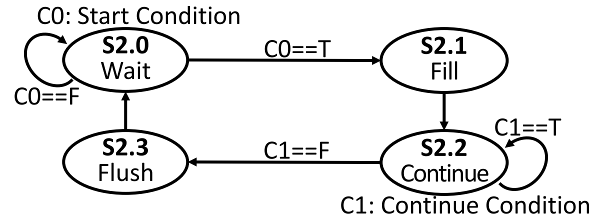


Fig. 2: The FSM of inner-loop pipeline controller for the example in Figure 1. The initial state for Stage 2 is `wait` state (S2.0).

a multi-dimensional state space that ensures correct ordering and data forwarding across iterations.

Loop pipelining is an HLS optimization that transforms the default sequential FSM to initiate new loop iterations before prior iterations have completed, enabling computation overlap and improving throughput. The HLS tool analyzes dependencies within the loop body to determine whether and how multiple iterations can safely be in flight simultaneously. The initiation interval (II), the minimum number of clock cycles required to start consecutive iterations, is calculated by the scheduler based on dependency and resource analysis. If dependencies allow, the pipelined FSM will issue new iterations every cycle (II=1); otherwise, the interval is increased to avoid hazards.

For a given program and PDG in Figure 1, a static HLS tool generates the pipeline shown in Figure 3a. Note that the for-loop nodes (`A,E`) are *strongly connected components* (SCC) that consist of loop headers (`i<N`, `j<e`) and latches (`++i`, `++j`). In the PDG, the compiler identifies the *loop control nodes* (`A` and `E`) by finding the cyclic control dependency, and the *loop body nodes* (`B,C,D,I,J` and `E,F,G`) by examining the control dependency from the control node. The compiler generates two different controllers for each control node.

For the operation scheduling, the compiler schedules independent SCCs (*e.g.,* `C,D`) in parallel, while placing dependent SCCs sequentially. The scheduling process analyzes data dependency cycles to identify the critical path and find II of the pipelined loop. The inner-loop II is decided by the loop-carried dependency of H, which takes two cycles. The outer loop cannot be pipelined due to an unpredictable stall from the inner loop. As a result, the final schedule initiates the inner-loop pipeline (`F,G,H`) every two cycles, and serializes the outer-loop as shown in Figure 3a.

Then, the HLS compiler generates the control logic based on the pipeline status and operation schedule. For example, the inner loop controller FSM (Figure 2) consists of `wait` (S2.0), `fill` (S2.1), `continue` (S2.2), and `flush` (S2.3) states. The *start condition* is a start signal from the outer loop controller, and *continue condition* is the inner-loop header condition. For scheduling the pipeline execution, the inner loop FSM (Figure 2) incorporates the II and latencies to each state, *e.g.,* `wait` (Cycle 0), `fill` (Cycle 1-2), `continue` (Cycle 3-6), and `flush` (Cycle 7-8) in Figure 3a.

The effectiveness of loop pipelining is fundamentally limited for irregular loops, such as those with data-dependent bounds and loop-carried dependencies that cannot be statically resolved. In such cases, the compiler cannot guarantee the safe overlap of iterations and must serialize execution or stall the pipeline, resulting in decreased throughput and poor hardware utilization. Figure 3a clearly shows that the inner-loop pipeline is not utilized during the outer-loop body execution.

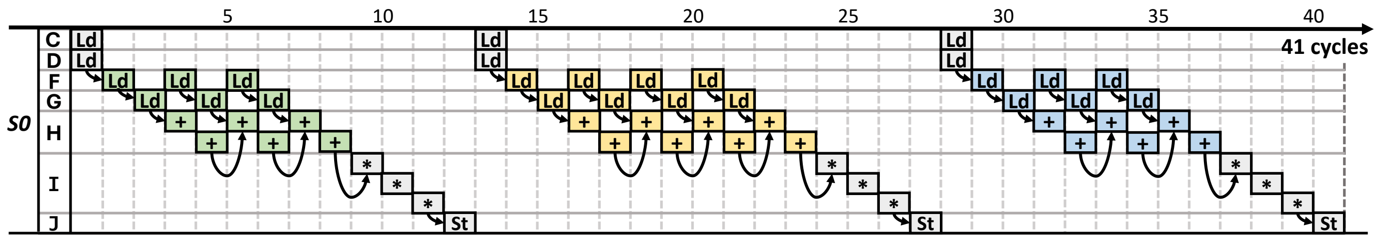
C. Dynamically Scheduled HLS

Dynamic scheduling [27], [28], [34], [35] approaches generate hardware where the execution order of operations or tasks is determined at runtime, rather than being fixed at compile time. At the heart of this paradigm is the dataflow circuit, in which independent operations or computation stages are implemented as separate hardware processes that communicate via FIFO queues. In such circuits, each operation or stage begins execution as soon as its input data is available, enabling fine-grained concurrency and natural load balancing without centralized control logic.

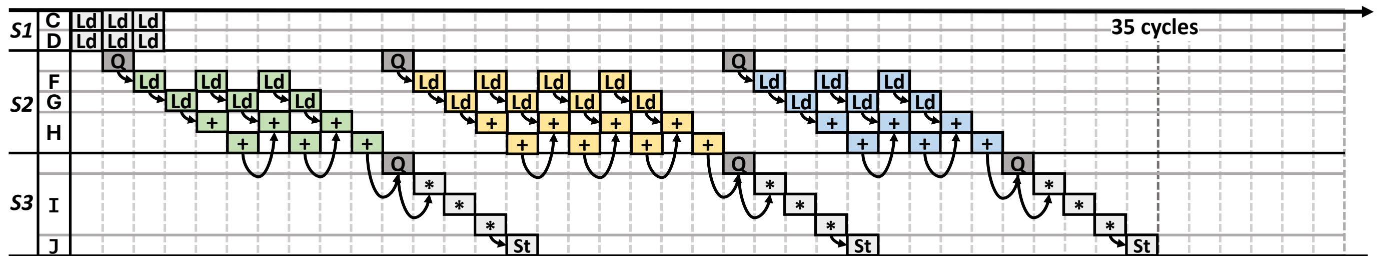
Most commercial HLS tools [13], [14] support task-level dataflow via explicit pragmas, such as `#pragma HLS dataflow`. In this approach, the hardware designer partitions the program into tasks or pipeline stages, typically at the function or loop level. The tool generates separate FSMs and datapaths for each stage by inserting FIFO queues to decouple their execution, allowing each stage to progress or stall independently. The queue separates the control of the inner loop from the outer loop, then the start condition (`C0`) of the inner-loop FSM (Figure 2) turns to *input queue is not empty*. While this method enables coarse-grained parallelism and can boost overall throughput for pipelined designs, it relies on manual partitioning and does not expose parallelism at the level of individual operations. All operations within each stage remain statically scheduled by the local FSM, limiting adaptability for highly irregular workloads.

Coarse-grained dynamic scheduling, such as coarse-grained pipeline architecture (CGPA) [34] and ElasticFlow [27], automates generation of dynamic dataflow circuits at the task

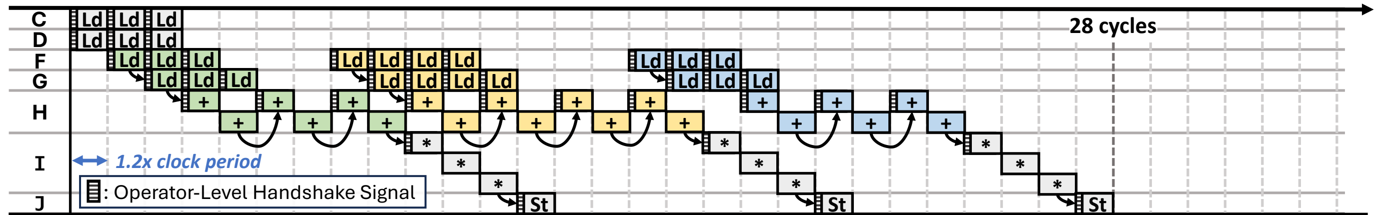
Authorized licensed use limited to: Yonsei Univ. Downloaded on April 07,2026 at 10:18:45 UTC from IEEE Xplore. Restrictions apply.



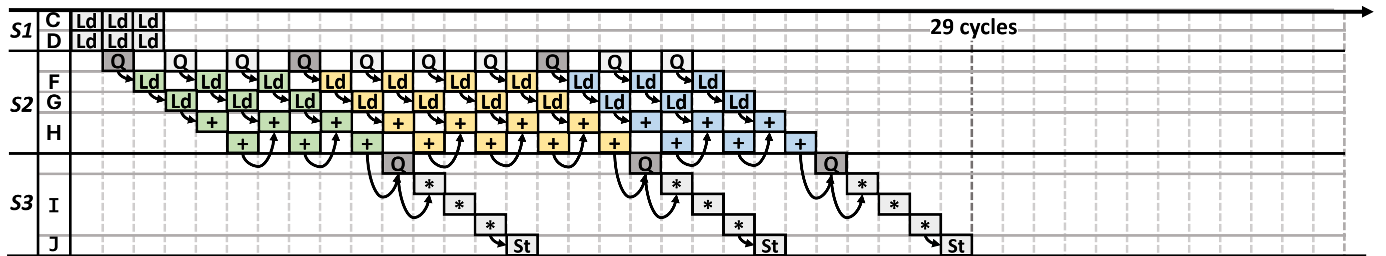
(a) Cycle count of the example code when executed using standard HLS.



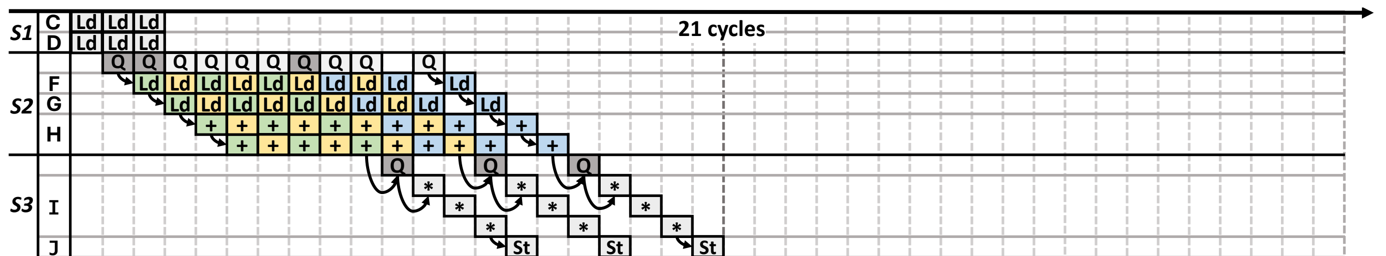
(b) Cycle count of the example code when executed using coarse-grained dataflow circuit [27].



(c) Cycle count of the example code when executed using fine-grained dataflow circuit (dynamatic [36]).



(d) Cycle count of the example code when executed using barrier-free pipelining.



(e) Cycle count of the example code when executed using barrier-free and cross-level scheduling.

Fig. 3: Execution cycle counts under different approaches for the example in Figure 1a. The leftmost column indicates the stage number, and the second column shows the instruction order from Figure 1a. `Ld` and `St` denote load and store instructions, respectively, while `Q` denotes a queue read. In subfigures (d) and (e), dark-colored `Q` corresponds to the outer loop, and light-colored `Q` to the inner loop. Blocks with the same color correspond to the same outer-loop iteration. Data dependencies are represented by black arrows. Here, additions (`H`) take two cycles, and multiplications (`I`) take three cycles.

Authorized licensed use limited to: Yonsei Univ. Downloaded on April 07,2026 at 10:18:45 UTC from IEEE Xplore. Restrictions apply.

level. CGPA automatically detects independent computational tasks and generates decoupled hardware modules connected by FIFO buffers. ElasticFlow extends this model for irregular loop nests by supporting dynamic buffering, adaptive pipeline depth, and superscalar scheduling which launches multiple tasks in parallel at each pipeline stage. These features allow both CGPA and ElasticFlow to dynamically balance workloads and improve hardware utilization for irregular applications.

Fine-grained dynamic scheduling achieves an even finer level of parallelism, as seen in frameworks like Dynamatic [28]. Here, the compiler treats each operation as an independent hardware process, with operation-level FIFOs or handshakes. Control is decentralized, enabling each operation to fire as soon as its inputs are ready, thereby maximizing pipeline utilization and naturally supporting irregular dataflow. While this approach excels for data-dependent and irregular workloads, it incurs significant hardware overhead from widespread buffering and distributed control, which can impact area and clock period compared to more traditional HLS techniques.

Fine-grained dynamic scheduling truly overlaps different outer-loop iterations by introducing operation-wise task queues (Figure 3c). Treating each operation as an independent stage with its own queue enables execution to proceed as soon as input operands are ready. This maximizes parallelism and can further increase pipeline utilization compared to the coarse-grained approach. However, this flexibility comes at a cost: every operation's queue adds datapath and control overhead, leading to significantly higher hardware resource usage and longer critical paths, resulting in a 1.2× increase in clock period as illustrated in Figure 3c.

III. Motivation

Dynamic scheduling techniques in HLS can effectively balance load and tolerate irregular task execution, but they are fundamentally limited by throughput loss resulting from pipeline underutilization due to task boundary barriers and loop-carried dependencies, or by increased datapath overhead from fine-grained queueing. To avoid the increased datapath overhead, this work analyzes the limitations of coarse-grained scheduling and proposes a balanced scheduling method between coarse-grained and fine-grained scheduling.

Limitation 1: Task boundary barrier prevents overlapping between tasks. Coarse-grained dynamic scheduling methods incur only modest hardware overhead due to their management and buffering at the task level; however, this coarse granularity incurs *task boundary barrier* that reduces the inner-loop pipeline utilization. Coarse-grained dynamic scheduling improves inner-loop pipeline utilization by enabling simultaneous execution of the inner-loop pipeline and the outer-loop body (Figure 3b). Unlike static scheduling (Figure 3a), dynamic HLS divides the computation into multiple stages (S1, S2, and S3) and inserts queues to dynamically feed inputs to each stage. This allows the inner-loop pipeline to begin processing a new outer-loop iteration (*e.g.,* yellow) immediately after the previous one (*e.g.,* green) completes. However, because the inner-loop controller schedules iterations independently from their associated outer-loop tasks, the system must still wait for the pipeline to flush before launching the next outer-loop iteration (S2.0 and S2.3 in Figure 2). As a result, a task boundary barrier remains, preventing full overlap of inner-loop work across different outer-loop iterations.

Limitation 2: Inner-loop-only parallelism leaves the pipeline underutilized. Existing dynamic scheduling approaches cannot maximize the inner-loop pipeline utilization, due to the true loop-carried dependency of inner-loop iterations. Although the barrier-free execution (Figure 3d) can overlap the new outer-loop iteration (*e.g.,* green) with the previous iteration (*e.g.,* yellow), they cannot fully utilize the inner-loop pipeline, because the accumulation on `acc` has a true loop-carried dependency. The loop-carried true dependencies, such as those arising from accumulations, can still limit the initiation interval (*e.g.,* II=2), thus reducing achievable pipeline utilization even with the barrier-free execution of fine-grained dynamic scheduling. With the interleaved cross-level scheduling (Figure 3e), the inner-loop pipeline could achieve near-optimal utilization (II$\approx$1) by utilizing the outer-loop task parallelism. However, existing approaches cannot schedule the new outer-loop iteration until the previous one is finished.

Research Goal: These analyses motivate two critical requirements for efficient HLS of irregular loop nests. First, there is a need for a *barrier-free pipeline architecture* (Figure 3d) that can enable full overlap of work across loop boundaries without incurring the substantial hardware cost of operation-wise queues. Such an approach would remove the task boundary barriers that limit utilization in coarse-grained dynamic scheduling, while avoiding the area and critical path overheads associated with fine-grained dataflow circuits. Second, effective *cross-level scheduling* of outer-loop iterations (Figure 3e) is required to further reduce the II and maximize pipeline utilization, even in the presence of true loop-carried dependencies. By intelligently interleaving and dispatching outer-loop iterations, the pipeline can remain busy and mitigate the performance impact of dependencies that would otherwise force idle cycles.

IV. Architecture Design

A. Proposed Architecture Overview

Figure 4 illustrates the architecture of the proposed barrier-free pipeline and cross-level scheduling for the example in Figure 1. To leverage the advantages of static scheduling and the mature optimizations available in commercial HLS tools, this work introduces a barrier-free pipeline architecture that can be realized using standard techniques such as task-level communication queues and function pipelining.

This work introduces a *dynamic context selector* (context multiplexer + loop control logic) and its associated control logic to implement the barrier-free pipeline. To eliminate task boundary barriers, the pipeline remains in `continue` even after completing the inner-loop task. By decoupling the loop control logic from the pipeline and generating a dedicated loop body pipeline, the design integrates the outer-loop context queue (original input queue) with the inner-loop context queues (*e.g.,* variables `j`, `s`, `e` in the for-loop). This integration

Authorized licensed use limited to: Yonsei Univ. Downloaded on April 07,2026 at 10:18:45 UTC from IEEE Xplore. Restrictions apply.

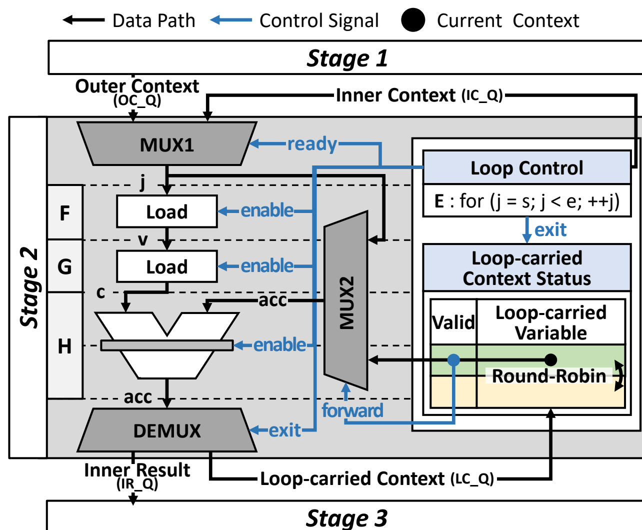


Fig. 4: Architecture of barrier-free pipeline with cross-level scheduling for the example code in Figure 1a.

allows the pipeline to treat both outer- and inner-loop contexts uniformly, thus removing the task boundary barriers associated with the `wait` and `flush` states in the FSM.

For cross-level scheduling, a dynamic context selector is designed to receive readiness control signals, enabling interleaved execution of different contexts while maintaining the FSM in `continue`. The enhanced selection logic evaluates the readiness of the inner-loop pipeline to determine when to dispatch a new outer-loop context. If `ready` is true, the logic dispatches an inner-loop context. Additionally, intermediate data involved in loop-carried dependencies (*i.e.,* loop-carried context) must be preserved during inner-loop execution. The loop-carried context status buffer automatically tracks the validity of such values, enabling correct interleaving and resolving loop-carried dependencies.

B. Barrier-free Pipelined Architecture

To support barrier-free execution, the architecture must dispatch the next outer-loop context immediately after the current inner-loop context completes, without requiring a full pipeline flush. To achieve this, the design incorporates three key architectural components: Context multiplexer and dual context queues, Loop-carried dependency forwarding, and Exit-driven context management.

Context multiplexer and dual context queues: To enable seamless progression across loop iterations without synchronization, the architecture uses separate context queues for the outer and inner loops. These queues serve as independent sources of execution context, allowing the controller to issue a new context immediately when the inner-loop iteration is finished. To ensure correct flow control, each queue uses a standard ready/valid protocol, where backpressure from a slower consumer automatically stalls its producer to prevent overflow. A datapath-level multiplexer selects between the two context sources based on the control signals from loop controller. This structure allows the pipeline to quickly switch to the next outer-loop context once the current inner-loop context has completed, without waiting for a full pipeline drain. The multiplexer is tightly integrated with the controller and is configured to ensure forward progress in the presence of dynamic loop bounds or variable latency conditions.

Loop-carried dependency forwarding: To support loop-carried dependencies within the inner loop, the architecture requires a dedicated mechanism to preserve and forward intermediate results across iterations. Specifically, each operator with a loop-carried dependency (*e.g.,* `H`) must be equipped with a forwarding path and a context validity checker. This checker determines whether the current pipeline context is active; if so, the input multiplexer for the operator selects the value propagated from the previous iteration (*e.g.,* `acc` from the pipeline). Crucially, this validity check must be performed at the same temporal granularity as the pipeline's initiation interval (*e.g.,* every 2 cycles for II=2). This is because contexts reside in the pipeline at that granularity, and any mismatch would lead to incorrect data forwarding or unnecessary stalls.

Exit-driven context management: Dynamic detection of the inner-loop exit condition is essential for avoiding pipeline stalls and ensuring correct iteration boundaries. Upon detecting the exit of the current context, the loop controller invalidates that context across the pipeline. This is implemented by deasserting the enable signals of all operators associated with that context, thereby preventing any further computation. Additionally, the context status register is updated to mark the context as inactive, allowing the system to accept a new outer-loop context when it becomes available. Importantly, this mechanism ensures that the first iteration of a new outer-loop context is computed using its initial value, not a forwarded value from a prior inner-loop context. As a result, in the loop controller's FSM shown in Figure 2, the start condition and continue condition correspond to the availability of an outer-loop context and a valid in-flight context, respectively.

C. Cross-level Scheduling

To support cross-level scheduling, the architecture needs to allow the interleaved scheduling of outer-loop iterations. To achieve this, we design three key architectural components: Readiness-aware context selector, Context status buffer, and Per-context control logic.

Readiness-aware context selector: The control unit first determines whether an inner-loop iteration is ready to be processed (already waited until II). If the inner-loop iteration is not ready because of task completion and dependency, the unit fetches an outer-loop context. If both outer- and inner-loop contexts are available at the same time, priority is given to the inner loop context to avoid delaying its completion, minimizing the turnaround time of the inner-loop execution. Furthermore, this design bounds the maximum in-flight context, preventing deadlocks from exhausted buffers.

Loop-carried context status buffer: When multiple outer-loop iterations share the pipeline, the values of loop-carried variables must be tracked independently for each context. A

Authorized licensed use limited to: Yonsei Univ. Downloaded on April 07,2026 at 10:18:45 UTC from IEEE Xplore. Restrictions apply.

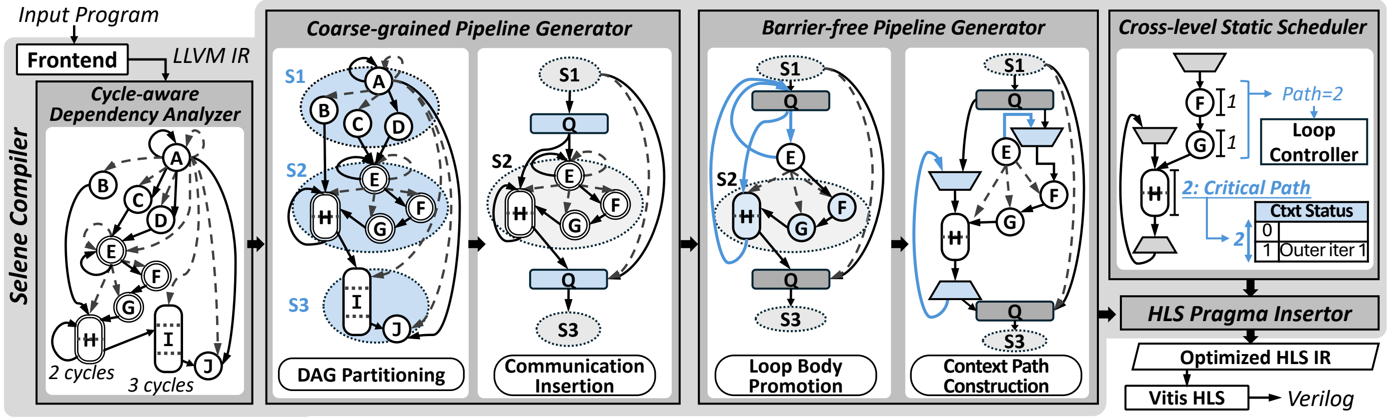


Fig. 5: Selene compiler design. Blue regions in each phase show modifications result from the applied transformation.

fixed-size context status buffer is used to store these values, with each entry representing a distinct in-flight outer-loop context. The number of concurrently active contexts is limited by the II of the inner-loop pipeline, because the context selector prioritizes the inner-loop context and the maximum in-flight context is bounded by II. Hence, the buffer only requires II entries. Adjusting the loop-carried context status buffer allows accurate tracking of when in-flight variables become available. For example, producing `acc` takes the latency of two cycles, and the II is the same, so the value can bypass the context buffer, then the entry could be eliminated.

Per-context control logic: The control unit issues control signals (*e.g.,* exit condition) for each pipeline stage by probing the status buffer in a round-robin manner. Since inner-loop contexts are scheduled at regular intervals (every II cycles), each inner-loop context deterministically arrives at the pipeline stage with fixed timing. This predictable behavior allows lightweight indexing logic to efficiently handle context-specific control without complex address computation.

V. Selene Compiler Framework

Selene compiler builds the proposed architecture (Section IV) from the input program. Figure 5 illustrates the overall compiler workflow, and Figure 6 presents the final output in HLS-C form, which is semantically equivalent to the IR produced after all the transformation passes.

A. Cycle-aware Dependency Analyzer

Selene compiler first performs dependency analysis to construct a PDG that captures both data and control dependencies across the operations. Then, the compiler marks the loop nesting level for each operation to help the subsequent optimizations. As shown in Figure 5, the generated PDG contains the loop nesting levels: nodes `A`, `B`, `C`, `D`, `I`, and `J` have a loop nesting depth of 1, and nodes `E`, `F`, `G`, and `H` have a nested depth of 2. Based on this PDG, the compiler conducts cycle analysis for each operation using target-specific device characteristics (*e.g.,* the latency of adder IP) and target clock period. Finally, the cycle-aware dependency analyzer produces a cycle-annotated PDG, which serves as the basis for subsequent pipeline scheduling step (*e.g.,* `H`: 2 cycles, `I` : 3 cycles).

B. Coarse-grained Pipeline Generator

To construct the coarse-grained pipeline, Selene performs DAG partitioning and communication insertion on the cycle-annotated PDG. In DAG Partitioning phase, the compiler detects and partitions the maximal chunk of pipelinable tasks. The pipelinable task consists of at most one control node with multiple *body* operations whose II is known. For example, `S1` and `S2` have a single control node, and `S3` has no control node. Inside their pipeline, all of the II of the operation is known. This approach allows for utilizing a static pipeline without incurring the overhead from excessive queueing.

Then, the compiler inserts communication by replacing inter-partition dependencies with explicit queues to transfer data between pipeline stages. Selene produces the coarse-grained pipelined loop, where each stage consists of a loop body wrapped with communication operations to interface with adjacent stages. For example, `S1` and `S3` are transformed into single-level loops, while `S2` becomes a two-level nested loop, with communication inserted into the pre-header and exit block of the inner loop.

C. Barrier-free Pipeline Generator

Coarse-grained pipeline stages that contain nested loops (*e.g.,* `S2`) introduce implicit barriers between outer-loop iterations due to their hierarchical control structure. Selene transforms such nested loops into barrier-free code through *Loop Body Promotion* and *Context Path Construction* passes.

Loop Body Promotion: As described in Section IV-B, the barrier-free pipeline requires the ability to selectively feed outer or inner loop contexts into the pipeline. Both outer and inner loop contexts must be exposed at the same loop level in the program to enable uniform handling and scheduling. Selene achieves this by promoting the inner loop body to the same level as the outer loop body, transforming the original nested loop into a flat loop where the induction variable of the inner loop can be dynamically selected. Selene performs loop body promotion through the following steps:

Authorized licensed use limited to: Yonsei Univ. Downloaded on April 07,2026 at 10:18:45 UTC from IEEE Xplore. Restrictions apply.

First, loop-carried dependencies in the inner loop are transformed into explicit queue-based communication. After loop promotion, the original control structure, including these dependencies, disappears from the program. To preserve the semantics, Selene inserts a push operation at each dependence source and a corresponding pop operation at its destination within the promoted inner-loop body. As shown in Figure 5, the loop-carried dependencies involving nodes `H` and `E` are redirected through the queue (`Q`) of the previous stage.

Second, the control flow of the inner loop is restructured by redirecting all back edges to a common post-dominator block and replacing the conditional branch at the loop header with an unconditional branch to the inner loop body. Since this removes the original control dependency from the loop header, Selene annotates all nodes that were originally control-dependent on the header with metadata that records this dependency. Additionally, Selene inserts a push operation to forward the exit condition into a queue in the previous stage, preventing it from being removed as dead code. This promotion process flattens the nested loop into single-level loops.

Context Path Construction: This pass decomposes the promoted loop body into pipeline stages and constructs explicit context paths that propagate data and control signals. The resulting structure is shown in Figure 6.

After promotion, the inner and outer loops execute within the same loop body, but still require distinct control behavior to preserve correctness. To ensure this, Selene converts operations that were control-dependent on the inner-loop header into predicated forms guarded by an *enable* signal (Line 22 and Line 30). This allows inner-loop operations to be conditionally executed under the appropriate control while coexisting with outer-loop operations in the same loop level.

Nodes with loop-carried dependencies (*e.g.,* `H`) require being isolated into separate loops. Since their execution depends on results from previous iterations, they must wait for a *forward* signal indicating the data is valid. Selene isolates such nodes into dedicated loops with their own control logic and connects to producer and consumer stages via FIFO queues (Line 26-33). If multiple loop-carried dependencies exist, Selene isolates each corresponding group of nodes into its own loop to control their execution independently. After isolation, their loop-carried dependencies are explicitly redirected: the corresponding push/pop operations are rewritten to use stage-local feedback queues (*e.g.,* `LC_Q`), and a phi-node (MUX) is inserted to select between the forwarded value and the newly produced value based on the forward signal (Line 29).

Although each node could be isolated into its own loop connected by queues, this would incur excessive FIFO overhead. To avoid this, Selene merges as many nodes as possible into a single loop while ensuring that the resulting stage-level dependency graph remains acyclic. Subsequently, Selene inserts communication operations for inter-stage dependencies, packing the corresponding data and control signals (Line 14, Line 21, Line 23 and Line 28). As a result, the program is transformed into a controllable pipeline composed of parallel loops, with each loop corresponding to a pipeline stage.

```
1  #pragma hls dataflow
2  // Stage1: A, B, C, D
3  for (i = 0; i < N; i++) {...}
4  // Stage2_1: E
5  while (...) {
6    #pragma hls pipeline II = 1
7    if (isReady()) { // MUX1
8      (j, acc) = OC_Q.pop(); // j = s, acc
9      fwd = 0;
10   } else {
11     (j, acc) = IC_Q.pop(); // loop-carried j, acc
12     fwd = 1; }
13   (enable, exit) = j<e ? (1, 0): (0, 1);
14   IS_Q1.push((j, acc), (fwd, enable, exit));
15   j++;
16   if (!exit) IC_Q.push(vSet);
17 }
18 // Stage2_2: F, G
19 while (...) {
20   #pragma hls pipeline II = 1
21   ((j, acc), (fwd, enable, exit)) = IS_Q1.pop();
22   if (enable) {v = col_idx[j]; c = contrib[v]; }
23   IS_Q2.push((acc, c), (fwd, enable, exit));
24 }
25 //Stage2_3: H
26 while (...) {
27   #pragma hls pipeline II = 1
28   ((acc, c), (fwd, enable, exit)) = IS_Q2.pop();
29   acc = fwd ? LC_Q.pop() : acc; // MUX2
30   if (enable) acc = acc + c;
31   if (exit) IR_Q.push(acc); // DEMUX
32   else LC_Q.push(acc);
33 }
34 // Stage3: I, J
35 while(...) {...}
```

Fig. 6: Output IR for the example in Figure 1a, presented in HLS-C form. Green-highlighted lines correspond to the original program. Queue names match those in Figure 4, where `IS_Q` denoting inter-stage queues within Stage 2.

D. Cross-level Static Scheduler

This pass constructs the cross-level control unit that orchestrates the execution of each pipeline stage. The resulting control logic appears in Figure 6 (Line 5-17).

To construct this control unit, Selene first analyzes the cycle-aware PDG to identify the critical path of the pipeline. The critical path is defined as the loop that contains the longest-latency loop-carried dependency (*e.g.,* `H`). Because this loop produces valid results once every critical-path cycle, the control unit should set and propagate signals at the same periodic interval to trigger the execution of subsequent stages. By propagating other contexts and their control signals in the idle cycles between these periodic triggers, different outer-loop contexts can be interleaved within the pipeline.

Selene generates the control unit as a single-loop pipeline stage. If there is an available execution slot for an outer loop iteration (the ready state in Line 7), the generated control unit newly fetches the data from the outer loop using an outer context queue (*e.g.,* `OC_Q`). If not, the control unit circulates in-flight inner loop contexts using a feedback queue (*e.g.,* `IC_Q`) for the loop-carried data, and sets the forward signals. The execution slot becomes available for a new outer loop iteration when an inner loop's exit condition arrives at the control unit.

Authorized licensed use limited to: Yonsei Univ. Downloaded on April 07,2026 at 10:18:45 UTC from IEEE Xplore. Restrictions apply.

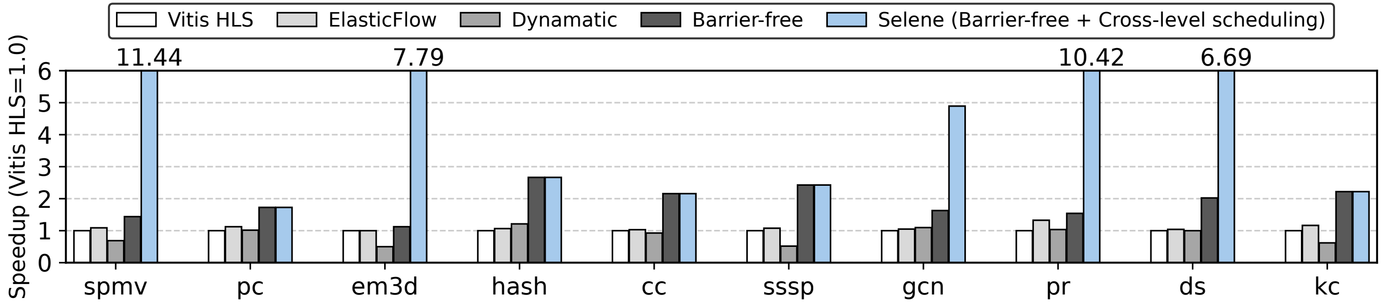


Fig. 7: Speedup of each scheme normalized to Vitis HLS.

TABLE I: Benchmark suite used in evaluation.

Benchmark	Domain	Description
spmv	Linear Algebra	Sparse matrix-vector multiplication
pc	General	Prefix count (histogram prefix sum)
em3d [37]	3D Simulation	Pointer-chasing over irregular structures
hash [38]	Database	Hash-based index construction
cc [39]	Graph	Connected components
sssp [39]	Graph	Single-source shortest path
gcn [40]	Graph	Graph convolution network aggregation
pr [39]	Graph	PageRank using pull-style updates
ds [41]	Graph	Dominating set construction
kc [42]	Graph	K-committee partitioning

TABLE II: Compile time in seconds.

	spmv	pc	em3d	hash	cc	sssp	gcn	pr	ds	kc
Vitis	20.9	21.1	21.4	20.9	20.9	21.3	21.2	21.3	21.4	20.7
Selene	35.0	36.2	36.2	34.3	38.6	39.8	37.0	39.4	38.1	34.3

The control unit sets the enable and exit signals based on the inner-loop exit conditions (*e.g.,* `j<e`), and issues the data to the inter-stage queues.

Finally, Selene inserts Vitis HLS-specific intrinsics into the LLVM IR to encode HLS directives such as `dataflow`, `pipeline`, and buffering pragmas, and then emits the annotated IR to the Vitis HLS LLVM backend, which lowers it into synthesizable Verilog for seamless integration with the commercial HLS flow.

VI. Evaluation

To evaluate Selene, we compared it against three HLS tools: (1) Vitis HLS, (2) ElasticFlow [27], and (3) Dynamatic [36], a dynamically scheduled HLS compiler. For ablation study, we compare a barrier-free pipeline without cross-level scheduling (Barrier-free) and with cross-level scheduling (Selene). We used Vitis HLS 2022.2 for synthesis and Vivado for RTL simulation and place-and-route analysis. All designs were synthesized for the Xilinx Kintex-7 FPGA (xc7k160tfbg484-1) with a target clock period of 5 ns. Functional correctness and cycle counts were measured via RTL simulation, and FPGA resource usage (LUTs, FFs, BRAMs, DSPs) was measured after place-and-route. Speedup is calculated based on the total execution time, which is obtained by multiplying the number of cycles with the target clock period. For designs which lead to timing violation, we calculated the execution time by multiplying its minimum clock period reported by Vivado.

We evaluated our approach using ten real-world applications from diverse domains, including linear algebra, 3D simulation [37], database [38], and graph analytics [39]–[42], which are irregular. Since Selene preserves the same scheduling behavior as Vitis HLS for regular workloads, this work evaluates the applications with irregular nested loops. The complete list of benchmarks is summarized in Table I. We used eight real-world datasets for graph benchmarks [43], [44]. To fit the limited on-chip resources of the target FPGA, we applied representative subgraph sampling using the Metropolis-Hastings Random Walk (MHRW) algorithm [45]. All experiments were performed on dual Intel Xeon Silver 4210 CPUs. Selene's compiler passes, excluding the downstream Vitis HLS synthesis, take at most 30 seconds per benchmark on this machine. The baseline Vitis HLS takes about 21 seconds. Table II shows the compile times for each benchmark.

A. Performance Improvement

Figure 7 shows the execution time of designs generated by Selene and other HLS tools. The barrier-free pipeline achieves a geometric mean speedup of 1.89× and 2.21× over the baseline and Dynamatic, respectively. This improvement is mainly due to the removal of task boundary barriers between outer-loop iterations, allowing the pipeline to remain active without waiting for the completion of previous iterations. The improvement depends on the characteristics of the inner loop and input data, particularly its pipeline depth and distribution of inner-loop bound. Benchmarks that feature short inner-loop bounds and long pipeline depths benefit the most from the barrier-free execution model, as overlapping outer-loop iterations without barrier can hide pipeline latency more effectively. Both `hash` and `sssp` exhibit larger improvements over `pc` due to their deeper pipelines, resulting in roughly 1.5× speedup. In contrast, benchmarks like `spmv`, `em3d`, `gcn`, `pr` and `ds` have long inner-loop pipelines but loop-carried dependencies in floating-point operations diminish the effect. These dependencies increase the initiation interval (II) to 7 or more, making the pipeline underutilized. As a result, the benefit of removing synchronization is less pronounced for these workloads, and the dominant performance bottleneck shifts to pipeline throughput limitations.

The barrier-free pipeline with cross-level scheduling outperforms Vitis HLS and Dynamatic by 4.74× and 5.46× on

Authorized licensed use limited to: Yonsei Univ. Downloaded on April 07,2026 at 10:18:45 UTC from IEEE Xplore. Restrictions apply.

TABLE III: Resource usage across benchmarks. Each Ratio shows the resource usage of Selene relative to Vitis. LUTs, FFs, and DSPs indicate the number of Look-Up Tables, Flip-Flops, and DSP blocks used; ElasticFlow and Dynamatic follow [27], [36]. The numbers in parentheses indicate usage as a percentage of total FPGA resources. All Block RAM usage is 0.

Benchmark	LUTs					DSPs					FFs				
	Vitis	[27]	[36]	Selene	Ratio	Vitis	[27]	[36]	Selene	Ratio	Vitis	[27]	[36]	Selene	Ratio
spmv	882	1129	2752	1730 (1.70%)	1.96×	14	14	9	14 (2.33%)	1.00×	1425	1534	2332	2948 (1.45%)	2.07×
pc	109	274	778	498 (0.49%)	4.57×	0	0	0	0 (0%)	–	126	185	788	665 (0.33%)	5.28×
em3d	946	1157	2795	1972 (1.94%)	2.08×	14	14	9	14 (2.33%)	1.00×	1421	1492	2413	4126 (2.03%)	2.90×
hash	59	262	1204	437 (0.43%)	7.41×	0	0	0	0 (0%)	–	188	233	1317	516 (0.25%)	2.74×
cc	86	192	487	277 (0.27%)	3.22×	0	0	0	0 (0%)	–	132	216	486	509 (0.25%)	3.86×
sssp	175	427	1341	592 (0.58%)	3.38×	0	0	0	0 (0%)	–	219	499	1484	1029 (0.51%)	4.70×
gcn	363	560	1433	1099 (1.08%)	3.03×	5	5	2	5 (0.83%)	1.00×	826	903	1332	1969 (0.97%)	2.38×
pr	401	741	1593	1197 (1.18%)	2.98×	5	7	2	7 (1.17%)	1.40×	803	1210	1540	2342 (1.15%)	2.92×
ds	302	469	1129	1136 (1.12%)	3.76×	2	2	0	2 (0.33%)	1.00×	649	721	1082	2181 (1.08%)	3.36×
kc	289	444	1197	791 (0.78%)	2.74×	0	0	0	0 (0%)	–	439	479	1144	1255 (0.62%)	2.86×

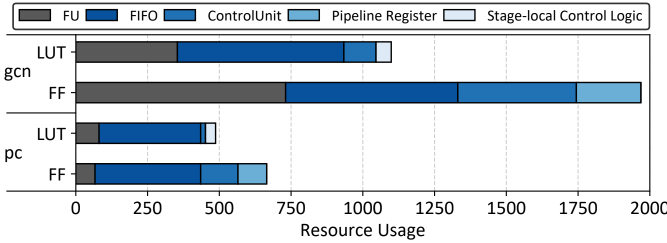


Fig. 8: Resource usage breakdown of `gcn` and `pc`.

average, respectively. The primary reason for this speedup is the improved utilization of inner-loop pipelines. Benchmarks such as `spmv`, `em3d`, `gcn`, `pr` and `ds` show more than 5× performance improvement due to their inner loops' large II. In these cases, overlapping outer-loop iterations allow the pipeline to stay fully utilized, significantly increasing throughput. On the other hand, for benchmarks like `pc`, `hash`, `cc`, `sssp`, and `kc`, cross-level scheduling achieves the same speedup compared to barrier-free pipeline. This is because their inner-loop pipelines already achieve II = 1 (without floating-point accumulation), leaving no room for outer-loop iterations to overlap.

Since Selene dynamically issues outer-loop iterations based on slot availability, it naturally maintains load balance even when the inner-loop bounds vary. In contrast, assigning iterations statically to fixed slots (as in cyclic scheduling) can lead to pipeline stalls due to load imbalance, as longer iterations delay the entire execution group. Across benchmarks with multiple slots such as `spmv`, `em3d`, `gcn`, `pr`, and `ds`, the static slot scheduling shows an average 1.55× slowdown in execution time compared to Selene.

B. Resource Overhead

Table III shows the FPGA resource usage of designs generated by each HLS compiler. LUT and FF usage represent the resource usage for the datapath and control logic without computation IPs like a floating-point adder. On average, LUT and FF usage increases by 3.21×, 1.93×, 0.86× over the static HLS, coarse- and fine-grained dynamic HLS tools, respectively. Figure 8 presents the breakdown of LUT and FF utilization for two representative benchmarks, `gcn` and `pc`. Excluding the functional units (FUs) required for the circuit's computation, the majority of the resources are consumed by the FIFO queues. Since Selene uses FIFO queues for inter-stage communication, the overhead inherently grows with the number of inter-stage dependencies. In addition, while Selene's design does not rely on stream-based communication, our implementation employs `hls::stream` for the control and data paths of the inner-loop pipeline, constrained by the Vitis HLS programming model. This overhead could be improved if the programming model natively supports the barrier-free architecture design.

Besides FIFO queues, Selene adds structural components that raise LUT and FF usage, including a centralized loop control unit and stage-local logic for control predication. Moreover, in the baseline design, the inner loop would be synthesized as a pipelined circuit nested inside a non-pipelined outer loop. In our design, however, the inner loop is promoted and synthesized with the outer loop as a fully pipelined circuit, which requires additional registers to preserve outer-loop context and increases the overall register footprint.

DSP usage of Selene is increased by 1.06×, 1.00×, and 2.09× over the static HLS, coarse- and fine-grained dynamic HLS tools, respectively. In general, the resource usage remains the same with the baselines. For some benchmarks (`pr` for Vitis and `spmv`, `em3d`, `gcn`, `pr`, and `ds` for Dynamatic), the DSP usage is increased. Selene uses more DSPs over Vitis HLS for `pr` because the pipelining privatizes DSPs that were shared between inner and outer loops. Dynamatic uses fewer DSPs for `gcn`, `pr`, and `ds` because its custom backend does not fully utilize DSPs and instead implements certain arithmetic operations using LUTs rather than DSPs. Although Selene uses more DSPs, the speedup dominates the DSP usage increases, and the throughput per DSP usage is increased.

VII. Related Work

HLS Loop Pipelining: HLS compilers have long used loop pipelining to overlap operations from successive iterations of a loop and thus improve throughput. Rotation scheduling [46] retimes operations across iterations to generate compact pipelines under dependency and resource constraints, targeting innermost loops with regular control flow. Morvan et al. [47] introduce flushing-enabled loop pipelining by generating schedules for

Authorized licensed use limited to: Yonsei Univ. Downloaded on April 07,2026 at 10:18:45 UTC from IEEE Xplore. Restrictions apply.

multiple IIs and handling dependency violations via runtime flushes, later extending this with dynamic hazard detection [48].

Coarse-grained Pipeline for Irregular Loops: To pipeline irregular nested loops, several coarse-grained approaches refactor computation into separate stages that communicate via streams or buffers, enabling outer-loop parallelism at a higher granularity. Polyhedral bubble insertion [49] analyzes iteration dependencies and injects bubbles to enforce correctness while allowing partial overlap of outer-loop iterations. Coarse-grained pipelined accelerators [34] apply a PS-DSWP-based strategy [50] to partition loop nests into decoupled pipeline stages. ElasticFlow [27] distributes outer-loop iterations to multiple Loop Processing Units (LPUs), overlapping task execution dynamically for irregular inner-loop workloads. Minutoli et al. [35] propose a dynamically scheduled architecture for graph query acceleration, using a task scheduler and hierarchical memory to enable out-of-order completion with concurrent memory accesses. Building on this, their Svelto [51] framework generalizes the idea into an HLS flow, combining spatial replication with fine-grained context switching to tolerate memory latency and synthesize multi-threaded accelerators for graph analytics.

Fine-grained Pipeline for Irregular Loop: Fine-grained dynamic scheduling abandons fixed schedules in favor of runtime decision-making. Josipović et al. [28] propose a fully latency-insensitive synthesis scheme that enables operation-level out-of-order execution, offering robustness for irregular and control-heavy workloads. Building on this, the same authors later introduce a method for buffer placement and sizing to further improve the throughput and timing closure of such dynamically scheduled circuits [52]. While powerful, these schemes tend to incur substantial area and control overhead. Hybrid models like DASS [53] attempt to statically schedule regular regions while leaving irregular ones dynamically scheduled, reducing control overheads.

Our approach adopts an intermediate design point: it preserves the benefits of static scheduling while improving pipeline throughput by overlapping outer-loop iterations through a compile-time transformation. This achieves much of the performance of dynamic scheduling while incurring only modest control and hardware overhead. Unlike coarse-grained pipeline accelerators that achieve performance by distributing work across multiple workers, Selene focuses on fine-grained scheduling within a single worker to fully utilize the pipeline.

Barrier-Free and Interleaved Execution Beyond HLS: The idea of relaxing loop iteration barriers to increase concurrency has also been explored in other domains. Elastic-Barriers [54] allow threads that reach synchronization early to advance to the next loop phase, mitigating idling while preserving inter-phase dependencies. PipeDream [55] and Megatron-LM [56] introduce an interleaved pipeline schedule that overlaps forward and backward passes across mini-batches, reducing pipeline idle time while maintaining optimizer semantics. WeiPipe [57] further proposes a weight-passing pipeline that overlaps computation to reduce bubbles and improve scalability for long-context language models.

VIII. Discussion

Applicability to Higher-order Loop Nests: While our current implementation targets two-level loop nests, our technique extends to deeper loop nests. An inner loop can be promoted to the level of its immediate enclosing loop as long as there are no loop-carried flow dependencies from the inner loop to the outer loop. By repeatedly applying this rule at each nesting level, Selene can be generalized to loop nests with more than two levels. The specific dependency conditions required for this generalization are discussed in the following section.

In terms of hardware cost, supporting higher-order loop nests is expected to be feasible without prohibitive overhead. The control logic can be implemented as a unified control unit that manages multiple nesting levels, so its complexity does not scale with the nesting depth. The number of FIFOs and the size of the context status buffer scale primarily with the number of loop-carried dependencies rather than the nesting depth itself. Pipeline registers will increase as the pipeline depth grows, as discussed in section VI-B. Overall, we expect that extending Selene to higher-order loop nests can be achieved without major structural or resource bottlenecks.

Limitations & Future Work: As a non-speculative solution, Selene does not permit loop-carried flow dependencies from the inner loop to the outer loop. Supporting such dependencies would require speculative execution and rollback mechanisms, which would incur additional hardware overhead. As future work, we plan to extend our framework to handle such complex dependencies with minimal overhead.

Currently, Selene considers only on-chip memory accesses within pipeline stages. Supporting larger datasets will require efficient interfaces for off-chip memory accesses. This aspect is orthogonal to our contribution, and can be combined with our framework. We leave it as future work.

IX. Conclusion

This work presents Selene, a novel HLS framework that overcomes the limitations of existing static and dynamic approaches for irregular, data-dependent loop nests. By combining a barrier-free pipeline architecture with hybrid cross-level scheduling, Selene enables fine-grained control over nested loop execution, reducing pipeline stalls and improving throughput. Across ten irregular benchmarks, Selene achieves speedups of up to 4.74× over static HLS tools and 5.46× over dynamic ones. These results demonstrate Selene's ability to deliver high performance with modest hardware overhead.

Acknowledgments

We thank the CoreLab members for their support and feedback during this work. We also thank the anonymous reviewers and the shepherd for their insightful comments and suggestions. This work is supported by No. RS-2025-02214497, No. RS-2024-00358765, No. RS-2023-00277060, and No. RS-2024-00395134 funded by the Ministry of Science and ICT. This work is also supported by SK hynix Inc. (Corresponding author: Hanjun Kim)

Authorized licensed use limited to: Yonsei Univ. Downloaded on April 07,2026 at 10:18:45 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[2] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, p. 1–12, Jun. 2017.

[3] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. IEEE Press, 2016, p. 243–254.

[4] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, "Going Deeper with Embedded FPGA Platform for Convolutional Neural Network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 26–35.

[5] A. Shawahna, S. M. Sait, and A. El-Maleh, "FPGA-based accelerators of deep learning networks for learning and classification: A review," *ieee Access*, vol. 7, pp. 7823–7859, 2018.

[6] A. Ejjeh, L. Medvinsky, A. Councilman, H. Nehra, S. Sharma, V. Adve, L. Nardi, E. Nurvitadhi, and R. A. Rutenbar, "HPVM2FPGA: Enabling true hardware-agnostic fpga programming," in *2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2022, pp. 1–10.

[7] S. Huang, K. Wu, H. Jeong, C. Wang, D. Chen, and W.-M. Hwu, "Pylog: An algorithm-centric python-based FPGA programming and synthesis flow," *IEEE Transactions on Computers*, vol. 70, no. 12, pp. 2015–2028, 2021.

[8] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, " HyGCN: A GCN Accelerator with Hybrid Architecture ," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Los Alamitos, CA, USA: IEEE Computer Society, Feb. 2020, pp. 15–29.

[9] G. Dai, Y. Chi, Y. Wang, and H. Yang, "FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 105–110.

[10] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[11] B. Betkaoui, Y. Wang, D. B. Thomas, and W. Luk, "A Reconfigurable Computing Approach for Efficient and Scalable Parallel Graph Exploration," in *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*, 2012, pp. 8–15.

[12] G. Sun, S. Kang, and S.-W. Jun, "BurstZ: a bandwidth-efficient scientific computing accelerator platform for large-scale data," in *Proceedings of the 34th ACM International Conference on Supercomputing*, ser. ICS '20. New York, NY, USA: Association for Computing Machinery, 2020.

[13] Advanced Micro Devices Inc., *Vitis High-Level Synthesis User Guide (UG1399)*, 2022. [Online]. Available: https://docs.amd.com/r/en-US/ug1399-vitis-hls

[14] Intel Inc., *Intel High Level Synthesis Compiler Pro Edition Reference Manual*, 2024. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683349/24-1/pro-edition-reference-manual.html

[15] S. Ghaffari and S. Sharifian, "FPGA-based convolutional neural network accelerator design using high level synthesize," in *2016 2nd International Conference of Signal Processing and Intelligent Systems (ICSPIS)*, 2016, pp. 1–6.

[16] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 45–54.

[17] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 161–170.

[18] Y. Zhou and J. Jiang, "An FPGA-based accelerator implementation for deep convolutional neural networks," in *2015 4th International Conference on Computer Science and Network Technology (ICCSNT)*, vol. 01, 2015, pp. 829–832.

[19] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on FPGAs," in *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, 2005, pp. 63–74.

[20] Y. Hu, Y. Du, E. Ustun, and Z. Zhang, "GraphLily: Accelerating Graph Linear Algebra on HBM-Equipped FPGAs," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021, pp. 1–9.

[21] Y. Du, Y. Hu, Z. Zhou, and Z. Zhang, "High-performance sparse linear algebra on hbm-equipped fpgas using hls: A case study on spmv," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2022, pp. 54–64.

[22] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "ForeGraph: Exploring large-scale graph processing on multi-FPGA architecture," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 217–226.

[23] X. Chen, H. Tan, Y. Chen, B. He, W.-F. Wong, and D. Chen, "ThunderGP: HLS-based graph processing framework on fpgas," in *Proc. ACM/SIGDA Int'l Symposium on Field-Programmable Gate Arrays (FPGA)*, 2021, pp. 12–22.

[24] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: a high-level synthesis framework for applying parallelizing compiler transformations," in *16th International Conference on VLSI Design, 2003. Proceedings.*, 2003, pp. 461–466.

[25] Y. Liang, K. Rupnow, Y. Li, D. Min, M. N. Do, and D. Chen, "High-Level Synthesis: Productivity, Performance, and Software Constraints," *Journal of Electrical and Computer Engineering*, vol. 2012, no. 1, p. 649057, 2012.

[26] J. de Fine Licht, M. Besta, S. Meierhans, and T. Hoefler, "Transformations of high-level synthesis codes for high-performance computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 5, pp. 1014–1029, 2020.

[27] M. Tan, G. Liu, R. Zhao, S. Dai, and Z. Zhang, "ElasticFlow: A complexity-effective approach for pipelining irregular loop nests," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015, pp. 78–85.

[28] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically Scheduled High-level Synthesis," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 127–136.

[29] S. Margerm, A. Sharifian, A. Guha, A. Shriraman, and G. Pokam, "TAPAS: Generating parallel accelerators from parallel programs," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 245–257.

[30] E. Logic and A. P. L. Laboratory, "Dynamatic: Dataflow-based High-Level Synthesis Compiler," https://github.com/EPFL-LAP/dynamatic/releases, 2024, accessed: 2025-05-30.

[31] D. D. Gajski, N. D. Dutt, A. C. Wu, and S. Y. Lin, *High—Level Synthesis: Introduction to Chip and System Design*. Springer Science & Business Media, 2012.

[32] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[33] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web." Stanford infolab, Tech. Rep., 1999.

[34] F. Liu, S. Ghosh, N. P. Johnson, and D. I. August, "CGPA: Coarse-Grained Pipelined Accelerators," in *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1–6.

Authorized licensed use limited to: Yonsei Univ. Downloaded on April 07,2026 at 10:18:45 UTC from IEEE Xplore. Restrictions apply.

[35] M. Minutoli, V. G. Castellana, A. Tumeo, M. Lattuada, and F. Ferrandi, "Efficient synthesis of graph methods: A dynamically scheduled architecture," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016, pp. 1–8.

[36] EPFL Processor Architecture Laboratory, "Dynamatic: From C/C++ to Dynamically-Scheduled Circuits," https://dynamatic.epfl.ch/.

[37] M. C. Carlisle and A. Rogers, "Software caching and computation migration in Olden," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 29–38.

[38] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the Walkers: Accelerating Index Traversals for In-memory Databases," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. ACM, 2013, pp. 468–479.

[39] S. Beamer, K. Asanović, and D. Patterson, "The GAP benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[40] T. N. Kipf and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks," 2017.

[41] N. Boria, C. Murat, and V. T. Paschos, "The probabilistic minimum dominating set problem," *Discrete Applied Mathematics*, vol. 234, pp. 93–113, 2018.

[42] S. Gupta and V. K. Nandivada, "IMSuite: A benchmark suite for simulating distributed algorithms," *Journal of Parallel and Distributed Computing*, vol. 75, no. 0, pp. 1 – 19, Jan. 2015.

[43] R. A. Rossi and N. K. Ahmed, "The Network Data Repository with Interactive Graph Analytics and Visualization," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015. [Online]. Available: http://networkrepository.com

[44] T. A. Davis and Y. Hu, "The university of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011.

[45] C. Hübler, H.-P. Kriegel, K. Borgwardt, and Z. Ghahramani, "Metropolis algorithms for representative subgraph sampling," in *2008 eighth ieee international conference on data mining*. IEEE, 2008, pp. 283–292.

[46] L.-F. Chao and A. LaPaugh, "Rotation scheduling: A loop pipelining algorithm," in *Proceedings of the 30th international Design Automation Conference*, 1993, pp. 566–572.

[47] S. Dai, M. Tan, K. Hao, and Z. Zhang, "Flushing-enabled loop pipelining for high-level synthesis," in *Proceedings of the 51st Annual Design Automation Conference*, 2014, pp. 1–6.

[48] S. Dai, R. Zhao, G. Liu, S. Srinath, U. Gupta, C. Batten, and Z. Zhang, "Dynamic hazard resolution for pipelining irregular loops in high-level synthesis," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 189–194.

[49] A. Morvan, S. Derrien, and P. Quinton, "Polyhedral bubble insertion: A method to improve nested loop pipelining for high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 3, pp. 339–352, 2013.

[50] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August, "Parallel-stage decoupled software pipelining," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, 2008, pp. 114–123.

[51] M. Minutoli, V. G. Castellana, N. Saporetti, S. Devecchi, M. Lattuada, P. Fezzardi, A. Tumeo, and F. Ferrandi, "Svelto: High-Level Synthesis of Multi-Threaded Accelerators for Graph Analytics," *IEEE Transactions on Computers*, vol. 71, no. 3, pp. 520–533, 2022.

[52] L. Josipović, S. Sheikhha, A. Guerrieri, P. Ienne, and J. Cortadella, "Buffer Placement and Sizing for High-Performance Dataflow Circuits," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 1, Nov. 2021.

[53] J. Cheng, L. Josipović, G. A. Constantinides, P. Ienne, and J. Wickerson, "DASS: Combining dynamic & static scheduling in high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 3, pp. 628–641, 2021.

[54] A. Tiwari and V. K. Nandivada, "Unleashing Parallelism with Elastic-Barriers," *ACM Trans. Archit. Code Optim.*, vol. 22, no. 2, Jul. 2025.

[55] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "PipeDream: Generalized pipeline parallelism for DNN training," in *Proceedings of the 27th ACM symposium on operating systems principles*, 2019, pp. 1–15.

[56] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro *et al.*, "Efficient large-scale language model training on gpu clusters using megatron-lm," in *Proceedings of the international conference for high performance computing, networking, storage and analysis*, 2021, pp. 1–15.

[57] J. Lin, Z. Liu, Y. You, J. Wang, W. Zhang, and R. Zhao, "WeiPipe: Weight Pipeline Parallelism for Communication-Effective Long-Context Large Model Training," in *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2025, pp. 225–238.

Authorized licensed use limited to: Yonsei Univ. Downloaded on April 07,2026 at 10:18:45 UTC from IEEE Xplore. Restrictions apply.